

Sequential Parameter Optimization Cookbook

A guide for scikit-learn and PyTorch

Thomas Bartz-Beielstein

Dec 20, 2025

Table of contents

Preface	1
I. Sequential Parameter Optimization Toolbox (SPOT)	3
1. SpotOptim Internal Methods Examples	5
1.1. Setup: Import Required Packages	5
1.2. 1. Main Optimization Method: <code>optimize()</code>	5
1.3. 2. Initial Design Methods	7
1.3.1. 2.1 <code>get_initial_design()</code>	7
1.3.2. 2.2 <code>_curate_initial_design()</code>	8
1.3.3. 2.3 <code>_handle_NA_initial_design()</code>	9
1.3.4. 2.4 <code>_check_size_initial_design()</code>	10
1.3.5. 2.5 <code>_get_best_xy_initial_design()</code>	10
1.4. 3. Surrogate Model Methods	11
1.4.1. 3.1 <code>_fit_surrogate()</code>	11
1.4.2. 3.2 <code>_predict_with_uncertainty()</code>	12
1.4.3. 3.3 <code>_acquisition_function()</code>	12
1.4.4. 3.4 <code>_suggest_next_point()</code>	13
1.5. 4. OCBA Method (Optimal Computing Budget Allocation)	14
1.5.1. 4.1 <code>_apply_ocba()</code>	14
1.6. 5. NaN/Inf Handling Methods	15
1.6.1. 5.1 <code>_apply_penalty_NA()</code>	15
1.6.2. 5.2 <code>_remove_nan()</code>	16
1.6.3. 5.3 <code>_handle_NA_new_points()</code>	16
1.7. 6. Main Loop Update Methods	18
1.7.1. 6.1 <code>_update_best_main_loop()</code>	18
1.8. 7. Termination Method	19
1.8.1. 7.1 <code>_determine_termination()</code>	19
1.9. 8. Utility Methods	20
1.9.1. 8.1 <code>_select_new()</code>	20
1.9.2. 8.2 <code>_repair_non_numeric()</code>	21
1.9.3. 8.3 <code>_map_to_factor_values()</code>	21
1.10. 9. Integration Examples	22
1.10.1. 9.1 Optimization with Custom Initial Design	22
1.10.2. 9.2 Optimization with Starting Point	23
1.10.3. 9.3 Optimization with Integer Variables	24

Table of contents

1.10.4. 9.4 Noisy Function Optimization with OCBA	25
1.10.5. 9.5 Optimization with NaN Handling	26
1.11. 10. Method Relationship Diagram	27
1.12. 11. Summary	28
1.13. Jupyter Notebook	29
2. Optimizing the Aircraft Wing Weight Example	31
2.1. The AWWE Objective Function	31
2.2. Baseline Configuration	32
2.3. Optimization Setup	33
2.4. Method 1: SpotOptim (Surrogate Model Based Optimization)	33
2.5. Design Table	34
2.6. Run optimization	35
2.7. Result Table	36
2.8. Progress of the Optimization	37
2.9. Contour Plots of Most Important Hyperparameters	38
2.10. Method 2: Nelder-Mead Simplex	41
2.11. Method 3: BFGS (Quasi-Newton)	42
2.12. Comparison of Results	43
2.13. Visualization: Convergence Plots	44
2.13.1. SpotOptim Convergence	44
2.14. Optimal Parameter Values	45
2.15. Analysis of Optimal Solutions	47
2.16. Key Insights from Optimal Solutions	48
2.17. Method Efficiency Comparison	49
2.18. Visualization: 2D Slices of Optimal Solutions	50
2.19. Conclusion	52
2.20. Jupyter Notebook	53
3. Benchmarking SpotOptim with Sklearn Kriging (Matern Kernel) on 6D Rosenbrock and 10D Michalewicz Functions	55
3.1. SpotOptim with Sklearn Kriging in 6 Dimensions: Rosenbrock Function	55
3.1.1. Define the 6D Rosenbrock Function	55
3.1.2. Set up SpotOptim Parameters	56
3.1.3. Sklearn Gaussian Process Regressor as Surrogate	56
3.1.4. Visualize Optimization Progress	59
3.1.5. Evaluation of Multiple Repeats	60
3.2. SpotOptim with Sklearn Kriging in 10 Dimensions: Michalewicz Function	61
3.2.1. Define the 10D Michalewicz Function	61
3.2.2. Set up SpotOptim Parameters	62
3.2.3. Sklearn Gaussian Process Regressor as Surrogate	62
3.2.4. Visualize Optimization Progress	69
3.2.5. Evaluation of Multiple Repeats	70
3.3. Comparison: SpotOptim vs SpotPython	71
3.4. Summary	72

3.5. Jupyter Notebook	72
4. Success Rate Tracking in SpotOptim	73
4.1. What is Success Rate?	73
4.2. First Example	73
4.3. Second Example	74
4.4. Accessing Success Rate	75
4.5. Interpreting Success Rate	76
4.5.1. High Success Rate (> 0.5)	76
4.5.2. Medium Success Rate ($0.2 - 0.5$)	76
4.5.3. Low Success Rate (< 0.2)	77
4.6. TensorBoard Visualization	77
4.7. Example: Comparing Multiple Runs	77
4.8. Success Rate with Noisy Functions	79
4.9. Advanced: Custom Window Size	80
4.10. Best Practices	81
4.10.1. 1. Monitor During Long Runs	81
4.10.2. 2. Combine with TensorBoard	81
4.10.3. 3. Use as Stopping Criterion	81
4.10.4. 4. Compare Different Strategies	82
4.11. Technical Details	82
4.11.1. How Success is Counted	82
4.11.2. Rolling Window Calculation	82
4.11.3. Update Frequency	82
4.12. Summary	83
4.13. Jupyter Notebook	83
 II. Variables and Hyperparameters	 85
5. Variable Type (var_type) Implementation in SpotOptim	87
5.1. Overview	87
5.2. Supported Variable Types	87
5.2.1. 1. 'float'	87
5.2.2. 2. 'int'	87
5.2.3. 3. 'factor'	87
5.3. Implementation Details	88
5.3.1. Where var_type is Used	88
5.3.2. Core Method: _repair_non_numeric()	88
5.4. Example Usage	89
5.4.1. Example 1: All Float Variables (Default)	89
5.4.2. Example 2: Pure Integer Optimization	89
5.4.3. Example 3: Categorical (Factor) Variables	90
5.4.4. Example 4: Mixed Variable Types	91
5.5. Key Findings	92

Table of contents

5.6. Recommendations	92
5.7. Future Enhancements (Optional)	92
5.8. Jupyter Notebook	93
6. Factor Variables for Categorical Hyperparameters	95
6.1. Overview	95
6.2. Quick Start	95
6.2.1. Basic Factor Variable Usage	95
6.2.2. Neural Network Activation Function Optimization	97
6.3. Mixed Variable Types	99
6.3.1. Combining Factor, Integer, and Continuous Variables	99
6.4. Multiple Factor Variables	102
6.4.1. Optimizing Both Activation and Optimizer	102
6.5. Advanced Usage	104
6.5.1. Custom Categorical Choices	104
6.5.2. Viewing All Evaluated Configurations	105
6.6. How It Works	108
6.6.1. Internal Mechanism	108
6.6.2. Variable Type Auto-Detection	109
6.7. Complete Example: Full Workflow	110
6.8. Best Practices	118
6.8.1. Do's	118
6.8.2. Don'ts	119
6.9. Troubleshooting	120
6.9.1. Common Issues	120
6.10. Summary	121
6.11. Jupyter Notebook	121
7. Variable Transformations for Search Space Scaling	123
7.1. Overview	123
7.2. Why Use Transformations?	123
7.2.1. Problem: Poorly Scaled Search Spaces	123
7.2.2. Solution: Logarithmic and Other Transformations	124
7.3. Quick Start	124
7.3.1. Basic Log-Scale Transformation	124
7.4. Supported Transformations	125
7.4.1. Transformation Guidelines	126
7.5. Detailed Examples	127
7.5.1. Example 1: Neural Network Hyperparameter Tuning	127
7.5.2. Example 2: Physics-Informed Neural Networks (PINNs)	130
7.5.3. Example 3: Mixing Transformations	131
7.6. Viewing Transformations in Tables	132
7.6.1. Design Table (Before Optimization)	132
7.6.2. Results Table (After Optimization)	133

7.7. Internal Architecture	134
7.7.1. Flow Diagram	134
7.7.2. Key Components	134
7.8. Best Practices	135
7.8.1. 1. Choose Appropriate Transformations	135
7.8.2. 2. Match Transformation to Range	135
7.8.3. 3. Validate Transformation Choice	136
7.8.4. 4. Combine with Variable Types	136
7.9. Troubleshooting	136
7.9.1. Issue: Values Out of Bounds	136
7.9.2. Issue: Poor Optimization Performance	137
7.9.3. Issue: Transformation Not Applied	137
7.10. Comparison: Manual vs Automatic Transformations	138
7.10.1. Manual Approach (Old Way)	138
7.10.2. Automatic Approach (New Way)	138
7.11. Summary	139
7.12. Jupyter Notebook	140

III. Visualization 141

8. Surrogate Model Visualization 143

8.1. Overview	143
8.2. Features	143
8.3. Usage	143
8.3.1. Basic Usage	143
8.3.2. With Custom Parameters	144
8.3.3. Higher-Dimensional Problems	145
8.4. Plot Interpretation	147
8.4.1. Top Left: Prediction Surface	147
8.4.2. Top Right: Prediction Uncertainty Surface	147
8.4.3. Bottom Left: Prediction Contour with Points	148
8.4.4. Bottom Right: Uncertainty Contour with Points	148
8.5. Parameters	148
8.5.1. Dimension Selection	148
8.5.2. Appearance	148
8.5.3. Grid and Resolution	148
8.5.4. Color Scaling	148
8.5.5. Display	149
8.6. Examples	149
8.6.1. Example 1: 2D Rosenbrock Function	149
8.6.2. Example 2: Using Kriging Surrogate	150
8.6.3. Example 3: Comparing Different Dimension Pairs	151
8.7. Tips and Best Practices	154

Table of contents

8.8. Comparison with spotpython's plotkd()	155
8.8.1. Similarities	155
8.8.2. Differences	155
8.9. Error Handling	155
8.10. Jupyter Notebook	156
9. TensorBoard Logging in SpotOptim	157
9.1. Quick Start	157
9.1.1. Enable TensorBoard Logging	157
9.1.2. View Logs in TensorBoard	158
9.1.3. Cleaning Old Logs	158
9.1.4. Use Cases	159
9.1.5. Custom Log Directory	159
9.1.6. What Gets Logged	159
9.1.7. Examples	160
9.2. TensorBoard Features	162
9.2.1. SCALARS Tab	162
9.2.2. HPARAMS Tab	163
9.2.3. Text Tab	163
9.3. TensorBoard Log Cleaning Feature in SpotOptim	163
9.3.1. Basic Usage	164
9.3.2. Use Cases	165
9.3.3. Implementation Details	165
9.3.4. Execution Flow	165
9.4. Safety Features	166
9.5. Tensorboard Demo Scripts	166
9.6. Troubleshooting	167
9.7. Related Parameters	167
9.8. Performance Notes	167
9.9. Jupyter Notebook	168
IV. Saving and Loading Models	169
10. Save and Load in SpotOptim	171
10.1. Key Concepts	171
10.1.1. Experiments vs Results	171
10.1.2. What Gets Saved	171
10.2. Quick Start	172
10.2.1. Basic Save and Load	172
10.3. Distributed Workflow	173
10.3.1. Step 1: Define Experiment Locally	173
10.3.2. Step 2: Execute on Remote Machine	173
10.3.3. Step 3: Analyze Results Locally	175

10.4. Advanced Usage	176
10.4.1. Custom Filenames and Paths	176
10.4.2. Overwrite Protection	177
10.4.3. Loading and Continuing Optimization	178
10.5. Working with Noisy Functions	179
10.6. Working with Different Variable Types	180
10.7. Best Practices	181
10.7.1. 1. Always Re-attach the Objective Function	181
10.7.2. 2. Use Meaningful Prefixes	181
10.7.3. 3. Save Experiments Before Remote Execution	182
10.7.4. 4. Version Your Experiments	182
10.7.5. 5. Handle File Paths Robustly	182
10.8. Complete Example: Multi-Machine Workflow	183
10.8.1. Local Machine (Setup)	183
10.8.2. Remote Machine (Execution)	184
10.8.3. Local Machine (Analysis)	186
10.9. Technical Details	188
10.9.1. Serialization Method	188
10.9.2. Component Reinitialization	188
10.9.3. Excluded Components	188
10.9.4. File Format	189
10.10. Troubleshooting	189
10.10.1. Issue: “AttributeError: ‘SpotOptim’ object has no attribute ‘fun’”	189
10.10.2. Issue: “FileNotFoundError: Experiment file not found”	189
10.10.3. Issue: “FileExistsError: File already exists”	190
10.10.4. Issue: Results differ after loading	190
10.11. Jupyter Notebook	190
11. Reproducibility in SpotOptim	191
11.1. Introduction	191
11.2. Basic Usage	191
11.2.1. Making Optimization Reproducible	191
11.2.2. Running Independent Experiments	192
11.3. Practical Examples	193
11.3.1. Example 1: Comparing Different Configurations	193
11.3.2. Example 2: Reproducible Research Experiment	194
11.3.3. Example 3: Multiple Independent Runs	196
11.3.4. Example 4: Reproducible Initial Design	198
11.3.5. Example 5: Custom Initial Design with Seed	199
11.4. Advanced Topics	200
11.4.1. Seed and Noisy Functions	200
11.4.2. Different Seeds for Different Exploration	201
11.5. Best Practices	202
11.5.1. 1. Always Use Seeds for Production Code	202
11.5.2. 2. Document Your Seeds	202

Table of contents

11.5.3. 3. Use Different Seeds for Different Experiments	202
11.5.4. 4. Test Robustness Across Multiple Seeds	203
11.6. What the Seed Controls	203
11.7. Common Questions	203
11.8. Summary	204
11.9. Jupyter Notebook	204
V. Surrogate Handling	205
12. Surrogate Model Selection in SpotOptim	207
12.1. Introduction	207
12.2. Setup and Imports	208
12.3. The AWWE Objective Function	209
12.4. 1. Default Surrogate: Gaussian Process with Matern Kernel	210
12.4.1. Visualization: Default Surrogate	211
12.5. 2. Gaussian Process with RBF (Radial Basis Function) Kernel	215
12.5.1. Visualization: RBF Kernel	217
12.6. 3. Gaussian Process with Matern $\nu=1.5$ Kernel	220
12.6.1. Visualization: Matern $\nu=1.5$ Kernel	222
12.7. 4. Gaussian Process with Rational Quadratic Kernel	225
12.7.1. Visualization: Rational Quadratic Kernel	227
12.8. 5. SpotOptim Kriging Model	230
12.8.1. Visualization: Kriging Model	232
12.9. 6. Random Forest Regressor	235
12.9.1. Visualization: Random Forest	237
12.10. XGBoost Regressor	240
12.11. Support Vector Regression (SVR)	246
12.11.1. Visualization: SVR	248
12.12. Gradient Boosting Regressor	251
12.12.1. Visualization: Gradient Boosting	253
12.13. Comprehensive Comparison	256
12.13.1. Visualization: Performance Comparison	257
12.14. Convergence Comparison	259
12.15. Key Insights and Recommendations	260
12.16. Summary Statistics	262
12.17. Conclusion	264
12.18. Jupyter Notebook	264
13. Acquisition Failure Handling in SpotOptim	265
13.1. What is Acquisition Failure?	265
13.2. Fallback Strategies	265
13.2.1. 1. Random Space-Filling Design (Default)	265
13.2.2. 2. Morris-Mitchell Minimizing Point	267
13.3. How It Works	270

13.4. Comparison of Strategies	270
13.5. Complete Example: Comparing Strategies	270
13.6. Advanced Usage: Setting Tolerance	277
13.7. Best Practices	278
13.7.1. 1. Use Random for Most Problems	278
13.7.2. 2. Use Morris-Mitchell for Intensive Sampling	278
13.7.3. 3. Monitor Fallback Activations	279
13.7.4. 4. Adjust Tolerance Based on Problem Scale	279
13.8. Technical Details	280
13.8.1. Morris-Mitchell Implementation	280
13.8.2. Random Strategy Implementation	280
13.9. Summary	280
13.10.Jupyter Notebook	281
14. Point Selection Implementation in SpotOptim	283
14.1. Overview	283
14.2. Implementation Details	283
14.2.1. Parameters	283
14.2.2. Methods	283
14.3. Key Differences from spotpython	284
14.4. Example Usage	284
14.5. Benefits	286
14.6. Comparison with spotpython	286
14.7. Jupyter Notebook	286
VI. Kriging	287
15. Kriging Surrogate Integration	289
15.1. Overview	289
15.2. Module Structure	289
15.3. Kriging Class (src/spotoptim/surrogate/kriging.py)	289
15.4. Integration with SpotOptim	290
15.5. Documentation	290
15.6. Usage Examples	291
15.6.1. Basic Usage	291
15.6.2. Custom Parameters	291
15.6.3. Prediction with Uncertainty	292
15.7. Technical Details	293
15.7.1. Kriging vs GaussianProcessRegressor	293
15.7.2. Algorithm	293
15.7.3. Key Arguments Passed from SpotOptim	294
15.8. Benefits	294
15.9. Future Enhancements	294
15.10.Jupyter Notebook	295

16. Kriging Surrogate Models in SpotOptim	297
16.1. Introduction	297
16.1.1. What is Kriging?	297
16.1.2. Why Use Kriging in SpotOptim?	298
16.2. Basic Usage	298
16.2.1. Creating a Simple Kriging Model	298
16.2.2. Default vs Custom Surrogate	299
16.3. Understanding Kriging Parameters	300
16.3.1. Method: Interpolation vs Regression	300
16.3.2. Noise Parameter and Regularization	302
16.3.3. Correlation Length Parameters (Theta)	303
16.3.4. Isotropic vs Anisotropic Correlation	304
16.4. Advanced Features	305
16.4.1. Mixed Variable Types	305
16.4.2. Customizing the Distance Metric for Factors	308
16.4.3. Handling High-Dimensional Problems	309
16.4.4. Uncertainty Quantification	310
16.5. Practical Examples	312
16.5.1. Example 1: Optimizing the Rastrigin Function	312
16.5.2. Example 2: Robust Optimization with Noise	314
16.5.3. Example 3: Real-World Machine Learning Hyperparameter Tuning	316
16.5.4. Example 4: Comparing Kriging Methods	318
16.5.5. Example 5: Sensitivity to Theta Bounds	319
16.6. Comparing Surrogates	321
16.6.1. Kriging vs Gaussian Process vs Random Forest	321
16.7. Best Practices	322
16.7.1. 1. Choosing the Right Method	322
16.7.2. 2. Setting Model Complexity	323
16.7.3. 3. Handling Different Variable Types	323
16.7.4. 4. Reproducibility	324
16.7.5. 5. Monitoring Surrogate Quality	324
16.8. Common Issues and Solutions	325
16.8.1. Issue 1: Slow Fitting for Large Datasets	325
16.8.2. Issue 2: Poor Predictions for Categorical Variables	325
16.8.3. Issue 3: Numerical Instability	326
16.8.4. Issue 4: Overfitting to Noisy Data	326
16.9. Summary	326
16.9.1. Key Takeaways	326
16.9.2. Quick Reference	327
16.9.3. Parameter Cheat Sheet	327
16.10 Jupyter Notebook	328
17. Kriging in SpotOptim: The Forrester Implementation	329
18. Introduction	331

19.1. The Core Kriging Model (Section 2.4)	333
19.1. Correlation Function	333
19.2. Training: Concentrated Log-Likelihood	333
19.3. Prediction	334
20.2. Handling Noisy Data	335
20.1. A. Interpolation (method="interpolation")	335
20.2. B. Regression (method="regression")	335
20.3. C. Re-interpolation (method="reinterpolation")	336
21.3. Examples	337
21.1. Setup	337
21.2. Comparisons	337
21.3. Visualization	338
21.3.1. Interpretation	339
22.4. Infill Criteria	341
23. References	343
23.1. Jupyter Notebook	343
24. Nystroem Kernel Approximation	345
24.1. Introduction	345
24.2. The Nystroem Method	345
24.2.1. Moderate n , Large k Use Case	346
24.3. Implementation in SpotOptim	346
24.3.1. Components	346
24.3.2. Example Usage	346
24.4. Handling Mixed Variables	348
24.4.1. Using SpotOptimKernel	348
24.5. Jupyter Notebook	349
VII. Multiobjective Optimization	351
25. Multi-Objective Optimization Support in SpotOptim	353
25.1. Overview	354
25.2. What Was Implemented	354
25.2.1. 1. Core Functionality	354
25.3. Usage Examples	355
25.3.1. Example 1: Default Behavior (Use First Objective)	355
25.3.2. Example 2: Weighted Sum Scalarization	356
25.3.3. Example 3: Min-Max Scalarization	356
25.3.4. Example 4: Three or More Objectives	357
25.3.5. Example 5: With Noise Handling	357

Table of contents

25.4. Common Scalarization Strategies	358
25.4.1. 1. Weighted Sum	358
25.4.2. 2. Weighted Sum with Normalization	358
25.4.3. 3. Min-Max (Chebyshev)	358
25.4.4. 4. Target Achievement	358
25.4.5. 5. Product	359
25.5. Integration with Other Features	359
25.6. Implementation Details	359
25.6.1. Automatic Detection	359
25.6.2. Data Flow	360
25.6.3. Backward Compatibility	360
25.7. Limitations and Notes	360
25.7.1. What This Is	360
25.7.2. What This Is Not	360
25.7.3. For True Multi-Objective Optimization	361
25.8. Demo Script	361
25.9. Benchmark Multi-Objective Test Functions	361
25.9.1. Function Overview	361
25.9.2. Example 6: ZDT1 - Convex Pareto Front	362
25.9.3. Example 7: ZDT2 - Non-Convex Pareto Front	365
25.9.4. Example 8: ZDT3 - Disconnected Pareto Front	366
25.9.5. Example 9: ZDT4 - Multimodal with Many Local Fronts	368
25.9.6. Example 10: ZDT6 - Non-Uniform Density	369
25.9.7. Example 11: DTLZ2 - Scalable Spherical Pareto Front	371
25.9.8. Example 12: Schaffer N1 - Simple Bi-Objective	373
25.9.9. Example 13: Fonseca-Fleming - Symmetric Concave Front	375
25.10. Example 14: Kursawe - Non-Convex Disconnected Front	377
25.10.1. Comparison: Different Scalarization Strategies on ZDT1	377
25.11. A Simple Visualization Example	379
25.12. Jupyter Notebook	386
VIII Hyperparameter Tuning	387
26. Physics-Informed Neural Networks (PINNs) Demo 1	389
27. Overview	391
27.1. The Differential Equation	391
28. Setup	393
29. The Neural Network	395
30. Generate Training Data	397

31. Collocation Points	401
32. PINN Training	403
32.1. Loss Components Explained	403
32.1.1. Neural Network Prediction on Data Points	403
32.1.2. Data Loss (Loss1)	403
32.1.3. Neural Network Prediction on Collocation Points	403
32.1.4. Computing Derivatives for the ODE	403
32.1.5. Physics Loss (Loss2)	404
32.1.6. Total Loss	404
32.2. Training Loop	404
33. Results Visualization	407
33.1. Training Progress	407
33.2. Solution Evolution During Training	408
33.3. Final Solution Comparison	410
33.4. Error Analysis	412
34. Summary	415
34.1. Key Advantages of PINNs	415
34.2. Using SpotOptim for Hyperparameter Optimization	415
34.3. Jupyter Notebook	416
35. Hyperparameter Tuning for Physics-Informed Neural Networks	417
36. Overview	419
36.1. Key Features	419
36.1.1. 1. PyTorch Dataset and DataLoader	419
36.1.2. 2. Automatic Transformation Handling	420
37. The Problem	421
38. Setup	423
39. Data Generation	425
39.1. Custom Dataset Classes	425
40. Define the PINN Training Function	429
41. Hyperparameter Optimization with SpotOptim	433
41.1. Define the Objective Function	433
41.2. Run the Optimization	435
42. Results Analysis	439
42.1. Best Configuration	439
42.1.1. Results Table with Importance Scores	440

Table of contents

42.2. Optimization History	440
42.3. Surrogate Visualization	441
42.4. Parameter Distribution Analysis	444
43. Train Final Model with Best Hyperparameters	447
44. Evaluate Final Model	451
45. Visualize Final Solution	453
46. Comparison with Baseline	455
47. Hyperparameter Sensitivity Analysis	459
47.1. Sensitivity Analysis (Spearman Correlation)	460
48. Summary	461
48.1. Key Findings	461
48.2. Recommendations	462
48.3. Using These Results	463
48.4. Using var_trans for Your Hyperparameter Optimization	464
48.5. Future Directions	465
48.6. Jupyter Notebook	465
49. Learning Rate Mapping for Unified Optimizer Interface	467
49.1. Overview	467
49.2. Quick Start	467
49.2.1. Basic Usage	467
49.2.2. Scaling Learning Rates	468
49.2.3. Integration with LinearRegressor	469
49.3. Function Reference	469
49.3.1. map_lr(lr_unified, optimizer_name, use_default_scale=True)	469
49.4. Supported Optimizers	470
49.5. Use Cases	471
49.5.1. Comparing Different Optimizers	471
49.5.2. Hyperparameter Optimization	472
49.5.3. Hyperparameter Optimization with SpotOptim	474
49.5.4. Log-Scale Hyperparameter Search	476
49.5.5. Custom Learning Rate Schedules	477
49.5.6. Direct Usage Without LinearRegressor	478
49.6. Best Practices	479
49.6.1. Choosing Unified Learning Rate	479
49.6.2. Optimizer Selection Guidelines	479
49.6.3. Common Patterns	480
49.7. Troubleshooting	480
49.7.1. Issue: Training is unstable (loss explodes)	480
49.7.2. Issue: Training is too slow (loss decreases very slowly)	481

49.7.3. Issue: Different results across optimizer runs	481
49.7.4. Issue: Want to use raw learning rate without mapping	481
49.7.5. Issue: Optimizer not supported	481
49.8. Technical Details	481
49.8.1. How It Works	481
49.8.2. Design Rationale	482
49.8.3. Default Learning Rates	482
49.9. Examples	483
49.9.1. Complete Example: Optimizer Comparison Study	483
49.10 Jupyter Notebook	486

IX. Data Sets 487

50. Diabetes Dataset Utilities 489

50.1. Overview	489
50.2. Quick Start	489
50.2.1. Basic Usage	489
50.2.2. Training a Model	490
50.3. Function Reference	492
50.3.1. get_diabetes_dataloaders()	492
50.4. DiabetesDataset Class	493
50.4.1. Manual Dataset Creation	494
50.5. Advanced Usage	495
50.5.1. Custom Transforms	495
50.5.2. Different Train/Test Splits	496
50.5.3. Without Feature Scaling	496
50.5.4. Larger Batch Sizes	497
50.5.5. GPU Training with Pin Memory	497
50.6. Complete Training Example	498
50.7. Integration with SpotOptim	501
50.8. Best Practices	504
50.8.1. 1. Always Use Feature Scaling	504
50.8.2. 2. Set Random Seeds for Reproducibility	504
50.8.3. 3. Don't Shuffle Test Data	505
50.8.4. 4. Choose Appropriate Batch Size	505
50.8.5. 5. Save the Scaler for Production	505
50.9. Troubleshooting	506
50.9.1. Issue: Out of Memory	506
50.9.2. Issue: Different Data Ranges	506
50.9.3. Issue: Non-Reproducible Results	507
50.9.4. Issue: Slow Data Loading	507
50.10 Summary	507
50.11 Jupyter Notebook	508

X. Details About SPOT	509
51. SpotOptim Step-by-Step Optimization Process	511
52. Introduction	513
53. Setup and Test Functions	515
54. Standard Optimization Workflow	517
54.1. Phase: Initialization and Setup	517
54.2. Phase: Initial Design Generation	518
54.2.1. Method: <code>get_initial_design()</code>	518
54.3. Phase 3: Initial Design Curation	521
54.3.1. Method: <code>_curate_initial_design()</code>	521
54.4. Phase 4: Initial Design Evaluation	521
54.4.1. Method: <code>_evaluate_function()</code>	521
54.5. Phase 5: Handling Failed Evaluations	522
54.5.1. Method: <code>_handle_NA_initial_design(X0,y0)</code>	522
54.6. Phase 6: Validation Check	523
54.6.1. Method: <code>_check_size_initial_design(y0, n_evaluated)</code>	523
54.7. Phase 7: Storage Initialization	523
54.7.1. Method: <code>_init_storage()</code>	523
54.8. Phase: Statistics Update	524
54.8.1. Method: <code>update_stats()</code>	524
54.9. Phase: Log initial Design to TensorBoard	524
54.9.1. Method: <code>_log_initial_design_tensorboard()</code>	524
54.10. Phase: Initial Best Point	525
54.10.1. Method: <code>_get_best_xy_initial_design()</code>	525
54.11. Illustration of the Initial Design Phase Results	525
55. Sequential Optimization Loop	527
55.1. Step: Surrogate Model Fitting	527
55.1.1. Method: <code>_fit_scheduler()</code>	527
55.1.2. Method: <code>_fit_surrogate()</code>	527
55.1.3. Method: <code>_selection_dispatcher()</code>	528
55.1.4. Step: Apply OCBA	529
55.1.5. Method: <code>_apply_ocba()</code>	529
55.2. Step: Predict with Uncertainty	530
55.2.1. Method: <code>_predict_with_uncertainty()</code>	530
55.3. Step: Next Point Suggestion	530
55.3.1. Method: <code>_suggest_next_point()</code>	530
55.4. Step: Acquisition Function Evaluation	531
55.4.1. Method: <code>_acquisition_function()</code>	531
55.5. Step: Update Repeats for Infill Points	532
55.5.1. Method: <code>_update_repeats_infill_points()</code>	532

55.6. Append OCBA Points to Infill Points	533
55.7. Step: Evaluation of New Points	533
55.7.1. Method: <code>_evaluate_function()</code> (again)	533
55.8. Step: Handle Failed Evaluations (Sequential)	533
55.8.1. Method: <code>_handle_NA_new_points()</code>	533
55.9. Step: Update Success Rate	534
55.9.1. Method: <code>_update_success_rate(y0)</code>	534
55.10 Step: Update Storage	535
55.10.1 Internal updates	535
55.11 Update Statistics	535
55.11.1. What happens in <code>update_stats()</code> :	535
55.12 Step: Update Best Solution	536
55.12.1 Method: <code>_update_best_main_loop()</code>	536
56. Complete Optimization Example	537
57. Noisy Functions with Repeats	539
57.1. Configuration for Noisy Functions	539
57.2. Noisy Optimization Workflow Differences	540
57.2.1. Modified Initial Design	540
57.2.2. Update Statistics Method	541
58. Optimal Computing Budget Allocation (OCBA)	543
58.1. OCBA Configuration	543
58.2. OCBA Method	544
58.2.1. Method: <code>_apply_ocba()</code>	544
59. Handling Function Evaluation Failures	547
59.1. Example with Failures	547
59.2. Failure Handling in Initial Design	548
59.2.1. Method: <code>_handle_NA_initial_design()</code>	548
59.3. Failure Handling in Sequential Phase	549
59.3.1. Method: <code>_handle_NA_new_points()</code>	549
59.4. Penalty Application	549
59.4.1. Method: <code>_apply_penalty_NA()</code>	549
60. Complete Method Summary	551
60.1. Methods Called During <code>optimize()</code>	551
60.1.1. Preparation Phase	551
60.1.2. Sequential Optimization Loop (each iteration)	551
60.1.3. Finalization	552
60.2. Helper Methods Used	552
61. Termination Conditions	553
61.1. Method: <code>_determine_termination()</code>	553

62. Performance Comparison on Noisy Functions	555
63. Summary	559
63.1. Complete Workflow Diagram	559
63.2. Key Concepts	561
63.2.1. 1. Initial Design	561
63.2.2. 2. Surrogate Model	561
63.2.3. 3. Acquisition Function	561
63.2.4. 4. Noise Handling	561
63.2.5. 5. Failure Handling	562
63.3. Best Practices	562
63.4. Conclusion	562
63.5. Jupyter Notebook	563
XI. Optimization	565
64. Aircraft Wing Weight Example	567
64.1. AWWE Equation	567
64.2. AWWE Parameters and Equations (Part 1)	567
64.3. Goals: Understanding and Optimization	568
64.4. Properties of the Python “Solver”	570
64.5. Plot 1: Load Factor (N_z) and Aspect Ratio (A)	571
64.6. Plot 2: Taper Ratio and Fuel Weight	573
64.7. The Big Picture: Combining all Variables	574
64.8. AWWE Landscape	577
64.9. Summary of the First Experiments	578
64.10 Exercise	578
64.10.1 Adding Paint Weight	578
64.11 Jupyter Notebook	579
XII. Numerical Methods	581
65. Simulation and Surrogate Modeling	583
65.1. Surrogates	583
65.1.1. Costs of Simulation	584
65.1.2. Mathematical Models and Meta-Models	584
65.1.3. Surrogates = Trained Meta-models	584
65.1.4. Computer Experiments	584
65.1.5. Limits of Mathematical Modeling	585
65.1.6. Why Computer Simulations are Necessary	585
65.1.7. Simulation Requirements	585
65.2. Applications of Surrogate Models	585

65.3. DACE and RSM	586
65.3.1. Noise Handling in RSM and DACE	587
65.4. Updating a Surrogate Model	587
66. Sampling Plans	589
66.1. Ideas and Concepts	589
66.1.1. The ‘Curse of Dimensionality’ and How to Avoid It	591
66.1.2. Physical versus Computational Experiments	591
66.1.3. Designing Preliminary Experiments (Screening)	592
66.1.4. Special Considerations When Deploying Screening Algorithms	601
66.2. Analyzing Variable Importance in Aircraft Wing Weight	601
66.3. Designing a Sampling Plan	604
66.3.1. Stratification	604
66.3.2. Latin Squares and Random Latin Hypercubes	609
66.3.3. Space-filling Designs: Maximin Plans	612
66.3.4. Memory Management	615
66.3.5. Optimizing the Morris-Mitchell Criterion Φ_q	625
66.3.6. Evolutionary Operation	627
66.3.7. Putting it all Together	628
66.4. Experimental Analysis of the Morris-Mitchell Criterion	635
66.4.1. Evaluation of Sampling Designs	635
66.4.2. Demonstrate the Impact of <code>mmphi</code> Parameters	639
66.4.3. Morris-Mitchell Criterion: Impact of Adding Points	639
66.5. A Sample-Size Invariant Version of the Morris-Mitchell Criterion	644
66.5.1. Comparison of <code>mmphi()</code> and <code>mmphi_intensive()</code>	644
66.5.2. Plotting the Two Morris-Mitchell Criteria for Different Sample Sizes	645
66.6. Jupyter Notebook	648
67. Constructing a Surrogate	649
67.1. Stage One: Preparing the Data and Choosing a Modelling Approach	649
67.2. Stage Two: Parameter Estimation and Training	651
67.3. Stage Three: Model Testing	653
67.4. Jupyter Notebook	654
68. Response Surface Methods	655
68.1. What is RSM?	655
68.1.1. Visualization: Problems in Practice	658
68.1.2. RSM: Strategies	658
68.1.3. RSM: Noise in the Empirical Model	659
68.1.4. RSM: Natural and Coded Variables	659
68.1.5. RSM Low-order Polynomials	660
68.2. First-Order Models (Main Effects Model)	660
68.2.1. First-Order Model Properties	661
68.2.2. First-order Model with Interactions in python	661

Table of contents

68.2.3. Observations: First-Order Model with Interactions	663
68.3. Second-Order Models	663
68.3.1. Second-Order Models: Properties	664
68.3.2. Example: Stationary Ridge	664
68.3.3. Observations: Second-Order Model (Ridge)	665
68.3.4. Example: Rising Ridge	666
68.3.5. Summary: Rising Ridge	667
68.3.6. Falling Ridge	667
68.3.7. Saddle Point	667
68.3.8. Interpretation: Saddle Points	669
68.3.9. Summary: Ridge Analysis	669
68.4. General RSM Models	669
68.4.1. Ordinary Least Squares	669
68.5. General Linear Regression	670
68.6. Designs	671
68.6.1. Different Designs	672
68.7. RSM Experimentation	672
68.7.1. First Step	672
68.7.2. Second Step	672
68.7.3. Third Step	672
68.8. RSM: Review and General Considerations	673
68.8.1. Historical Considerations about RSM	674
68.8.2. Status Quo	674
68.8.3. The Role of Statistics	674
68.8.4. New RSM is needed: DACE	674
68.9. Exercises	675
68.10 Jupyter Notebook	675
69. Polynomial Models	677
69.1. Fitting a Polynomial	677
69.2. Polynomial Fitting in Python	678
69.2.1. Fitting the Polynomial	678
69.2.2. Explaining the k -fold Cross-Validation	679
69.2.3. Making Predictions	680
69.2.4. Plotting the Results	681
69.3. Example One: Aerofoil Drag	682
69.4. Example Two: A Multimodal Test Case	684
69.5. Extending to Multivariable Polynomial Models	685
69.6. Jupyter Notebook	687
70. Radial Basis Function Models	689
70.1. Radial Basis Function Models	689
70.1.1. Fitting Noise-Free Data	690
70.1.2. Numerical Stability Through Positive Definite Matrices	694
70.1.3. Ill-Conditioning	695

70.1.4. Parameter Optimization: A Two-Level Approach	696
70.2. Python Implementation of the RBF Model	699
70.2.1. The Rbf Class	700
70.3. RBF Example: The One-Dimensional <code>sin</code> Function	706
70.4. RBF Example: The Two-Dimensional <code>dome</code> Function	708
70.4.1. The Connection Between RBF Models and Neural Networks	711
70.5. Radial Basis Function Models for Noisy Data	712
70.5.1. Ridge Regularization Approach	712
70.5.2. Reduced Basis Approach	712
70.6. Jupyter Notebook	713
71. Kriging (Gaussian Process Regression)	715
71.1. From Gaussian RBF to Kriging Basis Functions	715
71.2. Building the Kriging Model	716
71.3. MLE to estimate θ and p	723
71.3.1. The Log-Likelihood	723
71.3.2. Differentiation with Respect to μ	725
71.3.3. Differentiation with Respect to σ	726
71.3.4. Results of the Optimizations	727
71.3.5. The Concentrated Log-Likelihood Function	727
71.3.6. Optimizing the Parameters $\vec{\theta}$ and $\vec{p}$	728
71.3.7. Correlation and Covariance Matrices Revisited	728
71.4. Implementing an MLE of the Model Parameters	729
71.5. Kriging Prediction	729
71.6. Kriging Example: Sinusoid Function	731
71.6.1. Calculating the Correlation Matrix Ψ	731
71.6.2. Computing the Ψ Matrix	733
71.6.3. Selecting the New Locations	734
71.6.4. Computing the ψ Vector	734
71.6.5. Predicting at New Locations	735
71.6.6. Visualization	735
71.6.7. The Complete Python Code for the Example	736
71.7. Jupyter Notebook	742
72. Matrices	743
72.1. Derivatives of Quadratic Forms	743
72.2. The Condition Number	744
72.3. The Moore-Penrose Pseudoinverse	745
72.3.1. Definitions	745
72.3.2. Implementation in Python	745
72.4. Strictly Positive Definite Kernels	746
72.4.1. Definition	746
72.4.2. Connection to Positive Definite Matrices	746
72.4.3. Connection to RBF Models	747

Table of contents

72.5. Cholesky Decomposition and Positive Definite Matrices	747
72.5.1. Example of Cholesky Decomposition	750
72.5.2. Inverse Matrix Using Cholesky Decomposition	751
72.6. Nyström Approximation	752
72.6.1. What's the Big Idea?	752
72.6.2. How Does It Work?	752
72.6.3. Example	753
72.6.4. Why Is This Useful?	753
72.6.5. Applying the Nyström Approximation: How Nyström Approximation Helps Kriging	754
72.6.6. Example: Predicting Temperature with Nyström-Kriging	754
72.6.7. Details: Woodbury Matrix Identity for Avoiding the Big Inversion	755
72.6.8. The Example: Step-by-Step	756
72.7. Extending spotpython's Kriging Surrogate with Nyström Approximation for Enhanced Scalability	758
72.7.1. Introduction: Overcoming the Scalability Challenge in Kriging for Sequential Optimization	758
72.7.2. Report Objectives and Structure	759
72.8. Theoretical Foundations: The Nyström-Kriging Framework	759
72.8.1. A Primer on Kriging (Gaussian Process Regression)	759
72.8.2. The Nyström Method for Low-Rank Kernel Approximation	760
72.8.3. Justification for Landmark Selection	760
72.9. Implementation: A Scalable Kriging Class for spotpython	760
72.9.1. Updated kriging.py with Nyström Approximation (excerpt)	760
72.10. Implementation Details	763
72.10.1. Architectural Enhancements to <code>init</code>	763
72.10.2. The <code>fit()</code> Method: A Dual-Pathway Approach	763
72.10.3. The <code>predict()</code> Method: Conditional Prediction Logic	763
72.10.4. Critical Detail: Preserving Mixed Variable Type Functionality	764
72.10.5. Seamless Integration into the Nyström Workflow	764
72.11. Jupyter Notebook	764
73. Infill Criteria	765
73.1. Balancing Exploitation and Exploration	766
73.2. Expected Improvement (EI)	766
73.3. Expected Improvement in the Hyperparameter Tuning Cookbook (Python Implementation)	767
73.4. Jupyter Notebook	768
74. Remote Optimization Challenge: A Benchmarking Study	769
74.1. Benchmarking Best Practices	769
74.2. Experimental Setup	769
74.2.1. The Remote Objective Function	769
74.2.2. Algorithms	770
74.2.3. Budget and Settings	770

Table of contents

74.3. Comparison of the Difficulty	770
74.4. Statistical Power Analysis	773
74.4.1. 1. Estimating Noise (σ)	773
74.4.2. 2. Determining Sample Size (n)	773
74.5. Experimental Comparison	774
74.6. Convergence Analysis (Noisy)	778
74.7. Comparative Analysis	780
74.8. Conclusion	782
74.9. References	782
74.10.Jupyter Notebook	782
References	783

Preface

This document provides a comprehensive guide to optimization and hyperparameter tuning using the Sequential Parameter Optimization Toolbox (SPOT) for Python.

Part I.

**Sequential Parameter
Optimization Toolbox (SPOT)**

1. SpotOptim Internal Methods Examples

1.1. Setup: Import Required Packages

First, we import all necessary packages for the examples.

```
# Core imports
import numpy as np
import time
from scipy.optimize import OptimizeResult

# SpotOptim
from spotoptim import SpotOptim

# Surrogate models
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, ConstantKernel

# Set random seed for reproducibility
np.random.seed(42)

print("All packages imported successfully!")
```

All packages imported successfully!

1.2. 1. Main Optimization Method: `optimize()`

The `optimize()` method is the main entry point for running the optimization process. It coordinates all other methods in the optimization workflow:

1. **Initial Design Phase:** `get_initial_design()`, `_curate_initial_design()`, `_handle_NA_initial_design()`, `_check_size_initial_design()`, `_get_best_xy_initial_design()`
2. **Main Loop:** Surrogate fitting, OCBA application, point suggestion, evaluation
3. **Termination:** `_determine_termination()`

1. SpotOptim Internal Methods Examples

Let's see a complete optimization example:

```
# Define a simple quadratic function
def sphere(X):
    """Sphere function:  $f(x) = \sum(x^2)$ """
    X = np.atleast_2d(X)
    return np.sum(X**2, axis=1)

# Create optimizer
opt = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    n_initial=5,
    max_iter=20,
    verbose=True
)

# Run optimization
result = opt.optimize()

print(f"\nBest point found: {result.x}")
print(f"Best value: {result.fun:.6f}")
print(f"Total evaluations: {result.nfev}")
print(f"Sequential iterations: {result.nit}")
print(f"Success: {result.success}")
print(f"Message: {result.message}")
```

```
TensorBoard logging disabled
Initial best:  $f(x) = 3.258069$ 
Iteration 1: New best  $f(x) = 1.641363$ 
Iteration 2: New best  $f(x) = 1.445519$ 
Iteration 3: New best  $f(x) = 0.233819$ 
Iteration 4: New best  $f(x) = 0.152919$ 
Iteration 5: New best  $f(x) = 0.023471$ 
Iteration 6: New best  $f(x) = 0.000173$ 
Iteration 7: New best  $f(x) = 0.000042$ 
Iteration 8: New best  $f(x) = 0.000002$ 
Iteration 9: New best  $f(x) = 0.000001$ 
Iteration 10: New best  $f(x) = 0.000001$ 
Iteration 11:  $f(x) = 0.000001$ 
Iteration 12: New best  $f(x) = 0.000001$ 
Iteration 13:  $f(x) = 0.000001$ 
Iteration 14:  $f(x) = 0.000001$ 
Iteration 15:  $f(x) = 0.000001$ 
```

```
Best point found: [ 0.00060119 -0.00083804]
Best value: 0.000001
Total evaluations: 20
Sequential iterations: 15
Success: True
Message: Optimization terminated: maximum evaluations (20) reached
```

1.3. 2. Initial Design Methods

These methods handle the creation and validation of the initial design of experiments.

1.3.1. 2.1 `get_initial_design()`

Purpose: Generate or process initial design points

Used by: `optimize()` method at the start

Calls: `_generate_initial_design()` if `X0` is None

This method handles three scenarios: 1. Generate LHS design when `X0=None` 2. Include starting point `x0` if provided 3. Transform user-provided `X0`

```
# Example 1: Generate default LHS design
opt = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    n_initial=10,
    seed=42
)
X0 = opt.get_initial_design()
print(f"Generated LHS design shape: {X0.shape}")
print(f"First 3 points:\n{X0[:3]}")

# Example 2: With starting point x0
opt_x0 = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    n_initial=10,
    x0=[0.0, 0.0],
    seed=42
)
X0_with_x0 = opt_x0.get_initial_design()
print(f"\nDesign with x0, first point: {X0_with_x0[0]}")
```

1. SpotOptim Internal Methods Examples

```
# Example 3: Provide custom initial design
X0_custom = np.array([[0, 0], [1, 1], [2, 2]])
X0_processed = opt.get_initial_design(X0_custom)
print(f"\nCustom design shape: {X0_processed.shape}")
```

Generated LHS design shape: (10, 2)

First 3 points:

```
[[-2.77395605  1.56112156]
 [ 4.14140208 -1.69736803]
 [-4.09417735  3.02437765]]
```

Design with x0, first point: [0. 0.]

Custom design shape: (3, 2)

1.3.2. 2.2 _curate_initial_design()

Purpose: Remove duplicates and ensure sufficient unique points

Used by: optimize() after get_initial_design()

Handles: Duplicate removal, point generation, repetition for noisy functions

```
# Example 1: Remove duplicates
opt = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    n_initial=10,
    seed=42
)
X0_with_dups = np.array([[1, 2], [1, 2], [3, 4], [3, 4], [5, 6]])
X0_curated = opt._curate_initial_design(X0_with_dups)
print(f"Original points: {len(X0_with_dups)}")
print(f"After removing duplicates: {len(X0_curated)}")

# Example 2: With repeats for noisy functions
opt_repeat = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    n_initial=5,
    repeats_initial=3, # Repeat each point 3 times
    seed=42
)
X0 = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]])
```

```
X0_repeated = opt_repeat._curate_initial_design(X0)
print(f"\nOriginal points: {len(X0)}")
print(f"After repeating (3x): {len(X0_repeated)}")
```

```
Original points: 5
After removing duplicates: 10
```

```
Original points: 5
After repeating (3x): 15
```

1.3.3. 2.3 _handle_NA_initial_design()

Purpose: Remove NaN/inf values from initial design evaluations

Used by: `optimize()` after evaluating initial design

Returns: Cleaned arrays and original count

```
opt = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    n_initial=10,
    seed=42
)

# Simulate initial design with some NaN values
X0 = np.array([[1, 2], [3, 4], [5, 6]])
y0 = np.array([5.0, np.nan, np.inf])

X0_clean, y0_clean, n_eval = opt._handle_NA_initial_design(X0, y0)

print(f"Original evaluations: {n_eval}")
print(f"Valid points remaining: {X0_clean.shape[0]}")
print(f"Clean X0:\n{X0_clean}")
print(f"Clean y0: {y0_clean}")
```

```
Original evaluations: 3
Valid points remaining: 1
Clean X0:
[[1 2]]
Clean y0: [5.]
```

1. SpotOptim Internal Methods Examples

1.3.4. 2.4 _check_size_initial_design()

Purpose: Validate sufficient points for surrogate fitting

Used by: optimize() after handling NaN values

Raises: ValueError if insufficient points

```
opt = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    n_initial=10,
    seed=42
)

# Example 1: Sufficient points - no error
y0_sufficient = np.array([1.0, 2.0, 3.0, 4.0, 5.0])
try:
    opt._check_size_initial_design(y0_sufficient, n_evaluated=10)
    print(" Sufficient points - validation passed")
except ValueError as e:
    print(f" Error: {e}")

# Example 2: Insufficient points - raises error
y0_insufficient = np.array([1.0]) # Only 1 point, need at least 3 for 2D
try:
    opt._check_size_initial_design(y0_insufficient, n_evaluated=10)
    print(" Validation passed")
except ValueError as e:
    print(f" Expected error: {e}")
```

Sufficient points - validation passed

Expected error: Insufficient valid initial design points: only 1 finite value(s) out

1.3.5. 2.5 _get_best_xy_initial_design()

Purpose: Determine and store the best point from initial design

Used by: optimize() after initial design evaluation

Updates: self.best_x_ and self.best_y_ attributes

```
opt = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    n_initial=5,
    verbose=True,
```



```

    seed=42
)

# Simulate initial design (normally done in optimize())
opt.X_ = np.array([[1, 2], [0, 0], [2, 1]])
opt.y_ = np.array([5.0, 0.0, 5.0])

opt._get_best_xy_initial_design()

print(f"\nBest x from initial design: {opt.best_x_}")
print(f"Best y from initial design: {opt.best_y_}")

```

TensorBoard logging disabled
Initial best: f(x) = 0.000000

Best x from initial design: [0 0]
Best y from initial design: 0.0

1.4. 3. Surrogate Model Methods

These methods handle surrogate model operations during the optimization loop.

1.4.1. 3.1 _fit_surrogate()

Purpose: Fit surrogate model to data

Used by: optimize() in main loop

Calls: _selection_dispatcher() if max_surrogate_points exceeded

```

opt = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    max_surrogate_points=10,
    seed=42
)

# Generate some training data
X = np.random.rand(50, 2) * 10 - 5 # 50 points in [-5, 5]
y = np.sum(X**2, axis=1)

# Fit surrogate (will select 10 best points)
opt._fit_surrogate(X, y)

```

1. SpotOptim Internal Methods Examples

```
print(f"Surrogate fitted successfully!")
print(f"Surrogate model: {type(opt.surrogate).__name__}")
```

```
Surrogate fitted successfully!
Surrogate model: GaussianProcessRegressor
```

1.4.2. 3.2 _predict_with_uncertainty()

Purpose: Predict with uncertainty estimates, handling surrogates without return_std

Used by: _acquisition_function() and plot_surrogate()

Returns: Predictions and standard deviations

```
opt = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    surrogate=GaussianProcessRegressor(),
    seed=42
)

# Train surrogate
X_train = np.array([[0, 0], [1, 1], [2, 2]])
y_train = np.array([0, 2, 8])
opt._fit_surrogate(X_train, y_train)

# Predict on new points
X_test = np.array([[1.5, 1.5], [3.0, 3.0]])
preds, stds = opt._predict_with_uncertainty(X_test)

print(f"Test points:\n{X_test}")
print(f"Predictions: {preds}")
print(f"Uncertainties: {stds}")
```

```
Test points:
[[1.5 1.5]
 [3.  3. ]]
Predictions: [5.66013019 3.0829763 ]
Uncertainties: [0.31263595 0.92017596]
```

1.4.3. 3.3 _acquisition_function()

Purpose: Compute acquisition function value

Used by: _suggest_next_point() for optimization

Calls: `_predict_with_uncertainty()`

Supports: Expected Improvement (EI), Probability of Improvement (PI), Mean prediction

```
opt = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    surrogate=GaussianProcessRegressor(),
    acquisition='ei',
    seed=42
)

# Setup
X_train = np.array([[0, 0], [1, 1], [2, 2]])
y_train = np.array([0, 2, 8])
opt._fit_surrogate(X_train, y_train)
opt.y_ = y_train # Needed for acquisition function

# Evaluate acquisition function
x_eval = np.array([1.5, 1.5])
acq_value = opt._acquisition_function(x_eval)

print(f"Point to evaluate: {x_eval}")
print(f"Acquisition function value (EI): {acq_value:.6f}")
print(f"(Lower is better for minimization)")
```

```
Point to evaluate: [1.5 1.5]
Acquisition function value (EI): -0.000000
(Lower is better for minimization)
```

1.4.4. 3.4 `_suggest_next_point()`

Purpose: Suggest next point to evaluate using acquisition function optimization

Used by: `optimize()` in main loop

Calls: `_acquisition_function()`, `_handle_acquisition_failure()` if needed

Handles: Integer/factor rounding, duplicate avoidance

```
opt = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    surrogate=GaussianProcessRegressor(),
    acquisition='ei',
    seed=42
```

1. SpotOptim Internal Methods Examples

```
)

# Setup
X_train = np.array([[0, 0], [1, 1], [2, 2]])
y_train = np.array([0, 2, 8])
opt._fit_surrogate(X_train, y_train)
opt.X_ = X_train
opt.y_ = y_train

# Suggest next point
x_next = opt._suggest_next_point()

print(f"Next point to evaluate: {x_next}")
print(f"Expected to be between known points or in unexplored regions")
```

Next point to evaluate: [-1.6288683 1.99062025]
Expected to be between known points or in unexplored regions

1.5. 4. OCBA Method (Optimal Computing Budget Allocation)

1.5.1. 4.1 _apply_ocba()

Purpose: Apply OCBA for noisy functions to determine which points to re-evaluate

Used by: optimize() in main loop when noise=True and ocba_delta > 0

Returns: Points to re-evaluate or None

```
# Example with OCBA enabled
opt = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    repeats_initial=2, # Enable noise handling
    ocba_delta=5,
    verbose=True,
    seed=42
)

# Simulate optimization state
opt.mean_X = np.array([[1, 2], [0, 0], [2, 1], [1, 1]])
opt.mean_y = np.array([5.0, 0.1, 5.0, 2.0])
opt.var_y = np.array([0.1, 0.05, 0.15, 0.08])
```

```

X_ocba = opt._apply_ocba()

if X_ocba is not None:
    print(f"\nOCBA returned {X_ocba.shape[0]} points for re-evaluation")
else:
    print("\nOCBA: No re-evaluation needed")

```

TensorBoard logging disabled

```

In _get_ocba():
means: [5.  0.1 5.  2. ]
vars: [0.1  0.05 0.15 0.08]
delta: 5
n_designs: 4
Ratios: [0.18794252 0.81801772 0.28191379 1.          ]
Best: 1, Second best: 3
      OCBA: Adding 5 re-evaluation(s)

```

OCBA returned 5 points for re-evaluation

1.6. 5. NaN/Inf Handling Methods

1.6.1. 5.1 _apply_penalty_NA()

Purpose: Replace NaN and infinite values with penalty plus random noise

Used by: `_handle_NA_new_points()` and indirectly by `optimize()`

Algorithm: $\text{penalty} = \max(\text{finite\_y}) + 3 * \text{std}(\text{finite\_y}) + \text{noise}$

```

opt = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5)],
    seed=42
)

# Historical values (used for penalty computation)
y_hist = np.array([1.0, 2.0, 3.0, 5.0])

# New evaluations with NaN/inf
y_new = np.array([4.0, np.nan, np.inf])

y_clean = opt._apply_penalty_NA(y_new, y_history=y_hist)

```

1. SpotOptim Internal Methods Examples

```
print(f"Original: {y_new}")
print(f"After penalty: {y_clean}")
print(f"All finite: {np.all(np.isfinite(y_clean))}")
print(f"Penalty values > max(hist): {y_clean[1] > np.max(y_hist)}")
```

```
Original: [ 4. nan inf]
After penalty: [ 4.          10.22100895 10.1332034 ]
All finite: True
Penalty values > max(hist): True
```

1.6.2. 5.2 _remove_nan()

Purpose: Remove rows where y contains NaN or inf values

Used by: optimize() after function evaluations

Returns: Cleaned arrays

```
opt = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5)],
    seed=42
)

X = np.array([[1, 2], [3, 4], [5, 6]])
y = np.array([1.0, np.nan, np.inf])

X_clean, y_clean = opt._remove_nan(X, y, stop_on_zero_return=False)

print(f"Original X shape: {X.shape}")
print(f"Clean X shape: {X_clean.shape}")
print(f"Clean X:\n{X_clean}")
print(f"Clean y: {y_clean}")
```

```
Original X shape: (3, 2)
Clean X shape: (1, 2)
Clean X:
[[1 2]]
Clean y: [1.]
```

1.6.3. 5.3 _handle_NA_new_points()

Purpose: Handle NaN/inf values in new evaluation points during main loop

Used by: optimize() after evaluating new points

Calls: `_apply_penalty_NA()` and `_remove_nan()`

Returns: None, None if all evaluations invalid (skip iteration)

```
opt = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    n_initial=5,
    verbose=True,
    seed=42
)

# Simulate optimization state
opt.y_ = np.array([1.0, 2.0, 3.0]) # Historical values
opt.n_iter_ = 1

# Case 1: Some valid values
print("Case 1: Some valid evaluations")
x_next = np.array([[1, 2], [3, 4], [5, 6]])
y_next = np.array([5.0, np.nan, 10.0])
x_clean, y_clean = opt._handle_NA_new_points(x_next, y_next)
print(f"Valid points remaining: {x_clean.shape[0] if x_clean is not None else 0}")

# Case 2: All invalid - should return None
print("\nCase 2: All invalid evaluations")
x_all_bad = np.array([[1, 2], [3, 4]])
y_all_bad = np.array([np.nan, np.inf])
x_clean, y_clean = opt._handle_NA_new_points(x_all_bad, y_all_bad)
print(f"Result: {'None (skip iteration)' if x_clean is None else 'Valid points'}")
```

TensorBoard logging disabled

Case 1: Some valid evaluations

Warning: Found 1 NaN/inf value(s), replacing with adaptive penalty (max + 3*std = 6.0000)

Valid points remaining: 3

Case 2: All invalid evaluations

Warning: Found 2 NaN/inf value(s), replacing with adaptive penalty (max + 3*std = 6.0000)

Result: Valid points

1.7. 6. Main Loop Update Methods

1.7.1. 6.1 _update_best_main_loop()

Purpose: Update best solution found during main optimization loop

Used by: optimize() after each iteration

Updates: self.best_x_ and self.best_y_ if improvement found

```
opt = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    n_initial=5,
    verbose=True,
    seed=42
)

# Simulate optimization state
opt.n_iter_ = 1
opt.best_x_ = np.array([1.0, 1.0])
opt.best_y_ = 2.0

# Case 1: New best found
print("Case 1: New best found")
x_new = np.array([[0.1, 0.1], [0.5, 0.5]])
y_new = np.array([0.02, 0.5])
opt._update_best_main_loop(x_new, y_new)
print(f"Updated best_y: {opt.best_y_}\n")

# Case 2: No improvement
print("Case 2: No improvement")
opt.n_iter_ = 2
x_no_improve = np.array([[1.5, 1.5]])
y_no_improve = np.array([4.5])
opt._update_best_main_loop(x_no_improve, y_no_improve)
print(f"Best_y unchanged: {opt.best_y_}")
```

```
TensorBoard logging disabled
Case 1: New best found
Iteration 1: New best f(x) = 0.020000
Updated best_y: 0.02
```

```
Case 2: No improvement
Iteration 2: f(x) = 4.500000
Best_y unchanged: 0.02
```


1.8. 7. Termination Method

1.8.1. 7.1 _determine_termination()

Purpose: Determine termination reason for optimization

Used by: optimize() at the end

Checks: Max iterations, time limit, or successful completion

```
opt = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=20,
    max_time=10.0,
    seed=42
)

# Case 1: Maximum evaluations reached
print("Case 1: Maximum evaluations reached")
opt.y_ = np.zeros(20)
start_time = time.time()
msg = opt._determine_termination(start_time)
print(f"Message: {msg}\n")

# Case 2: Time limit exceeded (simulated)
print("Case 2: Time limit exceeded")
opt.y_ = np.zeros(10)
start_time = time.time() - 700 # 11.67 minutes ago
msg = opt._determine_termination(start_time)
print(f"Message: {msg}\n")

# Case 3: Successful completion
print("Case 3: Successful completion")
opt.y_ = np.zeros(10)
start_time = time.time()
msg = opt._determine_termination(start_time)
print(f"Message: {msg}")
```

Case 1: Maximum evaluations reached

Message: Optimization terminated: maximum evaluations (20) reached

Case 2: Time limit exceeded

Message: Optimization terminated: time limit (10.00 min) reached

1. SpotOptim Internal Methods Examples

Case 3: Successful completion

Message: Optimization finished successfully

1.9. 8. Utility Methods

1.9.1. 8.1 _select_new()

Purpose: Select rows from A that are not in X (avoid duplicate evaluations)

Used by: `_suggest_next_point()` to ensure new points are different from evaluated points

```
opt = SpotOptim(  
    fun=sphere,  
    bounds=[(-5, 5)],  
    seed=42  
)  
  
A = np.array([[1, 2], [3, 4], [5, 6]])  
X = np.array([[3, 4], [7, 8]])  
  
new_A, is_new = opt._select_new(A, X)  
  
print(f"Candidate points A:\n{A}")  
print(f"\nKnown points X:\n{X}")  
print(f"\nNew points from A:\n{new_A}")  
print(f"\nIs new mask: {is_new}")
```

Candidate points A:

```
[[1 2]  
 [3 4]  
 [5 6]]
```

Known points X:

```
[[3 4]  
 [7 8]]
```

New points from A:

```
[[1 2]  
 [5 6]]
```

Is new mask: [True False True]

1.9.2. 8.2 _repair_non_numeric()**Purpose:** Round non-numeric values (int, factor) to integers**Used by:** Various methods to ensure integer/factor variables have valid values

```

opt = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    var_type=['int', 'float'],
    seed=42
)

X = np.array([[1.2, 2.5], [3.7, 4.1], [5.9, 6.8]])
X_repaired = opt._repair_non_numeric(X.copy(), opt.var_type)

print(f"Original X:\n{X}")
print(f"\nRepaired X (first column rounded to int):\n{X_repaired}")

```

Original X:

```

[[1.2 2.5]
 [3.7 4.1]
 [5.9 6.8]]

```

Repaired X (first column rounded to int):

```

[[1. 2.5]
 [4. 4.1]
 [6. 6.8]]

```

1.9.3. 8.3 _map_to_factor_values()**Purpose:** Map internal integer values to original factor strings**Used by:** optimize() when preparing results for user**Handles:** Factor (categorical) variables

```

opt = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), ('red', 'green', 'blue')],
    var_type=['float', 'factor'],
    seed=42
)

# Internal representation (integers for factors)
X_internal = np.array([[1.0, 0], [2.0, 1], [3.0, 2]])

```

1. SpotOptim Internal Methods Examples

```
X_mapped = opt._map_to_factor_values(X_internal)

print(f"Internal representation:\n{X_internal}")
print(f"\nMapped to factor values:\n{X_mapped}")
```

```
Internal representation:
[[1. 0.]
 [2. 1.]
 [3. 2.]]
```

```
Mapped to factor values:
[[1.0 'red']
 [2.0 'green']
 [3.0 'blue']]
```

1.10. 9. Integration Examples

1.10.1. 9.1 Optimization with Custom Initial Design

```
opt = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=15,
    seed=42,
    verbose=True
)

# Provide custom initial design
X0 = np.array([[0, 0], [1, 1], [2, 2], [-1, -1], [-2, -2]])
result = opt.optimize(X0=X0)

print(f"\nBest point: {result.x}")
print(f"Best value: {result.fun:.6f}")
```

```
TensorBoard logging disabled
Note: Initial design size (5) is smaller than requested (10) due to NaN/inf values
Initial best: f(x) = 0.000000
Iteration 1: f(x) = 0.000000
Iteration 2: f(x) = 0.000000
Iteration 3: f(x) = 0.000000
```

Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.

Iteration 4: $f(x) = 7.878776$
Iteration 5: $f(x) = 0.000000$
Iteration 6: $f(x) = 0.099268$
Iteration 7: $f(x) = 0.003575$
Iteration 8: $f(x) = 0.003187$
Iteration 9: $f(x) = 0.000019$
Iteration 10: $f(x) = 0.000001$

Best point: [0 0]

Best value: 0.000000

1.10.2. 9.2 Optimization with Starting Point

```
opt = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=15,
    n_initial=5,
    x0=[0.5, 0.5], # Starting point near optimum
    seed=42,
    verbose=True
)

result = opt.optimize()

print(f"\nBest point: {result.x}")
print(f"Best value: {result.fun:.6f}")
```

Starting point x0 validated and processed successfully.

Original scale: [0.5 0.5]

Internal scale: [0.5 0.5]

TensorBoard logging disabled

Including starting point x0 in initial design as first evaluation.

Initial best: $f(x) = 0.500000$

Iteration 1: $f(x) = 0.691041$

Iteration 2: New best $f(x) = 0.149351$

Iteration 3: New best $f(x) = 0.015439$

Iteration 4: New best $f(x) = 0.000530$

Iteration 5: New best $f(x) = 0.000070$

Iteration 6: New best $f(x) = 0.000004$

1. SpotOptim Internal Methods Examples

```
Iteration 7: New best f(x) = 0.000003
Iteration 8: New best f(x) = 0.000003
Iteration 9: New best f(x) = 0.000003
Iteration 10: f(x) = 0.000003
```

```
Best point: [0.00169077 0.0001854 ]
Best value: 0.000003
```

1.10.3. 9.3 Optimization with Integer Variables

```
opt = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    var_type=['int', 'int'],
    max_iter=20,
    n_initial=10,
    seed=42,
    verbose=True
)

result = opt.optimize()

print(f"\nBest point: {result.x}")
print(f"Best value: {result.fun:.6f}")
print(f"Point has integers: {np.all(result.x == np.round(result.x))}")
```

```
TensorBoard logging disabled
Initial best: f(x) = 4.000000
  Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 1: f(x) = 29.000000
  Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 2: New best f(x) = 1.000000
Iteration 3: New best f(x) = 0.000000
  Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 4: f(x) = 25.000000
  Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 5: f(x) = 9.000000
  Attempt 2/10: Previous point was duplicate after rounding, trying fallback
```

```

Acquisition failure: Using random space-filling design as fallback.
Iteration 6: f(x) = 5.000000
  Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 7: f(x) = 13.000000
  Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 8: f(x) = 13.000000
  Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 9: f(x) = 10.000000
  Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 10: f(x) = 8.000000

Best point: [ 0. -0.]
Best value: 0.000000
Point has integers: True

```

1.10.4. 9.4 Noisy Function Optimization with OCBA

```

# Noisy sphere function
def noisy_sphere(X):
    X = np.atleast_2d(X)
    return np.sum(X**2, axis=1) + np.random.normal(0, 0.1, X.shape[0])

opt = SpotOptim(
    fun=noisy_sphere,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=25,
    n_initial=10,
    repeats_initial=2, # Enable noise handling
    ocba_delta=5, # Enable OCBA with 5 re-evaluations
    seed=42,
    verbose=True
)

result = opt.optimize()

print(f"\nBest point: {result.x}")
print(f"Best value: {result.fun:.6f}")
print(f>Note: OCBA re-evaluated promising points to reduce uncertainty")

```

1. SpotOptim Internal Methods Examples

```
TensorBoard logging disabled
Initial best: f(x) = 2.515129, mean best: f(x) = 2.528228

In _get_ocba():
means: [25.98208182 11.38525607 10.14299112  2.52822799 20.7340389  21.54008811
        16.41407594 14.27200231 18.24226149 20.03621041]
vars:  [1.88322602e-02 3.25775173e-04 4.44120528e-03 1.71576414e-04
        4.13689680e-03 1.54846106e-02 2.08413843e-02 1.69875196e-04
        2.48234692e-04 6.43994670e-07]
delta: 5
n_designs: 10
Ratios: [4.46977812e-01 5.42192978e-02 1.00000000e+00 2.51684038e-01
        1.62954583e-01 5.59323435e-01 1.41121667e+00 1.60814990e-02
        1.31249929e-02 2.74297606e-05]
Best: 3, Second best: 2
OCBA: Adding 5 re-evaluation(s)
Iteration 1: mean f(x) = 15.485008

Best point: [-1.55458479 -0.06381726]
Best value: 2.515129
Note: OCBA re-evaluated promising points to reduce uncertainty
```

1.10.5. 9.5 Optimization with NaN Handling

```
# Function that sometimes returns NaN
def sometimes_nan(X):
    X = np.atleast_2d(X)
    y = np.sum(X**2, axis=1)
    # Return NaN for large values (penalty will be applied)
    y[y > 100] = np.nan
    return y

opt = SpotOptim(
    fun=sometimes_nan,
    bounds=[(-10, 10), (-10, 10)],
    max_iter=20,
    n_initial=10,
    seed=42,
    verbose=True
)

result = opt.optimize()
```


1.11. 10. Method Relationship Diagram

```
print(f"\nBest point: {result.x}")
print(f"Best value: {result.fun:.6f}")
print(f"Optimization succeeded despite NaN values: {result.success}")
```

TensorBoard logging disabled

Warning: 1 initial design point(s) returned NaN/inf and will be ignored (reduced from 10 to 9 points)

Note: Initial design size (9) is smaller than requested (10) due to NaN/inf values

Initial best: $f(x) = 9.683226$

Iteration 1: New best $f(x) = 8.496828$

Iteration 2: New best $f(x) = 4.452328$

Iteration 3: New best $f(x) = 0.174337$

Iteration 4: New best $f(x) = 0.038871$

Iteration 5: New best $f(x) = 0.000492$

Iteration 6: $f(x) = 0.000649$

Iteration 7: New best $f(x) = 0.000114$

Iteration 8: New best $f(x) = 0.000003$

Iteration 9: New best $f(x) = 0.000002$

Iteration 10: $f(x) = 0.000002$

Iteration 11: $f(x) = 0.000002$

Best point: $[-0.00082295 \ -0.00113022]$

Best value: 0.000002

Optimization succeeded despite NaN values: True

1.11. 10. Method Relationship Diagram

Here's how all the methods relate to each other in the optimization workflow:

`optimize()`

Initial Design Phase

`get_initial_design()`

`_generate_initial_design()` [if X_0 is None]

`_curate_initial_design()`

`_evaluate_function()`

`_handle_NA_initial_design()`

`_check_size_initial_design()`

`_get_best_xy_initial_design()`

Main Optimization Loop (while not terminated)

`_fit_surrogate()`

1. SpotOptim Internal Methods Examples

```
        _selection_dispatcher() [if max_surrogate_points exceeded]

        _apply_ocba() [if noise=True and ocba_delta > 0]

        _suggest_next_point()
            _acquisition_function()
            _predict_with_uncertainty()
            _repair_non_numeric()
            _select_new()
            _handle_acquisition_failure() [if needed]

        _evaluate_function()

        _handle_NA_new_points()
            _apply_penalty_NA()
            _remove_nan()

        _update_best_main_loop()

Termination Phase
    _determine_termination()
    _map_to_factor_values() [for results]
```

1.12. 11. Summary

This notebook demonstrated all major methods in SpotOptim with executable examples:

Core Flow: 1. **Initial Design:** Generate/process → Curate → Evaluate → Handle NaN → Validate → Get best 2. **Main Loop:** Fit surrogate → Apply OCBA → Suggest point → Evaluate → Handle NaN → Update best 3. **Termination:** Determine reason → Prepare results

Key Features: - Automatic handling of NaN/inf values with penalties - Support for noisy functions with OCBA re-evaluation - Integer and factor variable support - Flexible initial design (LHS, custom, or with starting point) - Multiple acquisition functions (EI, PI, mean) - Termination by max iterations or time limit

All examples can be run independently and demonstrate the modular design of SpotOptim!

1.13. Jupyter Notebook

i Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

2. Optimizing the Aircraft Wing Weight Example

i Note

- This section demonstrates optimization of the Aircraft Wing Weight Example (AWWE) function
- We compare three optimization methods:
 - **SpotOptim**: Bayesian optimization with Gaussian Process surrogate
 - **Nelder-Mead**: Derivative-free simplex method from `scipy.optimize`
 - **BFGS**: Quasi-Newton method from `scipy.optimize`
- The following Python packages are imported:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy.optimize import minimize
from spotoptim import SpotOptim
import time
import pprint
```

2.1. The AWWE Objective Function

We use the same AWWE function from Chapter 64, which models the weight of an unpainted light aircraft wing. The function accepts inputs in the unit cube $[0, 1]^9$ and returns the wing weight.

```
def wingwt(x):
    """
    Aircraft Wing Weight function.

    Args:
        x: array-like of 9 values in [0,1]
```

2. Optimizing the Aircraft Wing Weight Example

```
[Sw, Wfw, A, L, q, l, Rtc, Nz, Wdg]

Returns:
    Wing weight (scalar)
    """
    # Ensure x is a 2D array for batch evaluation
    x = np.atleast_2d(x)

    # Transform from unit cube to natural scales
    Sw = x[:, 0] * (200 - 150) + 150
    Wfw = x[:, 1] * (300 - 220) + 220
    A = x[:, 2] * (10 - 6) + 6
    L = (x[:, 3] * (10 - (-10)) - 10) * np.pi/180
    q = x[:, 4] * (45 - 16) + 16
    l = x[:, 5] * (1 - 0.5) + 0.5
    Rtc = x[:, 6] * (0.18 - 0.08) + 0.08
    Nz = x[:, 7] * (6 - 2.5) + 2.5
    Wdg = x[:, 8] * (2500 - 1700) + 1700

    # Calculate weight on natural scale
    W = 0.036 * Sw**0.758 * Wfw**0.0035 * (A/np.cos(L)**2)**0.6 * q**0.006
    W = W * l**0.04 * (100*Rtc/np.cos(L))**(-0.3) * (Nz*Wdg)**(0.49)

    return W.ravel()

# Wrapper for scipy.optimize (expects 1D input, returns scalar)
def wingwt_scipy(x):
    return float(wingwt(x.reshape(1, -1))[0])
```

2.2. Baseline Configuration

The baseline Cessna C172 Skyhawk configuration (coded in unit cube):

```
baseline_coded = np.array([0.48, 0.4, 0.38, 0.5, 0.62, 0.344, 0.4, 0.37, 0.38])
baseline_weight = wingwt(baseline_coded)[0]
print(f"Baseline wing weight: {baseline_weight:.2f} lb")
```

Baseline wing weight: 233.91 lb

2.3. Optimization Setup

We'll optimize the AWE function starting from the baseline configuration using three different methods:

1. **SpotOptim**: Bayesian optimization (good for expensive black-box functions)
2. **Nelder-Mead**: Derivative-free simplex method (robust but can be slow)
3. **BFGS**: Quasi-Newton method (fast but requires smooth functions)

```
# Starting point (baseline configuration)
x0 = baseline_coded.copy()

# Bounds for all methods (unit cube)
bounds = [(0, 1)] * 9

# Number of function evaluations budget
max_evals = 30

print(f"Starting point: {x0}")
print(f"Starting weight: {baseline_weight:.2f} lb")
```

```
Starting point: [0.48  0.4   0.38  0.5   0.62  0.344 0.4   0.37  0.38 ]
Starting weight: 233.91 lb
```

2.4. Method 1: SpotOptim (Surrogate Model Based Optimization)

```
# Start timing
start_time = time.time()

# Configure SpotOptim
optimizer_spot = SpotOptim(
    fun=wingwt,
    bounds=bounds,
    x0=None,
    max_iter=max_evals,
    n_initial=10, # Initial design points
    var_name=['Sw', 'Wfw', 'A', 'L', 'q', 'l', 'Rtc', 'Nz', 'Wdg'],
    acquisition='y', # ei: Expected Improvement
    max_surrogate_points=100,
    seed=42,
```

2. Optimizing the Aircraft Wing Weight Example

```
verbose=True,  
tensorboard_log=True,  
tensorboard_clean=True  
)
```

```
Removed old TensorBoard logs: runs/spotoptim_20251220_094012  
Cleaned 1 old TensorBoard log directory  
TensorBoard logging enabled: runs/spotoptim_20251220_094024
```

2.5. Design Table

```
pprint.pprint(optimizer_spot.print_design_table())
```

name	type	lower	upper	default	transform
Sw	float	0.0000	1.0000	0.5000	-
Wfw	float	0.0000	1.0000	0.5000	-
A	float	0.0000	1.0000	0.5000	-
L	float	0.0000	1.0000	0.5000	-
q	float	0.0000	1.0000	0.5000	-
l	float	0.0000	1.0000	0.5000	-
Rtc	float	0.0000	1.0000	0.5000	-
Nz	float	0.0000	1.0000	0.5000	-
Wdg	float	0.0000	1.0000	0.5000	-

```
(' name | type | lower | upper | default | transform | \n'  
' |-----|-----|-----|-----|-----|-----| \n'  
' | Sw | float | 0.0000 | 1.0000 | 0.5000 | - | \n'  
' | Wfw | float | 0.0000 | 1.0000 | 0.5000 | - | \n'  
' | A | float | 0.0000 | 1.0000 | 0.5000 | - | \n'  
' | L | float | 0.0000 | 1.0000 | 0.5000 | - | \n'  
' | q | float | 0.0000 | 1.0000 | 0.5000 | - | \n'  
' | l | float | 0.0000 | 1.0000 | 0.5000 | - | \n'  
' | Rtc | float | 0.0000 | 1.0000 | 0.5000 | - | \n'  
' | Nz | float | 0.0000 | 1.0000 | 0.5000 | - | \n'  
' | Wdg | float | 0.0000 | 1.0000 | 0.5000 | - | \n')
```


2.6. Run optimization

```
result_spot = optimizer_spot.optimize()
```

```
Initial best: f(x) = 212.219714
Iteration 1: New best f(x) = 142.977744
Iteration 2: New best f(x) = 140.904175
Iteration 3: New best f(x) = 120.465251
Iteration 4: f(x) = 121.575842
Iteration 5: New best f(x) = 120.101053
Iteration 6: New best f(x) = 119.997018
Iteration 7: f(x) = 120.018134
Iteration 8: New best f(x) = 119.958665
Iteration 9: f(x) = 120.386522
Iteration 10: New best f(x) = 119.691338
Iteration 11: New best f(x) = 119.530080
Iteration 12: f(x) = 119.539469
Iteration 13: New best f(x) = 119.503677
Iteration 14: f(x) = 119.503691
Iteration 15: New best f(x) = 119.503672
Iteration 16: f(x) = 119.503718
Iteration 17: f(x) = 119.503685
Iteration 18: f(x) = 119.503693
Iteration 19: f(x) = 119.503726
Iteration 20: f(x) = 119.503749
TensorBoard writer closed. View logs with: tensorboard --logdir=runs/spotoptim_20251220_094024
```

```
# End timing
spot_time = time.time() - start_time

print(f"\nSpotOptim Results:")
print(f"  Best weight: {result_spot.fun:.4f} lb")
print(f"  Function evaluations: {result_spot.nfev}")
print(f"  Time elapsed: {spot_time:.2f} seconds")
print(f"  Success: {result_spot.success}")
```

```
SpotOptim Results:
  Best weight: 119.5037 lb
  Function evaluations: 30
  Time elapsed: 5.92 seconds
  Success: True
```

2. Optimizing the Aircraft Wing Weight Example

```
optimizer_spot.print_best()
```

Best Solution Found:

```
-----  
Sw: 0.0000  
Wfw: 0.0000  
A: 0.0000  
L: 0.5000  
q: 0.0000  
l: 0.0000  
Rtc: 1.0000  
Nz: 0.0000  
Wdg: 0.0000  
Objective Value: 119.5037  
Total Evaluations: 30
```

2.7. Result Table

```
pprint.pprint(optimizer_spot.print_results_table(show_importance=True))
```

name	type	default	lower	upper	tuned	transform	importance
Sw	float	0.5000	0.0000	1.0000	0.0000	-	9
Wfw	float	0.5000	0.0000	1.0000	0.0000	-	12
A	float	0.5000	0.0000	1.0000	0.0000	-	14
L	float	0.5000	0.0000	1.0000	0.5000	-	4
q	float	0.5000	0.0000	1.0000	0.0000	-	6
l	float	0.5000	0.0000	1.0000	0.0000	-	11
Rtc	float	0.5000	0.0000	1.0000	1.0000	-	13
Nz	float	0.5000	0.0000	1.0000	0.0000	-	14
Wdg	float	0.5000	0.0000	1.0000	0.0000	-	14

Interpretation: ***: >99%, **: >75%, *: >50%, .: >10%

```
('| name | type | default | lower | upper | tuned | transform |  
| importance | stars | \n'  
'|-----|-----|-----|-----|-----|-----|-----|  
|-----|-----| \n'  
'| Sw | float | 0.5000 | 0.0000 | 1.0000 | 0.0000 | - |
```

2.8. Progress of the Optimization

```
'|          9.52 |          |\n'| Wfw      | float |    0.5000 | 0.0000 | 1.0000 | 0.0000 | -      '|
'|          12.76 | .      |\n'| A        | float |    0.5000 | 0.0000 | 1.0000 | 0.0000 | -      '|
'|          14.11 | .      |\n'| L        | float |    0.5000 | 0.0000 | 1.0000 | 0.5000 | -      '|
'|          4.30 |          |\n'| q        | float |    0.5000 | 0.0000 | 1.0000 | 0.0000 | -      '|
'|          6.10 |          |\n'| l        | float |    0.5000 | 0.0000 | 1.0000 | 0.0000 | -      '|
'|          11.05 | .      |\n'| Rtc      | float |    0.5000 | 0.0000 | 1.0000 | 1.0000 | -      '|
'|          13.87 | .      |\n'| Nz       | float |    0.5000 | 0.0000 | 1.0000 | 0.0000 | -      '|
'|          14.03 | .      |\n'| Wdg      | float |    0.5000 | 0.0000 | 1.0000 | 0.0000 | -      '|
'|          14.27 | .      |\n'|          |\n'|
'Interpretation: ***: >99%, **: >75%, *: >50%, .: >10%')
```

2.8. Progress of the Optimization

```
optimizer_spot.plot_progress(log_y=False)
```

2. Optimizing the Aircraft Wing Weight Example



2.9. Contour Plots of Most Important Hyperparameters

```
optimizer_spot.plot_important_hyperparameter_contour(max_imp=3)
```

Plotting surrogate contours for top 3 most important parameters:

Wdg: importance = 14.27% (type: float)

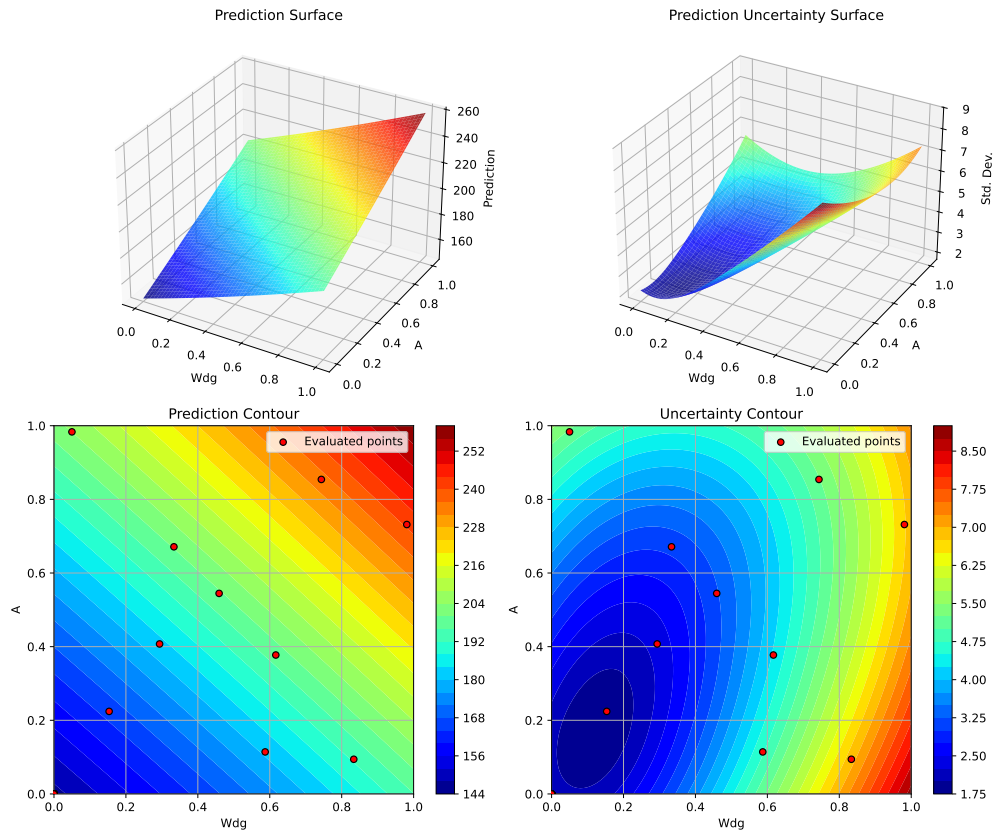
A: importance = 14.11% (type: float)

Nz: importance = 14.03% (type: float)

Generating 3 surrogate plots...

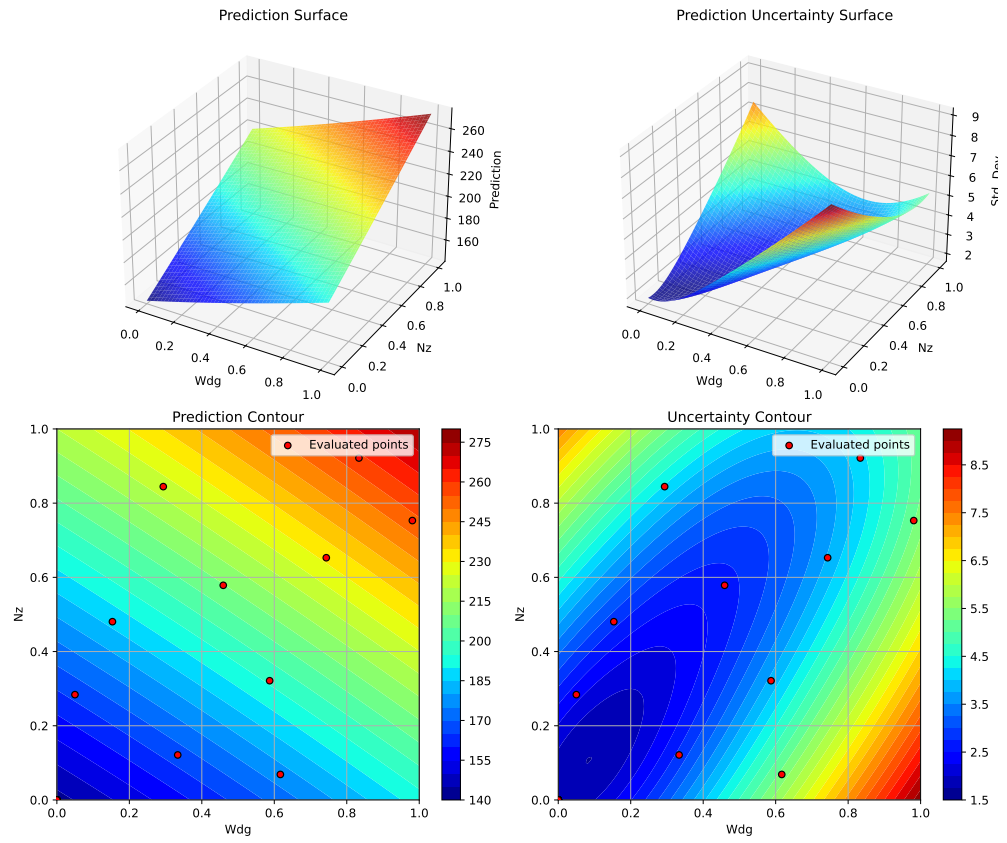
Plotting Wdg vs A

2.9. Contour Plots of Most Important Hyperparameters



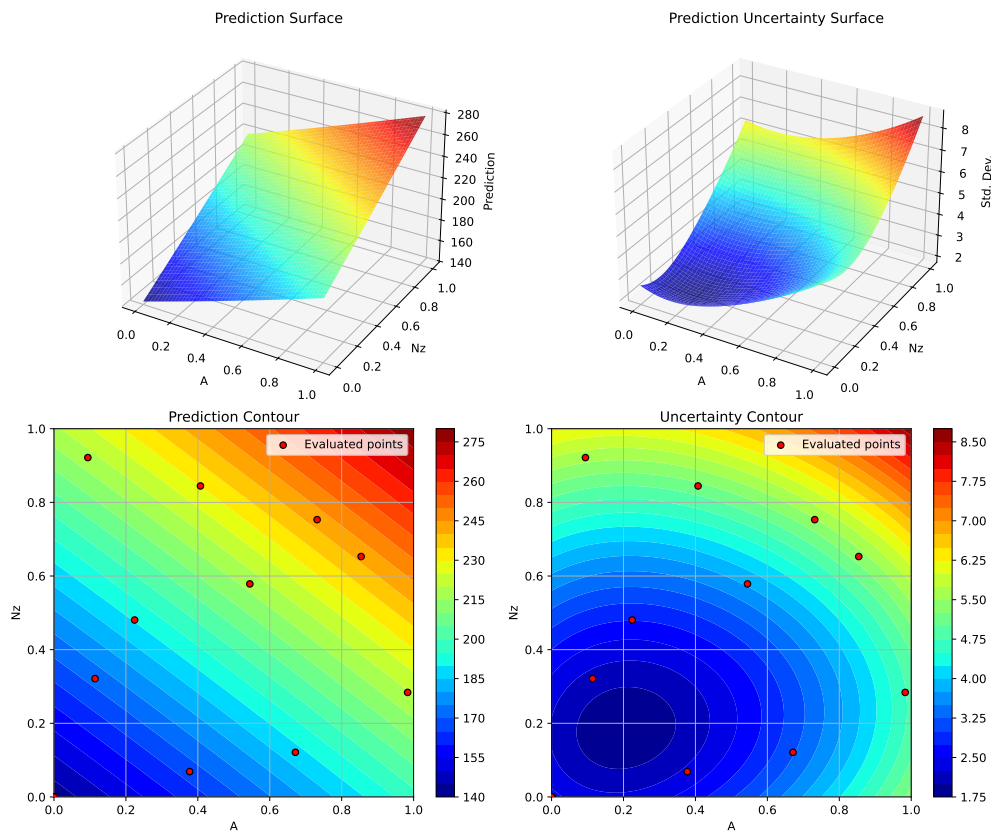
Plotting Wdg vs Nz

2. Optimizing the Aircraft Wing Weight Example



Plotting A vs Nz

2.10. Method 2: Nelder-Mead Simplex



2.10. Method 2: Nelder-Mead Simplex

```
print("\n" + "=" * 60)
print("Running Nelder-Mead Simplex...")
print("=" * 60)

# Start timing
start_time = time.time()

# Run optimization
result_nm = minimize(
    wingwt_scipy,
    x0=x0,
    method='Nelder-Mead',
```

2. Optimizing the Aircraft Wing Weight Example

```
        bounds=bounds,
        options={'maxfev': max_evals, 'disp': True}
    )

    # End timing
    nm_time = time.time() - start_time

    print(f"\nNelder-Mead Results:")
    print(f"    Best weight: {result_nm.fun:.4f} lb")
    print(f"    Function evaluations: {result_nm.nfev}")
    print(f"    Time elapsed: {nm_time:.2f} seconds")
    print(f"    Success: {result_nm.success}")
```

```
=====
Running Nelder-Mead Simplex...
=====
```

```
Nelder-Mead Results:
    Best weight: 220.5449 lb
    Function evaluations: 30
    Time elapsed: 0.00 seconds
    Success: False
```

2.11. Method 3: BFGS (Quasi-Newton)

```
print("\n" + "=" * 60)
print("Running BFGS (Quasi-Newton)...")
print("=" * 60)

# Start timing
start_time = time.time()

# Run optimization
result_bfgs = minimize(
    wingwt_scipy,
    x0=x0,
    method='L-BFGS-B', # Bounded BFGS
    bounds=bounds,
    options={'maxfun': max_evals, 'disp': True})
```



```

)

# End timing
bfgs_time = time.time() - start_time

print(f"\nBFGS Results:")
print(f"  Best weight: {result_bfgs.fun:.4f} lb")
print(f"  Function evaluations: {result_bfgs.nfev}")
print(f"  Time elapsed: {bfgs_time:.2f} seconds")
print(f"  Success: {result_bfgs.success}")

```

```

=====
Running BFGS (Quasi-Newton)...
=====

```

```

BFGS Results:
  Best weight: 119.5037 lb
  Function evaluations: 60
  Time elapsed: 0.00 seconds
  Success: False

```

2.12. Comparison of Results

```

# Create comparison DataFrame
comparison = pd.DataFrame({
    'Method': ['Baseline', 'SpotOptim', 'Nelder-Mead', 'BFGS'],
    'Best Weight (lb)': [
        baseline_weight,
        result_spot.fun,
        result_nm.fun,
        result_bfgs.fun
    ],
    'Improvement (%)': [
        0.0,
        (baseline_weight - result_spot.fun) / baseline_weight * 100,
        (baseline_weight - result_nm.fun) / baseline_weight * 100,
        (baseline_weight - result_bfgs.fun) / baseline_weight * 100
    ],
    'Function Evals': [
        1,

```

2. Optimizing the Aircraft Wing Weight Example

```
        result_spot.nfev,
        result_nm.nfev,
        result_bfgs.nfev
    ],
    'Time (s)': [
        0.0,
        spot_time,
        nm_time,
        bfgs_time
    ],
    'Success': [
        True,
        result_spot.success,
        result_nm.success,
        result_bfgs.success
    ]
]

print("\n" + "=" * 80)
print("OPTIMIZATION COMPARISON")
print("=" * 80)
print(comparison.to_string(index=False))
print("=" * 80)
```

```
=====
OPTIMIZATION COMPARISON
=====
```

Method	Best Weight (lb)	Improvement (%)	Function Evals	Time (s)	Success
Baseline	233.908405	0.000000	1	0.000000	True
SpotOptim	119.503672	48.910057	30	5.920530	True
Nelder-Mead	220.544928	5.713124	30	0.000847	False
BFGS	119.503672	48.910057	60	0.001405	False

```
=====
```

2.13. Visualization: Convergence Plots

2.13.1. SpotOptim Convergence

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))
```

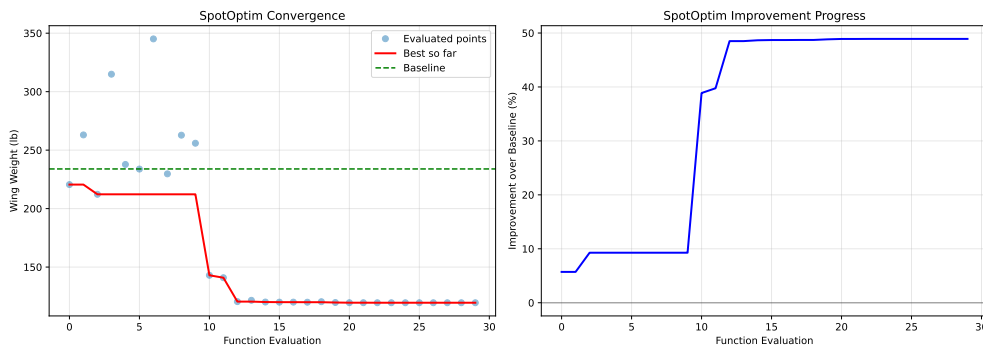
2.14. Optimal Parameter Values

```
# Plot 1: Best value over iterations
y_history = optimizer_spot.y_
best_so_far = np.minimum.accumulate(y_history)

ax1.plot(range(len(y_history)), y_history, 'o', alpha=0.5, label='Evaluated points')
ax1.plot(range(len(best_so_far)), best_so_far, 'r-', linewidth=2, label='Best so far')
ax1.axhline(y=baseline_weight, color='g', linestyle='--', label='Baseline')
ax1.set_xlabel('Function Evaluation')
ax1.set_ylabel('Wing Weight (lb)')
ax1.set_title('SpotOptim Convergence')
ax1.legend()
ax1.grid(True, alpha=0.3)

# Plot 2: Improvement over baseline
improvement = (baseline_weight - best_so_far) / baseline_weight * 100
ax2.plot(range(len(improvement)), improvement, 'b-', linewidth=2)
ax2.set_xlabel('Function Evaluation')
ax2.set_ylabel('Improvement over Baseline (%)')
ax2.set_title('SpotOptim Improvement Progress')
ax2.grid(True, alpha=0.3)
ax2.axhline(y=0, color='k', linestyle='--', alpha=0.3)

plt.tight_layout()
plt.show()
```



2.14. Optimal Parameter Values

Let's examine the optimal parameter values found by each method:

2. Optimizing the Aircraft Wing Weight Example

```
# Parameter names
param_names = ['Sw', 'Wfw', 'A', 'L', 'q', 'l', 'Rtc', 'Nz', 'Wdg']

# Transform from unit cube to natural scales
def decode_params(x):
    scales = [
        (150, 200),      # Sw
        (220, 300),      # Wfw
        (6, 10),          # A
        (-10, 10),        # L (degrees)
        (16, 45),          # q
        (0.5, 1),          # l
        (0.08, 0.18),     # Rtc
        (2.5, 6),          # Nz
        (1700, 2500)      # Wdg
    ]
    decoded = []
    for i, (low, high) in enumerate(scales):
        decoded.append(x[i] * (high - low) + low)
    return decoded

# Create comparison table
baseline_decoded = decode_params(baseline_coded)
spot_decoded = decode_params(result_spot.x)
nm_decoded = decode_params(result_nm.x)
bfgs_decoded = decode_params(result_bfgs.x)

param_comparison = pd.DataFrame({
    'Parameter': param_names,
    'Baseline': baseline_decoded,
    'SpotOptim': spot_decoded,
    'Nelder-Mead': nm_decoded,
    'BFGS': bfgs_decoded
})

print("\n" + "=" * 100)
print("OPTIMAL PARAMETER VALUES (Natural Scale)")
print("=" * 100)
print(param_comparison.to_string(index=False))
print("=" * 100)
```

=====

OPTIMAL PARAMETER VALUES (Natural Scale)

Parameter	Baseline	SpotOptim	Nelder-Mead	BFGS
Sw	174.000	150.000000	173.923947	150.00
Wfw	252.000	220.000000	254.534988	220.00
A	7.520	6.000000	7.484111	6.00
L	0.000	0.000531	-0.126137	0.00
q	33.980	16.000000	34.156818	16.00
l	0.672	0.500000	0.674613	0.50
Rtc	0.120	0.180000	0.124872	0.18
Nz	3.795	2.500000	3.427058	2.50
Wdg	2004.000	1700.000000	2028.938270	1700.00

2.15. Analysis of Optimal Solutions

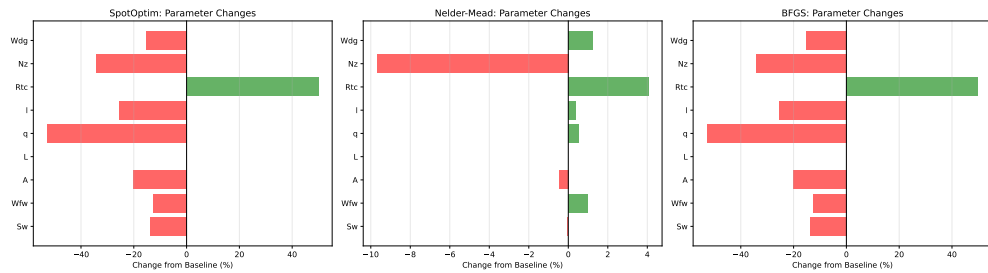
```
# Calculate percentage changes from baseline
changes_spot = [(spot_decoded[i] - baseline_decoded[i]) / baseline_decoded[i] * 100
                 for i in range(len(param_names))]
changes_nm = [(nm_decoded[i] - baseline_decoded[i]) / baseline_decoded[i] * 100
              for i in range(len(param_names))]
changes_bfgs = [(bfgs_decoded[i] - baseline_decoded[i]) / baseline_decoded[i] * 100
                for i in range(len(param_names))]

# Visualization
fig, axes = plt.subplots(1, 3, figsize=(18, 5))

for ax, changes, method in zip(axes,
                                [changes_spot, changes_nm, changes_bfgs],
                                ['SpotOptim', 'Nelder-Mead', 'BFGS']):
    colors = ['red' if c < 0 else 'green' for c in changes]
    ax.barh(param_names, changes, color=colors, alpha=0.6)
    ax.axvline(x=0, color='black', linestyle='--', linewidth=0.5)
    ax.set_xlabel('Change from Baseline (%)')
    ax.set_title(f'{method}: Parameter Changes')
    ax.grid(True, alpha=0.3, axis='x')

plt.tight_layout()
plt.show()
```

2. Optimizing the Aircraft Wing Weight Example



2.16. Key Insights from Optimal Solutions

```
# Find parameters with largest changes for each method
def analyze_changes(decoded, baseline_decoded, method_name):
    changes = {param_names[i]: decoded[i] - baseline_decoded[i]
               for i in range(len(param_names))}
    sorted_changes = sorted(changes.items(), key=lambda x: abs(x[1]), reverse=True)

    print(f"\n{method_name} - Top 5 Parameter Changes:")
    print("-" * 50)
    for param, change in sorted_changes[:5]:
        idx = param_names.index(param)
        pct = change / baseline_decoded[idx] * 100
        print(f" {param:5s}: {change:+8.2f} ( {pct:+6.1f}% )")

analyze_changes(spot_decoded, baseline_decoded, "SpotOptim")
analyze_changes(nm_decoded, baseline_decoded, "Nelder-Mead")
analyze_changes(bfgs_decoded, baseline_decoded, "BFGS")
```

SpotOptim - Top 5 Parameter Changes:

```
-----
Wdg  :  -304.00 (  -15.2%)
Wfw  :  -32.00 (  -12.7%)
Sw   :  -24.00 (  -13.8%)
q    :  -17.98 ( -52.9%)
A    :   -1.52 (  -20.2%)
```

Nelder-Mead - Top 5 Parameter Changes:

```
-----
Wdg  :   +24.94 (   +1.2%)
Wfw  :   +2.53 (   +1.0%)
```

2.17. Method Efficiency Comparison

Nz : -0.37 (-9.7%)
q : +0.18 (+0.5%)
L : -0.13 (-inf%)

BFGS - Top 5 Parameter Changes:

Wdg : -304.00 (-15.2%)
Wfw : -32.00 (-12.7%)
Sw : -24.00 (-13.8%)
q : -17.98 (-52.9%)
A : -1.52 (-20.2%)

2.17. Method Efficiency Comparison

```
# Calculate efficiency metrics
efficiency = pd.DataFrame({
    'Method': ['SpotOptim', 'Nelder-Mead', 'BFGS'],
    'Weight Reduction (lb)': [
        baseline_weight - result_spot.fun,
        baseline_weight - result_nm.fun,
        baseline_weight - result_bfgs.fun
    ],
    'Evals to Best': [
        np.argmin(optimizer_spot.y_) + 1,
        result_nm.nfev,
        result_bfgs.nfev
    ],
    'Time per Eval (ms)': [
        spot_time / result_spot.nfev * 1000,
        nm_time / result_nm.nfev * 1000,
        bfgs_time / result_bfgs.nfev * 1000
    ]
})

print("\n" + "=" * 80)
print("METHOD EFFICIENCY METRICS")
print("=" * 80)
print(efficiency.to_string(index=False))
print("=" * 80)
```

=====

2. Optimizing the Aircraft Wing Weight Example

METHOD EFFICIENCY METRICS

Method	Weight Reduction (lb)	Evals to Best	Time per Eval (ms)
SpotOptim	114.404734	25	197.351011
Nelder-Mead	13.363478	30	0.028237
BFGS	114.404734	60	0.023421

2.18. Visualization: 2D Slices of Optimal Solutions

Let's visualize how the optimal solutions compare in the most important 2D subspaces:

```
# Create 2D slices showing optimal points
fig, axes = plt.subplots(2, 2, figsize=(14, 12))

# Important parameter pairs based on sensitivity analysis
pairs = [
    (7, 2), # Nz vs A (load factor vs aspect ratio)
    (0, 8), # Sw vs Wdg (wing area vs gross weight)
    (7, 0), # Nz vs Sw (load factor vs wing area)
    (2, 6)  # A vs Rtc (aspect ratio vs thickness ratio)
]

for ax, (i, j) in zip(axes.flat, pairs):
    # Create meshgrid for contour plot
    x_range = np.linspace(0, 1, 50)
    y_range = np.linspace(0, 1, 50)
    X, Y = np.meshgrid(x_range, y_range)

    # Evaluate function on grid (fixing other parameters at baseline)
    Z = np.zeros_like(X)
    for ii in range(X.shape[0]):
        for jj in range(X.shape[1]):
            point = baseline_coded.copy()
            point[i] = X[ii, jj]
            point[j] = Y[ii, jj]
            Z[ii, jj] = wingwt(point)[0]

    # Plot contours
    contour = ax.contourf(X, Y, Z, levels=20, cmap='viridis', alpha=0.6)
    ax.contour(X, Y, Z, levels=10, colors='black', alpha=0.3, linewidths=0.5)

# Plot optimal points
```


2.18. Visualization: 2D Slices of Optimal Solutions

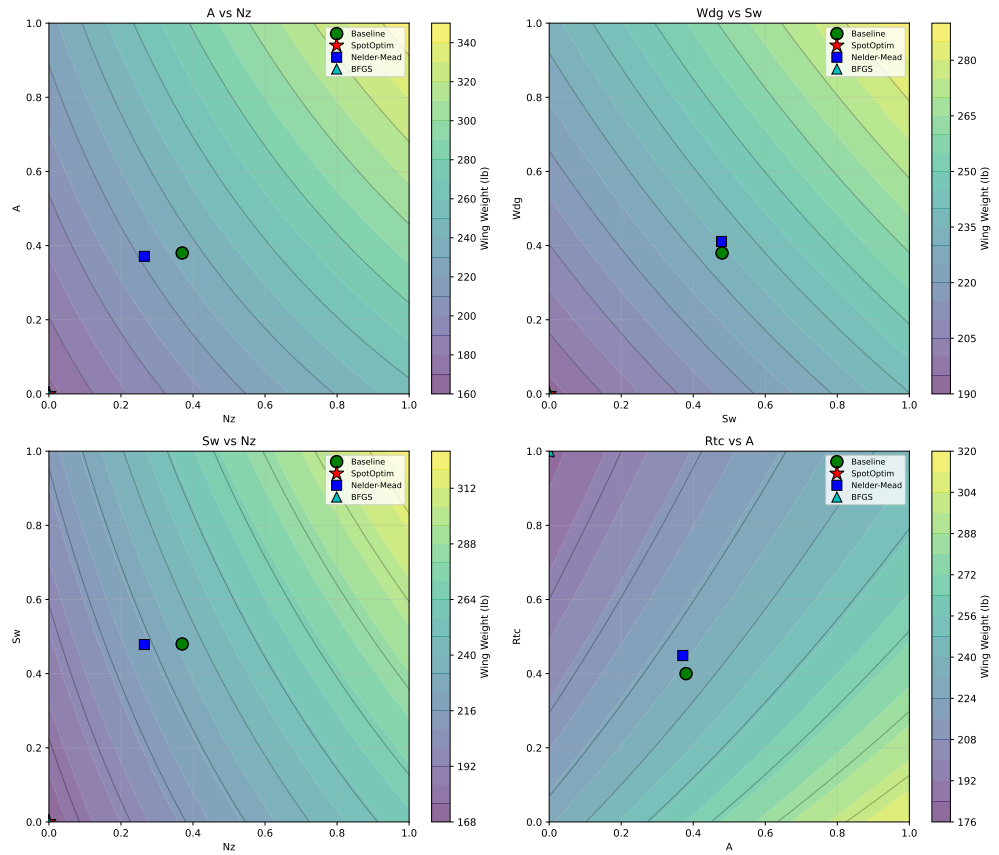
```
ax.plot(baseline_coded[i], baseline_coded[j], 'go', markersize=12,
        label='Baseline', markeredgecolor='black', markeredgewidth=1.5)
ax.plot(result_spot.x[i], result_spot.x[j], 'r*', markersize=15,
        label='SpotOptim', markeredgecolor='black', markeredgewidth=1)
ax.plot(result_nm.x[i], result_nm.x[j], 'bs', markersize=10,
        label='Nelder-Mead', markeredgecolor='black', markeredgewidth=1)
ax.plot(result_bfgs.x[i], result_bfgs.x[j], 'c^', markersize=10,
        label='BFGS', markeredgecolor='black', markeredgewidth=1)

ax.set_xlabel(param_names[i])
ax.set_ylabel(param_names[j])
ax.set_title(f'{param_names[j]} vs {param_names[i]}')
ax.legend(loc='best', fontsize=8)
ax.grid(True, alpha=0.3)

# Add colorbar
plt.colorbar(contour, ax=ax, label='Wing Weight (lb)')

plt.tight_layout()
plt.show()
```

2. Optimizing the Aircraft Wing Weight Example



2.19. Conclusion

This analysis demonstrates the application of different optimization methods to the Aircraft Wing Weight Example. Key takeaways:

1. **SpotOptim** provides efficient global optimization with good exploration of the design space
2. **Nelder-Mead** offers robust derivative-free optimization but may require more evaluations
3. **BFGS** converges quickly for smooth problems but can get trapped in local minima

For aircraft design problems with expensive simulations, Bayesian optimization (SpotOptim) offers the best balance of efficiency and solution quality, making it particularly suitable for real-world engineering applications.

2.20. Jupyter Notebook

i Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

3. Benchmarking SpotOptim with Sklearn Kriging (Matern Kernel) on 6D Rosenbrock and 10D Michalewicz Functions

i Note

These test functions were used during the Dagstuhl Seminar 25451 Bayesian Optimisation (Nov 02 – Nov 07, 2025), see [here](#).

This notebook demonstrates the use of `SpotOptim` with sklearn's Gaussian Process Regressor as a surrogate model.

3.1. SpotOptim with Sklearn Kriging in 6 Dimensions: Rosenbrock Function

This section demonstrates how to use the `SpotOptim` class with sklearn's Gaussian Process Regressor (using Matern kernel) as a surrogate on the 6-dimensional Rosenbrock function. We use a maximum of 100 function evaluations.

```
import warnings
warnings.filterwarnings("ignore")
import json
import numpy as np
from spotoptim import SpotOptim
from spotoptim.function import rosenbrock
```

3.1.1. Define the 6D Rosenbrock Function

3. Benchmarking SpotOptim with Sklearn Kriging (Matern Kernel) on 6D Rosenbrock and 10D Michalev

```
dim = 6
lower = np.full(dim, -2.0)
upper = np.full(dim, 2.0)
bounds = list(zip(lower, upper))
fun = rosenbrock
max_iter = 100
```

3.1.2. Set up SpotOptim Parameters

```
n_initial = dim
seed = 321
```

3.1.3. Sklearn Gaussian Process Regressor as Surrogate

```
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import Matern, ConstantKernel

# Use a Matern kernel instead of the standard RBF kernel
kernel = ConstantKernel(1.0, (1e-2, 1e12)) * Matern(
    length_scale=1.0,
    length_scale_bounds=(1e-4, 1e2),
    nu=2.5
)
surrogate = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=100)

# Create SpotOptim instance with sklearn surrogate
opt_rosen = SpotOptim(
    fun=fun,
    bounds=bounds,
    n_initial=n_initial,
    max_iter=max_iter,
    surrogate=surrogate,
    seed=seed,
    verbose=1
)

# Run optimization
result_rosen = opt_rosen.optimize()
```

TensorBoard logging disabled

3.1. SpotOptim with Sklearn Kriging in 6 Dimensions: Rosenbrock Function

Initial best: $f(x) = 321.834153$
Iteration 1: $f(x) = 3523.031155$
Iteration 2: $f(x) = 1535.228444$
Iteration 3: $f(x) = 326.972364$
Iteration 4: New best $f(x) = 179.355785$
Iteration 5: New best $f(x) = 147.216243$
Iteration 6: New best $f(x) = 126.871053$
Iteration 7: New best $f(x) = 106.906574$
Iteration 8: New best $f(x) = 77.679931$
Iteration 9: New best $f(x) = 67.629017$
Iteration 10: $f(x) = 70.971738$
Iteration 11: New best $f(x) = 66.969537$
Iteration 12: New best $f(x) = 66.896224$
Iteration 13: New best $f(x) = 63.379954$
Iteration 14: New best $f(x) = 54.067234$
Iteration 15: New best $f(x) = 53.621282$
Iteration 16: New best $f(x) = 53.320713$
Iteration 17: New best $f(x) = 48.663202$
Iteration 18: New best $f(x) = 48.540259$
Iteration 19: New best $f(x) = 47.725638$
Iteration 20: $f(x) = 48.258855$
Iteration 21: $f(x) = 48.197484$
Iteration 22: New best $f(x) = 46.298670$
Iteration 23: New best $f(x) = 41.616664$
Iteration 24: New best $f(x) = 40.724257$
Iteration 25: New best $f(x) = 38.350648$
Iteration 26: New best $f(x) = 37.674355$
Iteration 27: $f(x) = 38.494222$
Iteration 28: New best $f(x) = 32.960933$
Iteration 29: New best $f(x) = 29.033108$
Iteration 30: New best $f(x) = 28.179964$
Iteration 31: New best $f(x) = 24.939865$
Iteration 32: New best $f(x) = 24.333511$
Iteration 33: New best $f(x) = 22.262350$
Iteration 34: New best $f(x) = 21.939903$
Iteration 35: $f(x) = 21.963541$
Iteration 36: New best $f(x) = 19.993646$
Iteration 37: $f(x) = 20.122292$
Iteration 38: New best $f(x) = 17.966790$
Iteration 39: New best $f(x) = 15.565935$
Iteration 40: $f(x) = 16.288929$
Iteration 41: New best $f(x) = 13.197626$
Iteration 42: New best $f(x) = 11.351160$
Iteration 43: New best $f(x) = 10.922738$
Iteration 44: New best $f(x) = 9.156473$

3. Benchmarking SpotOptim with Sklearn Kriging (Matern Kernel) on 6D Rosenbrock and 10D Michalev

Iteration 45: New best $f(x)$ = 7.216227
Iteration 46: New best $f(x)$ = 6.910232
Iteration 47: $f(x)$ = 6.926496
Iteration 48: New best $f(x)$ = 6.798823
Iteration 49: New best $f(x)$ = 6.778998
Iteration 50: $f(x)$ = 6.912932
Iteration 51: New best $f(x)$ = 6.365072
Iteration 52: $f(x)$ = 6.408699
Iteration 53: New best $f(x)$ = 6.244889
Iteration 54: New best $f(x)$ = 5.704130
Iteration 55: New best $f(x)$ = 5.524463
Iteration 56: New best $f(x)$ = 5.464542
Iteration 57: New best $f(x)$ = 5.392374
Iteration 58: New best $f(x)$ = 5.349325
Iteration 59: $f(x)$ = 5.387901
Iteration 60: $f(x)$ = 5.380596
Iteration 61: $f(x)$ = 5.410148
Iteration 62: New best $f(x)$ = 5.310701
Iteration 63: New best $f(x)$ = 5.236669
Iteration 64: New best $f(x)$ = 5.205214
Iteration 65: $f(x)$ = 5.265825
Iteration 66: New best $f(x)$ = 5.177558
Iteration 67: New best $f(x)$ = 5.170012
Iteration 68: $f(x)$ = 5.199574
Iteration 69: New best $f(x)$ = 5.113514
Iteration 70: $f(x)$ = 5.116612
Iteration 71: New best $f(x)$ = 5.113489
Iteration 72: $f(x)$ = 5.124969
Iteration 73: New best $f(x)$ = 5.108980
Iteration 74: $f(x)$ = 5.129656
Iteration 75: New best $f(x)$ = 5.104135
Iteration 76: New best $f(x)$ = 5.079981
Iteration 77: New best $f(x)$ = 5.076221
Iteration 78: New best $f(x)$ = 5.058641
Iteration 79: New best $f(x)$ = 5.035672
Iteration 80: New best $f(x)$ = 5.013369
Iteration 81: New best $f(x)$ = 4.920557
Iteration 82: New best $f(x)$ = 4.873432
Iteration 83: $f(x)$ = 4.881239
Iteration 84: New best $f(x)$ = 4.869763
Iteration 85: $f(x)$ = 4.874309
Iteration 86: $f(x)$ = 4.888076
Iteration 87: New best $f(x)$ = 4.812761
Iteration 88: New best $f(x)$ = 4.757053
Iteration 89: New best $f(x)$ = 4.701481

3.1. SpotOptim with Sklearn Kriging in 6 Dimensions: Rosenbrock Function

```
Iteration 90: New best f(x) = 4.632554
Iteration 91: New best f(x) = 4.621460
Iteration 92: f(x) = 4.648070
Iteration 93: New best f(x) = 4.611398
Iteration 94: f(x) = 4.639891
```

```
print(f"[6D] Sklearn Kriging: min y = {result_rosen.fun:.4f} at x = {result_rosen.x}")
print(f"Number of function evaluations: {result_rosen.nfev}")
print(f"Number of iterations: {result_rosen.nit}")
```

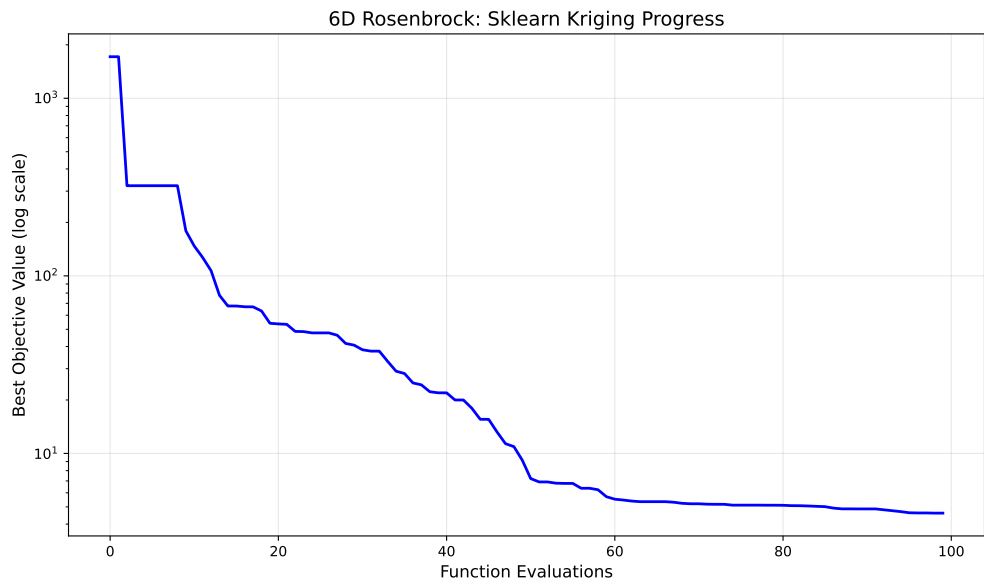
```
[6D] Sklearn Kriging: min y = 4.6114 at x = [0.17736959 0.03524346 0.0162764 0.00591552 0.02432165
Number of function evaluations: 100
Number of iterations: 94
```

3.1.4. Visualize Optimization Progress

```
import matplotlib.pyplot as plt

# Plot the optimization progress
plt.figure(figsize=(10, 6))
plt.semilogy(np.minimum.accumulate(opt_rosen.y_), 'b-', linewidth=2)
plt.xlabel('Function Evaluations', fontsize=12)
plt.ylabel('Best Objective Value (log scale)', fontsize=12)
plt.title('6D Rosenbrock: Sklearn Kriging Progress', fontsize=14)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

3. Benchmarking SpotOptim with Sklearn Kriging (Matern Kernel) on 6D Rosenbrock and 10D Michalev



3.1.5. Evaluation of Multiple Repeats

To perform 30 repeats and collect statistics:

```
# Perform 30 independent runs
n_repeats = 30
results = []

print(f"Running {n_repeats} independent optimizations...")
for i in range(n_repeats):
    kernel_i = ConstantKernel(1.0, (1e-2, 1e12)) * Matern(
        length_scale=1.0,
        length_scale_bounds=(1e-4, 1e2),
        nu=2.5
    )
    surrogate_i = GaussianProcessRegressor(kernel=kernel_i, n_restarts_optimizer=100)

    opt_i = SpotOptim(
        fun=fun,
        bounds=bounds,
        n_initial=n_initial,
        max_iter=max_iter,
        surrogate=surrogate_i,
        seed=seed + i, # Different seed for each run
```

3.2. SpotOptim with Sklearn Kriging in 10 Dimensions: Michalewicz Function

```
        verbose=0
    )

    result_i = opt_i.optimize()
    results.append(result_i.fun)

    if (i + 1) % 10 == 0:
        print(f" Completed {i + 1}/{n_repeats} runs")

# Compute statistics
mean_result = np.mean(results)
std_result = np.std(results)
min_result = np.min(results)
max_result = np.max(results)

print(f"\nResults over {n_repeats} runs:")
print(f" Mean of best values: {mean_result:.6f}")
print(f" Std of best values: {std_result:.6f}")
print(f" Min of best values: {min_result:.6f}")
print(f" Max of best values: {max_result:.6f}")
```

3.2. SpotOptim with Sklearn Kriging in 10 Dimensions: Michalewicz Function

This section demonstrates how to use the `SpotOptim` class with sklearn's Gaussian Process Regressor (using Matern kernel) as a surrogate on the 10-dimensional Michalewicz function. We use a maximum of 300 function evaluations.

3.2.1. Define the 10D Michalewicz Function

```
from spotoptim.function import michalewicz

dim = 10
lower = np.full(dim, 0.0)
upper = np.full(dim, np.pi)
bounds = list(zip(lower, upper))
fun = michalewicz
max_iter = 300
```

3.2.2. Set up SpotOptim Parameters

```
n_initial = dim
seed = 321
```

3.2.3. Sklearn Gaussian Process Regressor as Surrogate

```
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import Matern, ConstantKernel

# Use a Matern kernel instead of the standard RBF kernel
kernel = ConstantKernel(1.0, (1e-2, 1e12)) * Matern(
    length_scale=1.0,
    length_scale_bounds=(1e-4, 1e2),
    nu=2.5
)
surrogate = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=100)

# Create SpotOptim instance with sklearn surrogate
opt_micha = SpotOptim(
    fun=fun,
    bounds=bounds,
    n_initial=n_initial,
    max_iter=max_iter,
    surrogate=surrogate,
    seed=seed,
    verbose=1
)

# Run optimization
result_micha = opt_micha.optimize()
```

```
TensorBoard logging disabled
Initial best: f(x) = -1.909129
Iteration 1: f(x) = -0.471897
Iteration 2: New best f(x) = -2.778390
Iteration 3: f(x) = -1.576700
Iteration 4: f(x) = -1.695405
Iteration 5: New best f(x) = -3.210555
Iteration 6: f(x) = -2.820114
Iteration 7: f(x) = -2.982571
```

3.2. SpotOptim with Sklearn Kriging in 10 Dimensions: Michalewicz Function

Iteration 8: New best $f(x)$ = -3.366549
Iteration 9: $f(x)$ = -3.201496
Iteration 10: New best $f(x)$ = -3.432741
Iteration 11: New best $f(x)$ = -3.528267
Iteration 12: $f(x)$ = -3.504902
Iteration 13: New best $f(x)$ = -3.562886
Iteration 14: $f(x)$ = -3.547978
Iteration 15: $f(x)$ = -3.555427
Iteration 16: New best $f(x)$ = -3.584859
Iteration 17: $f(x)$ = -3.584117
Iteration 18: New best $f(x)$ = -3.594070
Iteration 19: New best $f(x)$ = -3.762200
Iteration 20: New best $f(x)$ = -4.048201
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 21: $f(x)$ = -2.167836
Iteration 22: New best $f(x)$ = -4.602166
Iteration 23: New best $f(x)$ = -5.077935
Iteration 24: New best $f(x)$ = -5.186293
Iteration 25: New best $f(x)$ = -5.203414
Iteration 26: New best $f(x)$ = -5.595463
Iteration 27: $f(x)$ = -5.457704
Iteration 28: New best $f(x)$ = -5.684386
Iteration 29: New best $f(x)$ = -5.737582
Iteration 30: New best $f(x)$ = -5.747975
Iteration 31: New best $f(x)$ = -5.748816
Iteration 32: New best $f(x)$ = -5.754623
Iteration 33: New best $f(x)$ = -5.799813
Iteration 34: New best $f(x)$ = -5.964056
Iteration 35: New best $f(x)$ = -6.022013
Iteration 36: New best $f(x)$ = -6.028901
Iteration 37: New best $f(x)$ = -6.082121
Iteration 38: New best $f(x)$ = -6.091717
Iteration 39: New best $f(x)$ = -6.095846
Iteration 40: New best $f(x)$ = -6.096010
Iteration 41: $f(x)$ = -6.049339
Iteration 42: $f(x)$ = -6.076236
Iteration 43: New best $f(x)$ = -6.122266
Iteration 44: New best $f(x)$ = -6.128134
Iteration 45: $f(x)$ = -6.125872
Iteration 46: New best $f(x)$ = -6.173651
Iteration 47: New best $f(x)$ = -6.175009
Iteration 48: New best $f(x)$ = -6.181490
Iteration 49: $f(x)$ = -6.148361
Iteration 50: New best $f(x)$ = -6.302875

3. Benchmarking SpotOptim with Sklearn Kriging (Matern Kernel) on 6D Rosenbrock and 10D Michalev

Iteration 51: $f(x) = -6.300179$
Iteration 52: $f(x) = -6.284843$
Iteration 53: New best $f(x) = -6.375193$
Iteration 54: New best $f(x) = -6.406638$
Iteration 55: New best $f(x) = -6.412983$
Iteration 56: New best $f(x) = -6.426663$
Iteration 57: New best $f(x) = -6.431333$
Iteration 58: New best $f(x) = -6.444586$
Iteration 59: New best $f(x) = -6.453929$
Iteration 60: $f(x) = -6.452940$
Iteration 61: New best $f(x) = -6.468183$
Iteration 62: New best $f(x) = -6.473056$
Iteration 63: New best $f(x) = -6.482561$
Iteration 64: New best $f(x) = -6.501237$
Iteration 65: New best $f(x) = -6.532827$
Iteration 66: New best $f(x) = -6.617722$
Iteration 67: New best $f(x) = -6.618386$
Iteration 68: $f(x) = -6.598316$
Iteration 69: New best $f(x) = -6.619635$
Iteration 70: $f(x) = -6.610526$
Iteration 71: New best $f(x) = -6.627680$
Iteration 72: New best $f(x) = -6.680090$
Iteration 73: New best $f(x) = -6.732310$
Iteration 74: New best $f(x) = -6.735648$
Iteration 75: New best $f(x) = -6.744943$
Iteration 76: New best $f(x) = -6.747280$
Iteration 77: New best $f(x) = -6.747509$
Iteration 78: New best $f(x) = -6.752044$
Iteration 79: New best $f(x) = -6.752271$
Iteration 80: New best $f(x) = -6.754208$
Iteration 81: $f(x) = -6.754075$
Iteration 82: $f(x) = -6.752283$
Iteration 83: New best $f(x) = -6.763816$
Iteration 84: $f(x) = -6.761101$
Iteration 85: New best $f(x) = -6.807427$
Iteration 86: New best $f(x) = -6.821693$
Iteration 87: New best $f(x) = -6.830020$
Iteration 88: New best $f(x) = -6.833581$
Iteration 89: New best $f(x) = -6.846502$
Iteration 90: New best $f(x) = -6.846678$
Iteration 91: New best $f(x) = -6.847082$
Iteration 92: $f(x) = -6.844715$
Iteration 93: New best $f(x) = -6.850282$
Iteration 94: New best $f(x) = -6.851126$
Iteration 95: $f(x) = -6.850650$

3.2. SpotOptim with Sklearn Kriging in 10 Dimensions: Michalewicz Function

Iteration 96: $f(x) = -6.849436$
Iteration 97: New best $f(x) = -6.857224$
Iteration 98: New best $f(x) = -6.858052$
Iteration 99: $f(x) = -6.856595$
Iteration 100: New best $f(x) = -6.865125$
Iteration 101: New best $f(x) = -6.865708$
Iteration 102: New best $f(x) = -6.866702$
Iteration 103: New best $f(x) = -6.870179$
Iteration 104: $f(x) = -6.832861$
Iteration 105: New best $f(x) = -6.870464$
Iteration 106: New best $f(x) = -6.870710$
Iteration 107: $f(x) = -6.870691$
Iteration 108: New best $f(x) = -6.870738$
Iteration 109: New best $f(x) = -6.871051$
Iteration 110: New best $f(x) = -6.872771$
Iteration 111: $f(x) = -6.872292$
Iteration 112: New best $f(x) = -6.873726$
Iteration 113: $f(x) = -6.873397$
Iteration 114: New best $f(x) = -6.874837$
Iteration 115: $f(x) = -6.874670$
Iteration 116: New best $f(x) = -6.875096$
Iteration 117: New best $f(x) = -6.875256$
Iteration 118: New best $f(x) = -6.875388$
Iteration 119: New best $f(x) = -6.875456$
Iteration 120: New best $f(x) = -6.876965$
Iteration 121: New best $f(x) = -6.877874$
Iteration 122: New best $f(x) = -6.878075$
Iteration 123: $f(x) = -6.878068$
Iteration 124: New best $f(x) = -6.878109$
Iteration 125: New best $f(x) = -6.878299$
Iteration 126: New best $f(x) = -6.879215$
Iteration 127: New best $f(x) = -6.879260$
Iteration 128: New best $f(x) = -6.880030$
Iteration 129: New best $f(x) = -6.880331$
Iteration 130: New best $f(x) = -6.880360$
Iteration 131: New best $f(x) = -6.880711$
Iteration 132: New best $f(x) = -6.881065$
Iteration 133: $f(x) = -6.881025$
Iteration 134: New best $f(x) = -6.881323$
Iteration 135: New best $f(x) = -6.882148$
Iteration 136: New best $f(x) = -6.883967$
Iteration 137: New best $f(x) = -6.884249$
Iteration 138: New best $f(x) = -6.884437$
Iteration 139: $f(x) = -6.884331$
Iteration 140: $f(x) = -6.852648$

3. Benchmarking SpotOptim with Sklearn Kriging (Matern Kernel) on 6D Rosenbrock and 10D Michalev

```
Iteration 141: New best f(x) = -6.885174
Iteration 142: New best f(x) = -6.885231
Iteration 143: f(x) = -6.885224
Iteration 144: New best f(x) = -6.885372
Iteration 145: New best f(x) = -6.885606
Iteration 146: f(x) = -6.885589
Iteration 147: New best f(x) = -6.885615
Iteration 148: f(x) = -6.885589
Iteration 149: f(x) = -6.885532
Iteration 150: New best f(x) = -6.885788
Iteration 151: f(x) = -6.885779
Iteration 152: New best f(x) = -6.885801
Iteration 153: New best f(x) = -6.885840
Iteration 154: New best f(x) = -6.885906
Iteration 155: New best f(x) = -6.885932
Iteration 156: New best f(x) = -6.885939
Iteration 157: New best f(x) = -6.885965
Iteration 158: New best f(x) = -6.886078
Iteration 159: f(x) = -6.886056
Iteration 160: New best f(x) = -6.886111
Iteration 161: New best f(x) = -6.886116
Iteration 162: New best f(x) = -6.886118
Iteration 163: f(x) = -6.886117
Iteration 164: f(x) = -6.886114
Iteration 165: New best f(x) = -6.886120
Iteration 166: New best f(x) = -6.886120
Iteration 167: New best f(x) = -6.886121
Iteration 168: f(x) = -6.886116
Iteration 169: f(x) = -6.886120
Iteration 170: f(x) = -6.886120
Iteration 171: f(x) = -6.886119
Iteration 172: New best f(x) = -6.886122
Iteration 173: f(x) = -6.886120
Iteration 174: New best f(x) = -6.886126
Iteration 175: New best f(x) = -6.886127
Iteration 176: f(x) = -6.844961
Iteration 177: New best f(x) = -6.886132
Iteration 178: New best f(x) = -6.886135
Iteration 179: New best f(x) = -6.886148
Iteration 180: New best f(x) = -6.886151
Iteration 181: New best f(x) = -6.886161
Iteration 182: New best f(x) = -6.886169
Iteration 183: New best f(x) = -6.886178
Iteration 184: f(x) = -6.886176
Iteration 185: f(x) = -6.886173
```


3.2. SpotOptim with Sklearn Kriging in 10 Dimensions: Michalewicz Function

Iteration 186: $f(x) = -6.886175$
Iteration 187: New best $f(x) = -6.886178$
Iteration 188: $f(x) = -6.886174$
Iteration 189: $f(x) = -6.886177$
Iteration 190: $f(x) = -6.886178$
Iteration 191: $f(x) = -6.886178$
Iteration 192: $f(x) = -6.886173$
Iteration 193: $f(x) = -6.886178$
Iteration 194: New best $f(x) = -6.886180$
Iteration 195: $f(x) = -6.886178$
Iteration 196: New best $f(x) = -6.886182$
Iteration 197: New best $f(x) = -6.886186$
Iteration 198: $f(x) = -6.886184$
Iteration 199: $f(x) = -6.886185$
Iteration 200: New best $f(x) = -6.886188$
Iteration 201: $f(x) = -6.886180$
Iteration 202: New best $f(x) = -6.886189$
Iteration 203: $f(x) = -6.886187$
Iteration 204: $f(x) = -6.886187$
Iteration 205: $f(x) = -1.911259$
Iteration 206: New best $f(x) = -6.886191$
Iteration 207: $f(x) = -6.886189$
Iteration 208: $f(x) = -6.886186$
Iteration 209: $f(x) = -6.886189$
Iteration 210: $f(x) = -6.886188$
Iteration 211: $f(x) = -6.886190$
Iteration 212: $f(x) = -6.886190$
Iteration 213: New best $f(x) = -6.886191$
Iteration 214: $f(x) = -6.886190$
Iteration 215: $f(x) = -6.886191$
Iteration 216: $f(x) = -6.886185$
Iteration 217: $f(x) = -6.886188$
Iteration 218: $f(x) = -6.886182$
Iteration 219: $f(x) = -6.886190$
Iteration 220: $f(x) = -6.886189$
Iteration 221: $f(x) = -6.886186$
Iteration 222: $f(x) = -6.886191$
Iteration 223: $f(x) = -6.886190$
Iteration 224: $f(x) = -6.886187$
Iteration 225: $f(x) = -6.886190$
Iteration 226: $f(x) = -6.886190$
Iteration 227: $f(x) = -6.886188$
Iteration 228: New best $f(x) = -6.886191$
Iteration 229: $f(x) = -6.886191$
Iteration 230: $f(x) = -6.886190$

3. Benchmarking SpotOptim with Sklearn Kriging (Matern Kernel) on 6D Rosenbrock and 10D Michalev

Iteration 231: $f(x) = -6.886190$
Iteration 232: New best $f(x) = -6.886192$
Iteration 233: New best $f(x) = -6.886192$
Iteration 234: $f(x) = -6.886191$
Iteration 235: New best $f(x) = -6.886193$
Iteration 236: $f(x) = -6.886192$
Iteration 237: $f(x) = -6.886191$
Iteration 238: $f(x) = -6.886189$
Iteration 239: $f(x) = -6.886191$
Iteration 240: $f(x) = -6.886184$
Iteration 241: $f(x) = -6.886193$
Iteration 242: $f(x) = -6.886191$
Iteration 243: $f(x) = -6.886188$
Iteration 244: $f(x) = -6.886190$
Iteration 245: $f(x) = -6.886192$
Iteration 246: $f(x) = -6.886193$
Iteration 247: $f(x) = -6.886191$
Iteration 248: $f(x) = -6.886192$
Iteration 249: $f(x) = -6.886191$
Iteration 250: $f(x) = -6.886190$
Iteration 251: $f(x) = -6.847637$
Iteration 252: $f(x) = -6.886192$
Iteration 253: $f(x) = -6.886192$
Iteration 254: $f(x) = -6.886192$
Iteration 255: $f(x) = -6.886187$
Iteration 256: $f(x) = -6.886187$
Iteration 257: $f(x) = -6.886189$
Iteration 258: $f(x) = -6.886191$
Iteration 259: $f(x) = -6.886193$
Iteration 260: $f(x) = -6.886178$
Iteration 261: $f(x) = -6.886192$
Iteration 262: $f(x) = -6.886177$
Iteration 263: $f(x) = -6.886190$
Iteration 264: $f(x) = -6.886193$
Iteration 265: $f(x) = -6.886192$
Iteration 266: $f(x) = -6.886193$
Iteration 267: $f(x) = -6.886192$
Iteration 268: $f(x) = -6.886192$
Iteration 269: $f(x) = -6.886191$
Iteration 270: $f(x) = -6.886190$
Iteration 271: $f(x) = -6.886192$
Iteration 272: $f(x) = -6.886190$
Iteration 273: $f(x) = -6.886192$
Iteration 274: $f(x) = -6.886191$
Iteration 275: $f(x) = -6.886188$

3.2. SpotOptim with Sklearn Kriging in 10 Dimensions: Michalewicz Function

```
Iteration 276: f(x) = -6.886184
Iteration 277: f(x) = -6.886192
Iteration 278: f(x) = -6.886176
Iteration 279: f(x) = -6.886187
Iteration 280: f(x) = -6.886191
Iteration 281: f(x) = -6.886186
Iteration 282: f(x) = -6.886182
Iteration 283: f(x) = -6.886190
Iteration 284: f(x) = -6.886192
Iteration 285: f(x) = -6.886193
Iteration 286: f(x) = -6.886192
Iteration 287: f(x) = -6.886192
Iteration 288: f(x) = -6.886191
Iteration 289: f(x) = -6.886193
Iteration 290: f(x) = -6.886193
```

```
print(f"[10D] Sklearn Kriging: min y = {result_micha.fun:.4f} at x = {result_micha.x}")
print(f"Number of function evaluations: {result_micha.nfev}")
print(f"Number of iterations: {result_micha.nit}")
```

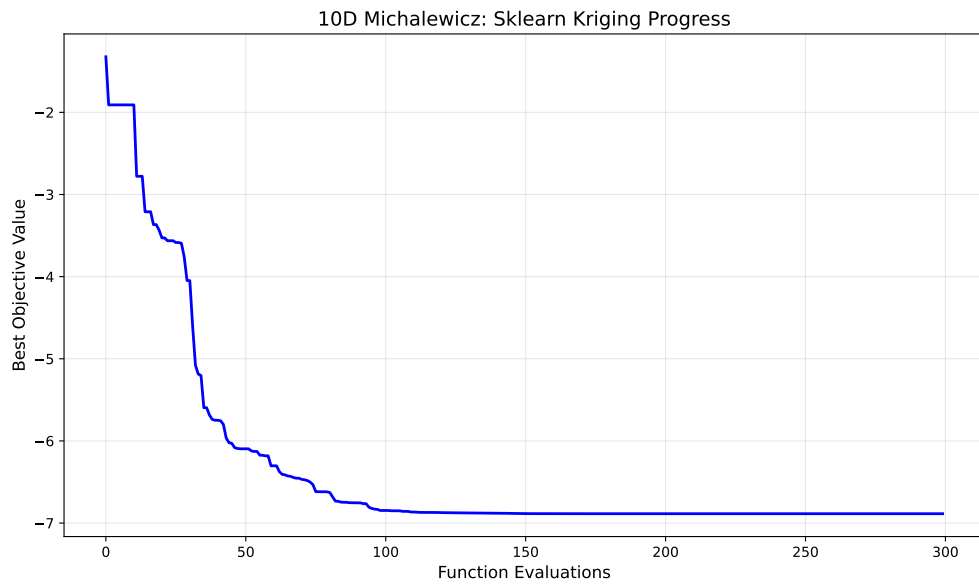
```
[10D] Sklearn Kriging: min y = -6.8862 at x = [2.20299373 2.71157921 2.21932073 2.48199421 2.6271922
2.00364851 1.36057903 1.28282431 1.21703873]
Number of function evaluations: 300
Number of iterations: 290
```

3.2.4. Visualize Optimization Progress

```
import matplotlib.pyplot as plt

# Plot the optimization progress
plt.figure(figsize=(10, 6))
plt.plot(np.minimum.accumulate(opt_micha.y_), 'b-', linewidth=2)
plt.xlabel('Function Evaluations', fontsize=12)
plt.ylabel('Best Objective Value', fontsize=12)
plt.title('10D Michalewicz: Sklearn Kriging Progress', fontsize=14)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

3. Benchmarking SpotOptim with Sklearn Kriging (Matern Kernel) on 6D Rosenbrock and 10D Michalewicz



3.2.5. Evaluation of Multiple Repeats

To perform 30 repeats and collect statistics:

```
# Perform 30 independent runs
n_repeats = 30
results = []

print(f"Running {n_repeats} independent optimizations...")
for i in range(n_repeats):
    kernel_i = ConstantKernel(1.0, (1e-2, 1e12)) * Matern(
        length_scale=1.0,
        length_scale_bounds=(1e-4, 1e2),
        nu=2.5
    )
    surrogate_i = GaussianProcessRegressor(kernel=kernel_i, n_restarts_optimizer=100)

    opt_i = SpotOptim(
        fun=fun,
        bounds=bounds,
        n_initial=n_initial,
        max_iter=max_iter,
        surrogate=surrogate_i,
        seed=seed + i, # Different seed for each run
```

```

        verbose=0
    )

    result_i = opt_i.optimize()
    results.append(result_i.fun)

    if (i + 1) % 10 == 0:
        print(f" Completed {i + 1}/{n_repeats} runs")

# Compute statistics
mean_result = np.mean(results)
std_result = np.std(results)
min_result = np.min(results)
max_result = np.max(results)

print(f"\nResults over {n_repeats} runs:")
print(f" Mean of best values: {mean_result:.6f}")
print(f" Std of best values: {std_result:.6f}")
print(f" Min of best values: {min_result:.6f}")
print(f" Max of best values: {max_result:.6f}")

```

3.3. Comparison: SpotOptim vs SpotPython

The **SpotOptim** package provides a scipy-compatible interface for Bayesian optimization with the following key features:

1. **Scipy-compatible API:** Returns `OptimizeResult` objects that work seamlessly with scipy's optimization ecosystem
2. **Custom Surrogates:** Supports any sklearn-compatible surrogate model (as demonstrated with `GaussianProcessRegressor`)
3. **Flexible Interface:** Simplified parameter specification with `bounds`, `n_initial`, and `max_iter`
4. **Analytical Test Functions:** Built-in test functions (`rosenbrock`, `ackley`, `michalewicz`) for benchmarking

The main differences from `spotpython` are:

- **SpotOptim:** Uses `bounds`, `n_initial`, `max_iter` parameters with scipy-style interface
- **SpotPython:** Uses `fun_control`, `design_control`, `surrogate_control` with more complex configuration

Both packages support custom surrogates and provide powerful Bayesian optimization capabilities.

3.4. Summary

This notebook demonstrated how to:

1. Use **SpotOptim** with sklearn's Gaussian Process Regressor (Matern kernel) as a surrogate
2. Optimize 6D Rosenbrock function with 100 evaluations
3. Optimize 10D Michalewicz function with 300 evaluations
4. Visualize optimization progress
5. Perform multiple independent runs for statistical analysis

The results show that **SpotOptim** with sklearn surrogates provides effective Bayesian optimization for challenging benchmark functions.

3.5. Jupyter Notebook

Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

4. Success Rate Tracking in SpotOptim

SpotOptim tracks the **success rate** of the optimization process, which measures how often the optimizer finds improvements over recent evaluations. This metric helps you understand whether the optimization is making progress or has stalled.

4.1. What is Success Rate?

The success rate is a **rolling metric** that tracks the percentage of recent evaluations that improved upon the best value found so far. It's calculated over a sliding window of the last 100 evaluations.

Key Points: - A “success” occurs when a new evaluation finds a value **better (smaller)** than the best found so far - The rate is computed over the last **100 evaluations** (window size) - Values range from **0.0** (no recent improvements) to **1.0** (all recent evaluations improved) - Helps identify when optimization is stalling and may need adjustment

4.2. First Example

- Start TensorBoard to visualize success rate in real-time:

```
tensorboard --logdir=runs
```

The execute the following code:

```
import numpy as np
from spotoptim import SpotOptim

def rosenbrock(X):
    """Rosenbrock function - challenging optimization problem"""
    x = X[:, 0]
    y = X[:, 1]
    return (1 - x)**2 + 100 * (y - x**2)**2
```

4. Success Rate Tracking in SpotOptim

```
# Run optimization with periodic success rate checks
optimizer = SpotOptim(
    fun=rosenbrock,
    bounds=[(-2, 2), (-2, 2)],
    max_iter=20,
    n_initial=10,
    tensorboard_log=True,
    tensorboard_clean=True,
    seed=42
)

result = optimizer.optimize()

# Analyze final success rate
print(f"\nOptimization Results:")
print(f"Best value: {result.fun:.6f}")
print(f"Total evaluations: {optimizer.counter}")
print(f"Final success rate: {optimizer.success_rate:.2%}")

# Interpret the result
if optimizer.success_rate > 0.5:
    print("→ High success rate: Optimization is still making good progress")
elif optimizer.success_rate > 0.2:
    print("→ Medium success rate: Approaching convergence")
else:
    print("→ Low success rate: Optimization has likely converged")
```

```
Optimization Results:
Best value: 0.011538
Total evaluations: 20
Final success rate: 30.00%
→ Medium success rate: Approaching convergence
```

4.3. Second Example

```
from spotoptim import SpotOptim
import numpy as np

def sphere(X):
    """Simple sphere function:  $f(x) = \sum(x^2)$ """
```



```

    return np.sum(X**2, axis=1)

# Create optimizer
optimizer = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5), (-5, 5)],
    max_iter=20,
    n_initial=10,
    verbose=True
)

# Run optimization
result = optimizer.optimize()

# Check success rate
print(f"Final success rate: {optimizer.success_rate:.2%}")
print(f"Total evaluations: {optimizer.counter}")

```

```

TensorBoard logging disabled
Initial best: f(x) = 11.891652
Iteration 1: f(x) = 12.848358
Iteration 2: New best f(x) = 9.938236
Iteration 3: New best f(x) = 8.241295
Iteration 4: f(x) = 9.008422
Iteration 5: New best f(x) = 2.891104
Iteration 6: New best f(x) = 0.293198
Iteration 7: New best f(x) = 0.010009
Iteration 8: New best f(x) = 0.000518
Iteration 9: New best f(x) = 0.000209
Iteration 10: f(x) = 0.000737
Final success rate: 70.00%
Total evaluations: 20

```

4.4. Accessing Success Rate

The success rate is stored in the `success_rate` attribute:

```

import numpy as np
from spotoptim import SpotOptim

def sphere(X):
    return np.sum(X**2, axis=1)

```

4. Success Rate Tracking in SpotOptim

```
optimizer = SpotOptim(fun=sphere, bounds=[(-5, 5), (-5, 5)], max_iter=20)
result = optimizer.optimize()

# Access success rate
current_rate = optimizer.success_rate
print(f"Success rate: {current_rate:.2%}")

# Also available via getter method
rate = optimizer._get_success_rate()
print(f"Via getter method: {rate:.2%}")
```

Success rate: 80.00%
Via getter method: 80.00%

4.5. Interpreting Success Rate

4.5.1. High Success Rate (> 0.5)

Success Rate: 75%

Interpretation: The optimizer is finding improvements frequently. This typically indicates: - The optimization is in an exploratory phase - The surrogate model is effectively guiding the search - There's still room for improvement in the search space

Action: Continue optimization - progress is good!

4.5.2. Medium Success Rate (0.2 - 0.5)

Success Rate: 35%

Interpretation: The optimizer occasionally finds improvements. This suggests: - The search is becoming more refined - The optimizer is balancing exploration and exploitation - Approaching a local or global optimum

Action: Monitor progress and consider stopping criteria.

4.5.3. Low Success Rate (< 0.2)

Success Rate: 8%

Interpretation: Few recent evaluations improve the best value. This may indicate: - The optimization has converged to a (local) optimum - The search is stuck in a plateau region - The budget may be exhausted in terms of meaningful progress

Action: Consider stopping optimization or adjusting parameters.

4.6. TensorBoard Visualization

When TensorBoard logging is enabled, success rate is automatically logged and can be visualized in real-time:

```
optimizer = SpotOptim(
    fun=objective,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=20,
    n_initial=10,
    tensorboard_log=True, # Enable logging
    verbose=True
)

result = optimizer.optimize()
```

View in TensorBoard:

```
tensorboard --logdir=runs
```

In the TensorBoard interface, look for: - **SCALARS** tab → **success_rate**: Rolling success rate over iterations - Compare multiple runs side-by-side - Identify when optimization stalls

4.7. Example: Comparing Multiple Runs

4. Success Rate Tracking in SpotOptim

```
import numpy as np
from spotoptim import SpotOptim

def ackley(X):
    """Ackley function - multimodal test function"""
    a = 20
    b = 0.2
    c = 2 * np.pi
    n = X.shape[1]

    sum_sq = np.sum(X**2, axis=1)
    sum_cos = np.sum(np.cos(c * X), axis=1)

    return -a * np.exp(-b * np.sqrt(sum_sq / n)) - np.exp(sum_cos / n) + a + np.e

# Run with different configurations
configs = [
    {"n_initial": 10, "max_iter": 30, "name": "Small initial"},
    {"n_initial": 20, "max_iter": 30, "name": "Large initial"},
]

results = []
for config in configs:
    optimizer = SpotOptim(
        fun=ackley,
        bounds=[(-5, 5), (-5, 5)],
        n_initial=config["n_initial"],
        max_iter=config["max_iter"],
        seed=42,
        verbose=False
    )
    result = optimizer.optimize()

    results.append({
        "name": config["name"],
        "best_value": result.fun,
        "success_rate": optimizer.success_rate,
        "n_evals": optimizer.counter
    })

print(f"\n{config['name']}:")
print(f"  Best value: {result.fun:.6f}")
print(f"  Success rate: {optimizer.success_rate:.2%}")
print(f"  Evaluations: {optimizer.counter}")
```

```
# Find best configuration
best = min(results, key=lambda x: x["best_value"])
print(f"\nBest configuration: {best['name']}")
print(f"  Achieved: f(x) = {best['best_value']:.6f}")
print(f"  Final success rate: {best['success_rate']:.2%}")
```

```
Small initial:
  Best value: 0.012066
  Success rate: 50.00%
  Evaluations: 30
```

```
Large initial:
  Best value: 0.006596
  Success rate: 30.00%
  Evaluations: 30
```

```
Best configuration: Large initial
  Achieved: f(x) = 0.006596
  Final success rate: 30.00%
```

4.8. Success Rate with Noisy Functions

For noisy functions (when `repeats_initial > 1` or `repeats_surrogate > 1`), the success rate tracks improvements in the **raw** y values, not the aggregated means:

```
import numpy as np
from spotoptim import SpotOptim

def noisy_sphere(X):
    """Sphere function with Gaussian noise"""
    base = np.sum(X**2, axis=1)
    noise = np.random.normal(0, 0.5, size=base.shape)
    return base + noise

optimizer = SpotOptim(
    fun=noisy_sphere,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=30,
    n_initial=10,
    repeats_initial=3,    # 3 evaluations per initial point
```

4. Success Rate Tracking in SpotOptim

```
repeats_surrogate=2, # 2 evaluations per new point
seed=42,
verbose=True
)

result = optimizer.optimize()

print(f"\nNoisy Optimization Results:")
print(f"Best raw value: {optimizer.min_y:.6f}")
print(f"Best mean value: {optimizer.min_mean_y:.6f}")
print(f"Success rate: {optimizer.success_rate:.2%}")
print(f"Total evaluations: {optimizer.counter}")
print(f"Unique design points: {optimizer.mean_X.shape[0]}")
```

TensorBoard logging disabled
Initial best: $f(x) = 0.936896$, mean best: $f(x) = 1.705270$

Noisy Optimization Results:
Best raw value: 0.936896
Best mean value: 1.705270
Success rate: 0.00%
Total evaluations: 30
Unique design points: 10

Note: With noisy functions, the success rate may be lower because: - Noise can mask true improvements - Multiple evaluations of the same point contribute to the window - Focus on the mean values (`min_mean_y`) for better assessment

4.9. Advanced: Custom Window Size

The success rate is calculated over a window of 100 evaluations by default. This is controlled by the `window_size` attribute:

```
import numpy as np
from spotoptim import SpotOptim

def sphere(X):
    return np.sum(X**2, axis=1)

optimizer = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
```

```

    max_iter=30,
    n_initial=10
)

# Check default window size
print(f"Window size: {optimizer.window_size}") # 100

# The window size is set during initialization
# To use a different window, you would need to modify it
# before running optimization (not typically recommended)

```

Window size: 100

4.10. Best Practices

4.10.1. 1. Monitor During Long Runs

For expensive optimization runs, periodically check success rate:

```

# Could be implemented with callbacks in future versions
# For now, success rate is updated automatically and logged to TensorBoard

```

4.10.2. 2. Combine with TensorBoard

Always enable TensorBoard logging for visual monitoring:

```

optimizer = SpotOptim(
    fun=expensive_function,
    bounds=bounds,
    max_iter=25,
    tensorboard_log=True, # Track success_rate visually
    tensorboard_path="runs/long_optimization"
)

```

4.10.3. 3. Use as Stopping Criterion

Consider stopping when success rate drops very low:

4. Success Rate Tracking in SpotOptim

```
# Manual stopping check (conceptual)
if optimizer.success_rate < 0.05 and optimizer.counter > 50:
    print("Success rate very low - optimization has likely converged")
```

4.10.4. 4. Compare Different Strategies

Use success rate to compare optimization strategies:

```
strategies = ["ei", "pi", "y"] # Different acquisition functions
for acq in strategies:
    opt = SpotOptim(fun=obj, bounds=bnds, acquisition=acq, max_iter=25)
    result = opt.optimize()
    print(f"{acq}: success_rate={opt.success_rate:.2%}, best={result.fun:.6f}")
```

4.11. Technical Details

4.11.1. How Success is Counted

A new evaluation `y_new` is considered a success if:

```
y_new < best_y_so_far
```

where `best_y_so_far` is the minimum value found in all previous evaluations.

4.11.2. Rolling Window Calculation

The success rate is computed as:

```
success_rate = (number of successes in last 100 evals) / (window size)
```

- Window size defaults to 100
- If fewer than 100 evaluations have been performed, the window size is the number of evaluations
- The window slides forward with each new evaluation

4.11.3. Update Frequency

The success rate is updated after: 1. Initial design evaluation 2. Each iteration's new point evaluation(s) 3. OCBA re-evaluations (if applicable)

4.12. Summary

- **Success rate** measures the percentage of recent evaluations that improve the best value
- Calculated over a rolling window of the last **100 evaluations**
- Values range from **0.0** to **1.0**
- High rates (>0.5) indicate active progress
- Low rates (<0.2) suggest convergence
- Automatically logged to **TensorBoard** when logging is enabled
- Available via `optimizer.success_rate` attribute after optimization

Use success rate to: - Monitor optimization progress in real-time - Identify when to stop optimization - Compare different optimization strategies - Assess optimization difficulty for different problems

4.13. Jupyter Notebook

Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

Part II.

Variables and Hyperparameters

5. Variable Type (`var_type`) Implementation in SpotOptim

5.1. Overview

This document describes the `var_type` implementation in SpotOptim, which allows users to specify different data types for optimization variables.

5.2. Supported Variable Types

SpotOptim supports three main data types:

5.2.1. 1. 'float'

- **Purpose:** Continuous optimization with Python floats
- **Behavior:** No rounding applied, values remain continuous
- **Use case:** Standard continuous optimization variables
- **Example:** Temperature (23.5°C), Distance (1.234m)

5.2.2. 2. 'int'

- **Purpose:** Discrete integer optimization
- **Behavior:** Float values are automatically rounded to integers
- **Use case:** Count variables, discrete parameters
- **Example:** Number of layers (5), Population size (100)

5.2.3. 3. 'factor'

- **Purpose:** Unordered categorical data
- **Behavior:** Internally mapped to integer values (0, 1, 2, ...)
- **Use case:** Categorical choices like colors, algorithms, modes
- **Example:** Color ("red"→0, "green"→1, "blue"→2)

5. Variable Type (`var_type`) Implementation in *SpotOptim*

- **Note:** The actual string-to-int mapping is external to *SpotOptim*; the optimizer works with the integer representation

5.3. Implementation Details

5.3.1. Where `var_type` is Used

The `var_type` parameter is properly propagated throughout the optimization process:

1. **Initialization** (`__init__`):
 - Stored as `self.var_type`
 - Default: `["float"] * n_dim` if not specified
2. **Initial Design Generation** (`_generate_initial_design`):
 - Applies type constraints via `_repair_non_numeric()`
 - Ensures initial points respect variable types
3. **New Point Suggestion** (`_suggest_next_point`):
 - Applies type constraints to acquisition function optimization results
 - Ensures suggested points respect variable types
4. **User-Provided Initial Design** (`optimize`):
 - Applies type constraints to `X0` if provided
 - Ensures consistency regardless of input source
5. **Mesh Grid Generation** (`_generate_mesh_grid`):
 - Used for plotting, respects variable types
 - Ensures visualization shows correct discrete/continuous behavior

5.3.2. Core Method: `_repair_non_numeric()`

This method enforces variable type constraints:

```
def _repair_non_numeric(self, X: np.ndarray, var_type: List[str]) -> np.ndarray:
    """Round non-continuous values to integers."""
    mask = np.isin(var_type, ["float"], invert=True)
    X[:, mask] = np.around(X[:, mask])
    return X
```

Logic:

- Variables with type 'float': No change (continuous)
- Variables with type 'int' or 'factor': Rounded to integers

5.4. Example Usage

5.4.1. Example 1: All Float Variables (Default)

```
import numpy as np
from spotoptim import SpotOptim

# Example 1: All float variables (default)
opt1 = SpotOptim(
    fun=lambda X: np.sum(X**2, axis=1),
    bounds=[(0, 10), (0, 10), (0, 10)],
    max_iter=20,
    n_initial=10,
    seed=42
    # var_type defaults to ["float", "float", "float"]
)
result1 = opt1.optimize()
print(f"Best value: {result1.fun:.6f}")
print(f"Best point (floats): {result1.x}")
```

```
Best value: 0.000000
Best point (floats): [0. 0. 0.]
```

5.4.2. Example 2: Pure Integer Optimization

```
import numpy as np
from spotoptim import SpotOptim

def discrete_func(X):
    return np.sum(X**2, axis=1)

bounds = [(-5, 5), (-5, 5)]
var_type = ["int", "int"]

opt = SpotOptim(
    fun=discrete_func,
```

5. Variable Type (*var_type*) Implementation in *SpotOptim*

```
        bounds=bounds,
        var_type=var_type,
        max_iter=20,
        n_initial=10,
        seed=42
    )

    result = opt.optimize()
    print(f"Best value: {result.fun:.6f}")
    print(f"Best point (integers): {result.x}")
    print(f"Note: Values are rounded to integers")
```

```
Best value: 0.000000
Best point (integers): [ 0. -0.]
Note: Values are rounded to integers
```

5.4.3. Example 3: Categorical (Factor) Variables

```
import numpy as np
from spotoptim import SpotOptim

def categorical_func(X):
    # Assume X[:, 0] represents 3 categories: 0, 1, 2
    # Category 0 is best
    return (X[:, 0]**2) + (X[:, 1]**2)

bounds = [(0, 2), (0, 3)] # 3 and 4 categories respectively
var_type = ["factor", "factor"]

opt = SpotOptim(
    fun=categorical_func,
    bounds=bounds,
    var_type=var_type,
    max_iter=20,
    n_initial=10,
    seed=42
)

result = opt.optimize()
print(f"Best value: {result.fun:.6f}")
print(f"Best point (categories): {result.x}")
print(f"Note: Values are integers representing categories")
```


Best value: 0.000000
 Best point (categories): [0. 0.]
 Note: Values are integers representing categories

5.4.4. Example 4: Mixed Variable Types

```
import numpy as np
from spotoptim import SpotOptim

def mixed_func(X):
    # X[:, 0]: continuous temperature
    # X[:, 1]: discrete number of iterations
    # X[:, 2]: categorical algorithm choice (0, 1, 2)
    return X[:, 0]**2 + X[:, 1]**2 + X[:, 2]**2

bounds = [(-5, 5), (1, 100), (0, 2)]
var_type = ["float", "int", "factor"]
var_name = ["temperature", "iterations", "algorithm"]

opt = SpotOptim(
    fun=mixed_func,
    bounds=bounds,
    var_type=var_type,
    var_name=var_name,
    max_iter=20,
    n_initial=10,
    seed=42
)

result = opt.optimize()
print(f"Best value: {result.fun:.6f}")
print(f"Best point: {result.x}")
print(f"  {var_name[0]} (float): {result.x[0]:.6f}")
print(f"  {var_name[1]} (int): {int(result.x[1])}")
print(f"  {var_name[2]} (factor): {int(result.x[2])}")
```

Best value: 1.000010
 Best point: [-0.00309017 1. 0.]
 temperature (float): -0.003090
 iterations (int): 1
 algorithm (factor): 0

5.5. Key Findings

1. **Type Persistence:** Variable types are correctly maintained throughout the entire optimization process, from initial design through all iterations.
2. **Automatic Enforcement:** The `_repair_non_numeric()` method is called at all critical points, ensuring type constraints are never violated.
3. **Three Explicit Types:** Only `'float'`, `'int'`, and `'factor'` are supported. The legacy `'num'` type has been removed for clarity.
4. **User-Provided Data:** Type constraints are applied even to user-provided initial designs, ensuring consistency.
5. **Plotting Compatibility:** The plotting functionality respects variable types, ensuring correct visualization of discrete vs. continuous variables.

5.6. Recommendations

1. **Always specify `var_type` explicitly** for clarity, especially in mixed-type problems
2. **Use appropriate bounds** for factor variables (e.g., `(0, n_categories-1)`)
3. **External mapping** for string categories: Maintain your own mapping dictionary outside *SpotOptim* (e.g., `{"red": 0, "green": 1, "blue": 2}`)
4. **Validation:** The current implementation doesn't validate `var_type` length matches bounds length - users should ensure this manually

5.7. Future Enhancements (Optional)

Potential improvements that could be added:

1. **Validation:** Add validation in `__init__` to check `len(var_type) == len(bounds)`
2. **String Categories:** Add built-in support for automatic string-to-int mapping
3. **Ordered Categories:** Support ordered categorical variables (ordinal data)
4. **Type Checking:** Validate that `var_type` values are one of the allowed strings
5. **Bounds Checking:** Warn if factor bounds are not integer ranges

5.8. Jupyter Notebook

i Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

6. Factor Variables for Categorical Hyperparameters

SpotOptim supports factor variables for optimizing categorical hyperparameters, such as activation functions, optimizers, or any discrete string-based choices. Factor variables are automatically converted between string values (external interface) and integers (internal optimization), making categorical optimization seamless.

6.1. Overview

What are Factor Variables?

Factor variables allow you to specify categorical choices as tuples of strings in the bounds. SpotOptim handles the conversion:

1. **String tuples in bounds** → Internal integer mapping (0, 1, 2, ...)
2. **Optimization uses integers** internally for surrogate modeling
3. **Objective function receives strings** after automatic conversion
4. **Results return strings** (not integers)

Module: `spotoptim.SpotOptim`

Key Features:

- Define categorical choices as string tuples: ("ReLU", "Sigmoid", "Tanh")
- Automatic integer string conversion
- Seamless integration with neural network hyperparameters
- Mix factor variables with numeric/integer variables

6.2. Quick Start

6.2.1. Basic Factor Variable Usage

6. Factor Variables for Categorical Hyperparameters

```
from spotoptim import SpotOptim
import numpy as np

def objective_function(X):
    """Objective function receives string values."""
    results = []
    for params in X:
        activation = params[0] # This is a string!
        print(f"Testing activation: {activation}")

        # Simple scoring based on activation choice (for demonstration)
        # In real use, you would train a model and return actual performance
        scores = {
            "ReLU": 3500.0,
            "Sigmoid": 4200.0,
            "Tanh": 3800.0,
            "LeakyReLU": 3600.0
        }
        score = scores.get(activation, 5000.0) + np.random.normal(0, 100)
        results.append(score)
    return np.array(results) # Return numpy array

# Define bounds with factor variable
optimizer = SpotOptim(
    fun=objective_function,
    bounds=[("ReLU", "Sigmoid", "Tanh", "LeakyReLU")],
    var_type=["factor"],
    max_iter=20,
    seed=42
)

result = optimizer.optimize()
print(f"\nBest activation: {result.x[0]}") # Returns string, e.g., "ReLU"
print(f"Best score: {result.fun:.4f}")
```

```
Testing activation: ReLU
Testing activation: Sigmoid
Testing activation: Tanh
Testing activation: LeakyReLU
Testing activation: Tanh
Testing activation: Sigmoid
Testing activation: Sigmoid
Testing activation: Sigmoid
Testing activation: Sigmoid
```

```

Testing activation: ReLU
Testing activation: Sigmoid
Testing activation: Tanh
Testing activation: LeakyReLU
Testing activation: Sigmoid
Testing activation: LeakyReLU
Testing activation: Tanh
Testing activation: Sigmoid
Testing activation: LeakyReLU
Testing activation: Sigmoid
Testing activation: Tanh

```

```

Best activation: LeakyReLU
Best score: 3505.9864

```

6.2.2. Neural Network Activation Function Optimization

```

import torch
import torch.nn as nn
from spotoptim import SpotOptim
from spotoptim.data import get_diabetes_dataloaders
from spotoptim.nn.linear_regressor import LinearRegressor
import numpy as np

def train_and_evaluate(X):
    """Train models with different activation functions."""
    results = []

    for params in X:
        activation = params[0] # String: "ReLU", "Sigmoid", etc.

        # Load data
        train_loader, test_loader, _ = get_diabetes_dataloaders()

        # Create model with the activation function
        model = LinearRegressor(
            input_dim=10,
            output_dim=1,
            l1=64,
            num_hidden_layers=2,
            activation=activation # Pass string directly!
        )

```

6. Factor Variables for Categorical Hyperparameters

```
# Train model
optimizer = model.get_optimizer("Adam", lr=0.01)
criterion = nn.MSELoss()

for epoch in range(50):
    model.train()
    for batch_X, batch_y in train_loader:
        predictions = model(batch_X)
        loss = criterion(predictions, batch_y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

# Evaluate
model.eval()
test_loss = 0.0
with torch.no_grad():
    for batch_X, batch_y in test_loader:
        predictions = model(batch_X)
        test_loss += criterion(predictions, batch_y).item()

avg_loss = test_loss / len(test_loader)
results.append(avg_loss)

return np.array(results) # Return numpy array

# Optimize activation function choice
optimizer = SpotOptim(
    fun=train_and_evaluate,
    bounds=[("ReLU", "Sigmoid", "Tanh", "LeakyReLU", "ELU")],
    var_type=["factor"],
    max_iter=30
)

result = optimizer.optimize()
print(f"Best activation function: {result.x[0]}")
print(f"Best test MSE: {result.fun:.4f}")
```

```
Best activation function: Sigmoid
Best test MSE: 26406.7357
```


6.3. Mixed Variable Types

6.3.1. Combining Factor, Integer, and Continuous Variables

```
import numpy as np
import torch
import torch.nn as nn
from spotoptim import SpotOptim
from spotoptim.data import get_diabetes_dataloaders
from spotoptim.nn.linear_regressor import LinearRegressor

def comprehensive_optimization(X):
    """Optimize learning rate, layer size, depth, and activation."""
    results = []

    for params in X:
        log_lr = params[0]      # Continuous (log scale)
        l1 = int(params[1])     # Integer
        n_layers = int(params[2]) # Integer
        activation = params[3]  # Factor (string)

        lr = 10 ** log_lr      # Convert from log scale

        print(f"lr={lr:.6f}, l1={l1}, layers={n_layers}, activation={activation}")

        # Load data
        train_loader, test_loader, _ = get_diabetes_dataloaders(
            batch_size=32,
            random_state=42
        )

        # Create model
        model = LinearRegressor(
            input_dim=10,
            output_dim=1,
            l1=l1,
            num_hidden_layers=n_layers,
            activation=activation
        )

        # Train
        optimizer = model.get_optimizer("Adam", lr=lr)
        criterion = nn.MSELoss()
```

6. Factor Variables for Categorical Hyperparameters

```
for epoch in range(30):
    model.train()
    for batch_X, batch_y in train_loader:
        predictions = model(batch_X)
        loss = criterion(predictions, batch_y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    # Evaluate
    model.eval()
    test_loss = 0.0
    with torch.no_grad():
        for batch_X, batch_y in test_loader:
            predictions = model(batch_X)
            test_loss += criterion(predictions, batch_y).item()

    results.append(test_loss / len(test_loader))

return np.array(results)

# Optimize all four hyperparameters simultaneously
optimizer = SpotOptim(
    fun=comprehensive_optimization,
    bounds=[
        (-4, -2), # log10(learning_rate)
        (16, 128), # l1 (neurons per layer)
        (0, 4), # num_hidden_layers
        ("ReLU", "Sigmoid", "Tanh", "LeakyReLU") # activation function
    ],
    var_type=["float", "int", "int", "factor"],
    max_iter=50
)

result = optimizer.optimize()

# Results contain original string values
print("\nOptimization Results:")
print(f"Best learning rate: {10**result.x[0]:.6f}")
print(f"Best layer size: {int(result.x[1])}")
print(f"Best num layers: {int(result.x[2])}")
print(f"Best activation: {result.x[3]}") # String value!
print(f"Best test MSE: {result.fun:.4f}")
```

6.3. Mixed Variable Types

lr=0.000715, l1=94, layers=4, activation=Tanh
lr=0.001695, l1=28, layers=1, activation=ReLU
lr=0.003198, l1=21, layers=2, activation=Sigmoid
lr=0.005605, l1=54, layers=0, activation=Sigmoid
lr=0.000438, l1=101, layers=3, activation=Tanh
lr=0.000188, l1=69, layers=1, activation=Sigmoid
lr=0.000137, l1=113, layers=3, activation=Sigmoid
lr=0.000279, l1=121, layers=1, activation=Tanh
lr=0.008389, l1=46, layers=2, activation=LeakyReLU
lr=0.001258, l1=73, layers=3, activation=LeakyReLU
lr=0.005513, l1=54, layers=0, activation=Sigmoid
lr=0.006387, l1=22, layers=3, activation=Sigmoid
lr=0.000910, l1=49, layers=0, activation=Tanh
lr=0.002885, l1=47, layers=3, activation=Sigmoid
lr=0.000625, l1=63, layers=4, activation=Sigmoid
lr=0.000687, l1=111, layers=1, activation=Sigmoid
lr=0.005881, l1=28, layers=2, activation=Tanh
lr=0.000239, l1=48, layers=2, activation=Tanh
lr=0.000106, l1=67, layers=0, activation=Sigmoid
lr=0.007171, l1=113, layers=2, activation=Tanh
lr=0.001921, l1=19, layers=2, activation=Sigmoid
lr=0.000104, l1=62, layers=0, activation=Sigmoid
lr=0.002751, l1=62, layers=3, activation=Tanh
lr=0.000145, l1=56, layers=2, activation=Tanh
lr=0.000944, l1=94, layers=4, activation=Tanh
lr=0.001615, l1=21, layers=3, activation=LeakyReLU
lr=0.002402, l1=99, layers=1, activation=Sigmoid
lr=0.000721, l1=88, layers=3, activation=Tanh
lr=0.000136, l1=125, layers=1, activation=Sigmoid
lr=0.000402, l1=33, layers=2, activation=ReLU
lr=0.002675, l1=83, layers=0, activation=Tanh
lr=0.001702, l1=43, layers=2, activation=ReLU
lr=0.005154, l1=116, layers=4, activation=Sigmoid
lr=0.002329, l1=78, layers=1, activation=ReLU
lr=0.002244, l1=26, layers=1, activation=Tanh
lr=0.004362, l1=109, layers=3, activation=Sigmoid
lr=0.001336, l1=75, layers=2, activation=Sigmoid
lr=0.000206, l1=32, layers=3, activation=Sigmoid
lr=0.000158, l1=50, layers=4, activation=Sigmoid
lr=0.000107, l1=41, layers=1, activation=ReLU
lr=0.008772, l1=102, layers=4, activation=ReLU
lr=0.001879, l1=122, layers=1, activation=Sigmoid
lr=0.001048, l1=27, layers=2, activation=Sigmoid
lr=0.002118, l1=96, layers=2, activation=ReLU
lr=0.000155, l1=112, layers=1, activation=LeakyReLU

6. Factor Variables for Categorical Hyperparameters

```
lr=0.005462, l1=67, layers=3, activation=LeakyReLU
lr=0.000266, l1=55, layers=2, activation=Tanh
lr=0.007063, l1=84, layers=1, activation=LeakyReLU
lr=0.000301, l1=48, layers=2, activation=Tanh
lr=0.003782, l1=32, layers=1, activation=ReLU
```

Optimization Results:

Best learning rate: 0.000687

Best layer size: 111

Best num layers: 1

Best activation: Sigmoid

Best test MSE: 26454.2025

6.4. Multiple Factor Variables

6.4.1. Optimizing Both Activation and Optimizer

```
from spotoptim import SpotOptim
from spotoptim.data import get_diabetes_data loaders
from spotoptim.nn.linear_regressor import LinearRegressor
import torch.nn as nn
import numpy as np

def optimize_activation_and_optimizer(X):
    """Optimize both activation function and optimizer choice."""
    results = []

    for params in X:
        activation = params[0]      # Factor variable 1
        optimizer_name = params[1]  # Factor variable 2
        lr = 10 ** params[2]        # Continuous variable

        train_loader, test_loader, _ = get_diabetes_data loaders()

        model = LinearRegressor(
            input_dim=10,
            output_dim=1,
            l1=64,
            num_hidden_layers=2,
            activation=activation
        )
```

```

# Use the optimizer string
optimizer = model.get_optimizer(optimizer_name, lr=lr)
criterion = nn.MSELoss()

# Train
for epoch in range(30):
    model.train()
    for batch_X, batch_y in train_loader:
        predictions = model(batch_X)
        loss = criterion(predictions, batch_y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

# Evaluate
model.eval()
test_loss = 0.0
with torch.no_grad():
    for batch_X, batch_y in test_loader:
        predictions = model(batch_X)
        test_loss += criterion(predictions, batch_y).item()

results.append(test_loss / len(test_loader))

return np.array(results) # Return numpy array

# Two factor variables + one continuous
opt = SpotOptim(
    fun=optimize_activation_and_optimizer,
    bounds=[
        ("ReLU", "Tanh", "Sigmoid", "LeakyReLU"), # Activation
        ("Adam", "SGD", "RMSprop", "AdamW"), # Optimizer
        (-4, -2) # log10(lr)
    ],
    var_type=["factor", "factor", "float"],
    max_iter=40
)

result = opt.optimize()
print(f"Best activation: {result.x[0]}")
print(f"Best optimizer: {result.x[1]}")
print(f"Best learning rate: {10**result.x[2]:.6f}")

```

Best activation: LeakyReLU

6. Factor Variables for Categorical Hyperparameters

Best optimizer: SGD
Best learning rate: 0.009243

6.5. Advanced Usage

6.5.1. Custom Categorical Choices

Factor variables work with any string values, not just activation functions:

```
from spotoptim import SpotOptim
import numpy as np

def train_model_with_config(dropout_policy, batch_norm, weight_init):
    """Simulate model training with different configurations."""
    # In real use, this would train an actual model
    # Here we return synthetic scores for demonstration
    base_score = 3000.0

    # Dropout impact
    dropout_scores = {"none": 200, "light": 0, "heavy": 100}
    # Batch norm impact
    bn_scores = {"before": -50, "after": 0, "none": 150}
    # Weight init impact
    init_scores = {"xavier": 0, "kaiming": -30, "normal": 100}

    score = (base_score +
             dropout_scores.get(dropout_policy, 0) +
             bn_scores.get(batch_norm, 0) +
             init_scores.get(weight_init, 0) +
             np.random.normal(0, 50))

    return score

def train_with_config(X):
    """Objective function with various categorical choices."""
    results = []

    for params in X:
        dropout_policy = params[0] # "none", "light", "heavy"
        batch_norm = params[1]     # "before", "after", "none"
        weight_init = params[2]    # "xavier", "kaiming", "normal"

        # Use these strings to configure your model
```

```

        score = train_model_with_config(
            dropout_policy=dropout_policy,
            batch_norm=batch_norm,
            weight_init=weight_init
        )
        results.append(score)

    return np.array(results) # Return numpy array

optimizer = SpotOptim(
    fun=train_with_config,
    bounds=[
        ("none", "light", "heavy"),          # Dropout policy
        ("before", "after", "none"),          # Batch norm position
        ("xavier", "kaiming", "normal")       # Weight initialization
    ],
    var_type=["factor", "factor", "factor"],
    max_iter=25,
    seed=42
)

result = optimizer.optimize()
print("Best configuration:")
print(f" Dropout: {result.x[0]}")
print(f" Batch norm: {result.x[1]}")
print(f" Weight init: {result.x[2]}")
print(f" Score: {result.fun:.4f}")

```

```

Best configuration:
Dropout: light
Batch norm: before
Weight init: kaiming
Score: 2902.0093

```

6.5.2. Viewing All Evaluated Configurations

```

import torch
import torch.nn as nn
from spotoptim import SpotOptim
from spotoptim.data import get_diabetes_dataloaders
from spotoptim.nn.linear_regressor import LinearRegressor
import numpy as np

```

6. Factor Variables for Categorical Hyperparameters

```
def train_and_evaluate(X):
    """Train models with different activation functions."""
    results = []

    for params in X:
        l1 = int(params[0])          # Integer: layer size
        activation = params[1]       # String: activation function

        # Load data
        train_loader, test_loader, _ = get_diabetes_dataloaders()

        # Create model with the activation function
        model = LinearRegressor(
            input_dim=10,
            output_dim=1,
            l1=l1,
            num_hidden_layers=2,
            activation=activation    # Pass string directly!
        )

        # Train model
        optimizer = model.get_optimizer("Adam", lr=0.01)
        criterion = nn.MSELoss()

        for epoch in range(50):
            model.train()
            for batch_X, batch_y in train_loader:
                predictions = model(batch_X)
                loss = criterion(predictions, batch_y)
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

            # Evaluate
            model.eval()
            test_loss = 0.0
            with torch.no_grad():
                for batch_X, batch_y in test_loader:
                    predictions = model(batch_X)
                    test_loss += criterion(predictions, batch_y).item()

            avg_loss = test_loss / len(test_loader)
            results.append(avg_loss)
```



```

    return np.array(results)

optimizer = SpotOptim(
    fun=train_and_evaluate,
    bounds=[
        (16, 128),                # Layer size
        ("ReLU", "Sigmoid", "Tanh", "LeakyReLU") # Activation
    ],
    var_type=["int", "factor"], # IMPORTANT: Specify variable types!
    max_iter=30,
    seed=42
)

result = optimizer.optimize()

# Access all evaluated configurations
print("\nAll evaluated configurations:")
print("Layer Size | Activation | Test MSE")
print("-" * 42)
for i in range(min(10, len(result.X))): # Show first 10
    l1 = int(result.X[i, 0])
    activation = result.X[i, 1] # String value!
    loss = result.y[i]
    print(f"{l1:10d} | {activation:10s} | {loss:.4f}")

# Find top 5 configurations
sorted_indices = result.y.argsort()[:5]
print("\nTop 5 configurations:")
for idx in sorted_indices:
    print(f"l1={int(result.X[idx, 0]):3d}, "
          f"activation={result.X[idx, 1]:10s}, "
          f"MSE={result.y[idx]:.4f}")

```

All evaluated configurations:

Layer Size | Activation | Test MSE

```

-----
    41 | Tanh      | 26583.5052
   118 | Sigmoid    | 26486.3841
    26 | Tanh      | 26634.2480
   108 | Sigmoid    | 26399.6211
    71 | LeakyReLU  | 26515.7402
    34 | Tanh      | 26701.5632
    87 | ReLU      | 26496.3730

```

6. Factor Variables for Categorical Hyperparameters

101	Tanh	26585.3066
55	Sigmoid	26547.7930
74	ReLU	26610.8079

Top 5 configurations:

```
l1=109, activation=Sigmoid , MSE=26242.3887
l1=127, activation=Sigmoid , MSE=26382.2493
l1=108, activation=Sigmoid , MSE=26399.6211
l1= 54, activation=Sigmoid , MSE=26417.8555
l1=107, activation=LeakyReLU , MSE=26437.5527
```

6.6. How It Works

6.6.1. Internal Mechanism

SpotOptim handles factor variables through automatic conversion:

1. **Initialization:** String tuples in bounds are detected

```
bounds = [("ReLU", "Sigmoid", "Tanh")]
# Internally mapped to: {0: "ReLU", 1: "Sigmoid", 2: "Tanh"}
# Bounds become: [(0, 2)]
```

2. **Sampling:** Initial design samples from `[0, n_levels-1]` and rounds to integers

```
# Samples might be: [0.3, 1.8, 2.1]
# After rounding: [0, 2, 2]
```

3. **Evaluation:** Before calling objective function, integers $\rightarrow$ strings

```
# [0, 2, 2]  $\rightarrow$  ["ReLU", "Tanh", "Tanh"]
# Objective function receives strings
```

4. **Optimization:** Surrogate model works with integers `[0, n_levels-1]`

5. **Results:** Final results mapped back to strings

```
result.x[0] # Returns "ReLU", not 0
result.X    # All rows contain strings for factor variables
```

```
array([[41.0, 'Tanh'],
       [118.0, 'Sigmoid'],
       [26.0, 'Tanh'],
       [108.0, 'Sigmoid'],
       [71.0, 'LeakyReLU'],
       [34.0, 'Tanh'],
```

```
[87.0, 'ReLU'],
[101.0, 'Tanh'],
[55.0, 'Sigmoid'],
[74.0, 'ReLU'],
[89.0, 'Sigmoid'],
[109.0, 'Sigmoid'],
[20.0, 'Tanh'],
[87.0, 'Tanh'],
[107.0, 'LeakyReLU'],
[75.0, 'Tanh'],
[53.0, 'Tanh'],
[35.0, 'Sigmoid'],
[93.0, 'Sigmoid'],
[38.0, 'Tanh'],
[96.0, 'Sigmoid'],
[112.0, 'Tanh'],
[127.0, 'Sigmoid'],
[54.0, 'Sigmoid'],
[64.0, 'LeakyReLU'],
[115.0, 'Sigmoid'],
[75.0, 'Sigmoid'],
[42.0, 'Sigmoid'],
[66.0, 'Sigmoid'],
[94.0, 'LeakyReLU']], dtype=object)
```

6.6.2. Variable Type Auto-Detection

If you don't specify `var_type`, SpotOptim automatically detects factor variables:

```
# Example 1: Explicit var_type (recommended)
# This shows the syntax - replace my_function with your actual function

# optimizer = SpotOptim(
#     fun=my_function,
#     bounds=[(-4, -2), ("ReLU", "Tanh")],
#     var_type=["float", "factor"] # Explicit
# )

# Example 2: Auto-detection (works but less explicit)
# optimizer = SpotOptim(
#     fun=my_function,
#     bounds=[(-4, -2), ("ReLU", "Tanh")]
#     # var_type automatically set to ["float", "factor"]
# )
```

6. Factor Variables for Categorical Hyperparameters

```
# Here's a working example:
from spotoptim import SpotOptim
import numpy as np

def demo_function(X):
    results = []
    for params in X:
        lr = 10 ** params[0] # Continuous parameter
        activation = params[1] # Factor parameter
        score = 3000 + lr * 100 + {"ReLU": 0, "Tanh": 50}.get(activation, 100)
        results.append(score + np.random.normal(0, 10))
    return np.array(results)

# With explicit var_type (recommended)
optimizer = SpotOptim(
    fun=demo_function,
    bounds=[(-4, -2), ("ReLU", "Tanh")],
    var_type=["float", "factor"], # Explicit is clearer
    max_iter=10,
    seed=42
)

result = optimizer.optimize()
print(f"Best lr: {10**result.x[0]:.6f}, Best activation: {result.x[1]}")
```

Best lr: 0.006734, Best activation: ReLU

6.7. Complete Example: Full Workflow

```
"""
Complete example: Neural network hyperparameter optimization with factor variables.
"""

import numpy as np
import torch
import torch.nn as nn
from spotoptim import SpotOptim
from spotoptim.data import get_diabetes_dataloaders
from spotoptim.nn.linear_regressor import LinearRegressor
```

6.7. Complete Example: Full Workflow

```
def objective_function(X):
    """Train and evaluate models with given hyperparameters."""
    results = []

    for params in X:
        # Extract hyperparameters
        log_lr = params[0]
        l1 = int(params[1])
        num_layers = int(params[2])
        activation = params[3] # String!

        lr = 10 ** log_lr

        print(f"Testing: lr={lr:.6f}, l1={l1}, layers={num_layers}, "
              f"activation={activation}")

        # Load data
        train_loader, test_loader, _ = get_diabetes_data loaders(
            test_size=0.2,
            batch_size=32,
            random_state=42
        )

        # Create and train model
        model = LinearRegressor(
            input_dim=10,
            output_dim=1,
            l1=l1,
            num_hidden_layers=num_layers,
            activation=activation
        )

        optimizer = model.get_optimizer("Adam", lr=lr)
        criterion = nn.MSELoss()

        # Training loop
        num_epochs = 30
        for epoch in range(num_epochs):
            model.train()
            for batch_X, batch_y in train_loader:
                predictions = model(batch_X)
                loss = criterion(predictions, batch_y)
                optimizer.zero_grad()
                loss.backward()
```

6. Factor Variables for Categorical Hyperparameters

```
        optimizer.step()

    # Evaluation
    model.eval()
    test_loss = 0.0
    with torch.no_grad():
        for batch_X, batch_y in test_loader:
            predictions = model(batch_X)
            loss = criterion(predictions, batch_y)
            test_loss += loss.item()

    avg_test_loss = test_loss / len(test_loader)
    results.append(avg_test_loss)
    print(f" → Test MSE: {avg_test_loss:.4f}")

return np.array(results)

def main():
    print("=" * 80)
    print("Neural Network Hyperparameter Optimization with Factor Variables")
    print("=" * 80)

    # Define optimization problem
    optimizer = SpotOptim(
        fun=objective_function,
        bounds=[
            (-4, -2), # log10(learning_rate)
            (16, 128), # l1 (neurons)
            (0, 4), # num_hidden_layers
            ("ReLU", "Sigmoid", "Tanh", "LeakyReLU") # activation (factor!)
        ],
        var_type=["float", "int", "int", "factor"],
        max_iter=50,
        seed=42
    )

    # Run optimization
    print("\nStarting optimization...")
    result = optimizer.optimize()

    # Display results
    print("\n" + "=" * 80)
    print("OPTIMIZATION RESULTS")
```

```

print("=" * 80)
print(f"Best learning rate: {10**result.x[0]:.6f}")
print(f"Best layer size (l1): {int(result.x[1])}")
print(f"Best num hidden layers: {int(result.x[2])}")
print(f"Best activation function: {result.x[3]}") # String value!
print(f"Best test MSE: {result.fun:.4f}")

# Show top 5 configurations
print("\n" + "=" * 80)
print("TOP 5 CONFIGURATIONS")
print("=" * 80)
sorted_indices = result.y.argsort()[:5]
print(f"{'Rank':<6} {'LR':<12} {'L1':<6} {'Layers':<8} "
      f"{'Activation':<12} {'MSE':<10}")
print("-" * 80)
for rank, idx in enumerate(sorted_indices, 1):
    lr = 10 ** result.X[idx, 0]
    l1 = int(result.X[idx, 1])
    layers = int(result.X[idx, 2])
    activation = result.X[idx, 3]
    mse = result.y[idx]
    print(f"{'rank':<6} {'lr':<12.6f} {'l1':<6} {'layers':<8} "
          f"{'activation':<12} {'mse':<10.4f}")

# Train final model with best configuration
print("\n" + "=" * 80)
print("TRAINING FINAL MODEL")
print("=" * 80)

best_lr = 10 ** result.x[0]
best_l1 = int(result.x[1])
best_layers = int(result.x[2])
best_activation = result.x[3]

print(f"Configuration: lr={best_lr:.6f}, l1={best_l1}, "
      f"layers={best_layers}, activation={best_activation}")

train_loader, test_loader, _ = get_diabetes_dataloaders(
    test_size=0.2,
    batch_size=32,
    random_state=42
)

final_model = LinearRegressor(

```

6. Factor Variables for Categorical Hyperparameters

```
        input_dim=10,
        output_dim=1,
        l1=best_l1,
        num_hidden_layers=best_layers,
        activation=best_activation
    )

    optimizer_final = final_model.get_optimizer("Adam", lr=best_lr)
    criterion = nn.MSELoss()

    # Extended training
    num_epochs = 100
    print(f"\nTraining for {num_epochs} epochs...")
    for epoch in range(num_epochs):
        final_model.train()
        train_loss = 0.0
        for batch_X, batch_y in train_loader:
            predictions = final_model(batch_X)
            loss = criterion(predictions, batch_y)
            optimizer_final.zero_grad()
            loss.backward()
            optimizer_final.step()
            train_loss += loss.item()

        if (epoch + 1) % 20 == 0:
            avg_train_loss = train_loss / len(train_loader)
            print(f"Epoch {epoch+1}/{num_epochs}: Train MSE = {avg_train_loss:.4f}")

    # Final evaluation
    final_model.eval()
    final_test_loss = 0.0
    with torch.no_grad():
        for batch_X, batch_y in test_loader:
            predictions = final_model(batch_X)
            final_test_loss += criterion(predictions, batch_y).item()

    final_avg_loss = final_test_loss / len(test_loader)
    print(f"\nFinal Test MSE: {final_avg_loss:.4f}")
    print("=" * 80)

if __name__ == "__main__":
    main()
```


6.7. Complete Example: Full Workflow

```
=====
Neural Network Hyperparameter Optimization with Factor Variables
=====
```

```
Starting optimization...
Testing: lr=0.007002, l1=101, layers=2, activation=ReLU
  → Test MSE: 26611.9297
Testing: lr=0.000604, l1=50, layers=2, activation=ReLU
  → Test MSE: 26643.1562
Testing: lr=0.000149, l1=67, layers=1, activation=Tanh
  → Test MSE: 26672.6816
Testing: lr=0.000296, l1=40, layers=0, activation=Tanh
  → Test MSE: 26648.8379
Testing: lr=0.004887, l1=116, layers=2, activation=Sigmoid
  → Test MSE: 26630.6296
Testing: lr=0.001772, l1=124, layers=3, activation=Sigmoid
  → Test MSE: 26525.2806
Testing: lr=0.001107, l1=36, layers=4, activation=Sigmoid
  → Test MSE: 26650.2389
Testing: lr=0.003708, l1=20, layers=1, activation=LeakyReLU
  → Test MSE: 26591.1322
Testing: lr=0.000861, l1=90, layers=1, activation=Tanh
  → Test MSE: 26643.9980
Testing: lr=0.000237, l1=78, layers=3, activation=Tanh
  → Test MSE: 26622.8711
Testing: lr=0.001762, l1=124, layers=3, activation=Sigmoid
  → Test MSE: 26557.4219
Testing: lr=0.009463, l1=24, layers=0, activation=LeakyReLU
  → Test MSE: 26649.9180
Testing: lr=0.000761, l1=94, layers=4, activation=Tanh
  → Test MSE: 26638.6699
Testing: lr=0.003722, l1=82, layers=1, activation=Tanh
  → Test MSE: 26592.6934
Testing: lr=0.007645, l1=96, layers=3, activation=Sigmoid
  → Test MSE: 26584.0163
Testing: lr=0.000769, l1=40, layers=1, activation=Tanh
  → Test MSE: 26668.7474
Testing: lr=0.000235, l1=109, layers=4, activation=LeakyReLU
  → Test MSE: 26622.8704
Testing: lr=0.000359, l1=76, layers=3, activation=Sigmoid
  → Test MSE: 26585.8783
Testing: lr=0.004959, l1=50, layers=2, activation=Tanh
  → Test MSE: 26604.1777
Testing: lr=0.002494, l1=57, layers=3, activation=LeakyReLU
  → Test MSE: 26586.5501
```

6. Factor Variables for Categorical Hyperparameters

Testing: lr=0.005807, l1=20, layers=0, activation=Sigmoid
→ Test MSE: 26652.9232

Testing: lr=0.002939, l1=19, layers=1, activation=Sigmoid
→ Test MSE: 26679.1842

Testing: lr=0.001263, l1=98, layers=4, activation=ReLU
→ Test MSE: 26600.2865

Testing: lr=0.001226, l1=105, layers=3, activation=Sigmoid
→ Test MSE: 26611.2689

Testing: lr=0.004431, l1=32, layers=1, activation=Sigmoid
→ Test MSE: 26626.6751

Testing: lr=0.001367, l1=58, layers=2, activation=Sigmoid
→ Test MSE: 26571.4199

Testing: lr=0.006778, l1=81, layers=4, activation=Tanh
→ Test MSE: 26540.2637

Testing: lr=0.002189, l1=112, layers=4, activation=Sigmoid
→ Test MSE: 26594.3451

Testing: lr=0.004558, l1=24, layers=2, activation=Tanh
→ Test MSE: 26638.2715

Testing: lr=0.000658, l1=125, layers=0, activation=Tanh
→ Test MSE: 26696.8926

Testing: lr=0.000272, l1=119, layers=2, activation=Tanh
→ Test MSE: 26619.2109

Testing: lr=0.000133, l1=64, layers=2, activation=Tanh
→ Test MSE: 26653.7038

Testing: lr=0.002172, l1=70, layers=2, activation=LeakyReLU
→ Test MSE: 26592.5020

Testing: lr=0.000223, l1=28, layers=3, activation=Sigmoid
→ Test MSE: 26620.8477

Testing: lr=0.006065, l1=53, layers=3, activation=Sigmoid
→ Test MSE: 26601.4251

Testing: lr=0.000352, l1=42, layers=4, activation=ReLU
→ Test MSE: 26623.2168

Testing: lr=0.003464, l1=124, layers=2, activation=Tanh
→ Test MSE: 26596.6315

Testing: lr=0.000219, l1=37, layers=3, activation=ReLU
→ Test MSE: 26585.4043

Testing: lr=0.002619, l1=70, layers=3, activation=ReLU
→ Test MSE: 26637.3704

Testing: lr=0.004686, l1=123, layers=2, activation=ReLU
→ Test MSE: 26569.2142

Testing: lr=0.000165, l1=44, layers=0, activation=ReLU
→ Test MSE: 26532.1517

Testing: lr=0.000917, l1=93, layers=1, activation=Sigmoid
→ Test MSE: 26705.4668

Testing: lr=0.001789, l1=117, layers=1, activation=Tanh

6.7. Complete Example: Full Workflow

```
→ Test MSE: 26634.4108
Testing: lr=0.000134, l1=101, layers=4, activation=Sigmoid
→ Test MSE: 26644.3034
Testing: lr=0.004937, l1=108, layers=2, activation=ReLU
→ Test MSE: 26586.0710
Testing: lr=0.004047, l1=93, layers=1, activation=ReLU
→ Test MSE: 26681.0638
Testing: lr=0.000997, l1=112, layers=4, activation=Tanh
→ Test MSE: 26598.5638
Testing: lr=0.005449, l1=52, layers=4, activation=Sigmoid
→ Test MSE: 26679.7363
Testing: lr=0.000409, l1=63, layers=3, activation=Sigmoid
→ Test MSE: 26713.3411
Testing: lr=0.000371, l1=45, layers=3, activation=LeakyReLU
→ Test MSE: 26627.5566
```

OPTIMIZATION RESULTS

```
Best learning rate: 0.001772
Best layer size (l1): 124
Best num hidden layers: 3
Best activation function: Sigmoid
Best test MSE: 26525.2806
```

TOP 5 CONFIGURATIONS

Rank	LR	L1	Layers	Activation	MSE
1	0.001772	124	3	Sigmoid	26525.2806
2	0.000165	44	0	ReLU	26532.1517
3	0.006778	81	4	Tanh	26540.2637
4	0.001762	124	3	Sigmoid	26557.4219
5	0.004686	123	2	ReLU	26569.2142

TRAINING FINAL MODEL

```
Configuration: lr=0.001772, l1=124, layers=3, activation=Sigmoid
```

```
Training for 100 epochs...
```

```
Epoch 20/100: Train MSE = 27789.9785
```

```
Epoch 40/100: Train MSE = 27511.7025
```

```
Epoch 60/100: Train MSE = 34997.5954
```

6. Factor Variables for Categorical Hyperparameters

Epoch 80/100: Train MSE = 27670.7340
Epoch 100/100: Train MSE = 28047.8629

Final Test MSE: 26527.3613

=====

6.8. Best Practices

6.8.1. Do's

Use descriptive string values

```
bounds=[("xavier_uniform", "kaiming_normal", "orthogonal")]
```

Explicitly specify `var_type` for clarity

```
var_type=["float", "int", "factor"]
```

Access results as strings

```
# Example: Accessing factor variable results as strings
# (This assumes you've run an optimization with activation as a factor variable)

# If you have a result from the previous examples:
# best_activation = result.x[3] # For 4-parameter optimization
# Or for simpler cases:
# best_activation = result.x[0] # For single-parameter optimization

# Example with inline optimization:
from spotoptim import SpotOptim
import numpy as np

def quick_test(X):
    results = []
    for params in X:
        activation = params[0]
        score = {"ReLU": 3500, "Tanh": 3600}.get(activation, 4000)
        results.append(score + np.random.normal(0, 50))
    return np.array(results)

opt = SpotOptim(
    fun=quick_test,
```

```

    bounds=[("ReLU", "Tanh")],
    var_type=["factor"],
    max_iter=10,
    seed=42
)
result = opt.optimize()

# Access as string - this is the correct way
best_activation = result.x[0] # String value like "ReLU"
print(f"Best activation: {best_activation} (type: {type(best_activation).__name__})")

# You can use it directly in your model
# model = LinearRegressor(activation=best_activation)

```

Best activation: ReLU (type: str)

Mix factor variables with numeric/integer variables

```

bounds=[(-4, -2), (16, 128), ("ReLU", "Tanh")]
var_type=["float", "int", "factor"]

```

6.8.2. Don'ts

Don't use integers in factor bounds

```

# Wrong: Use strings, not integers
bounds=[(0, 1, 2)] # Wrong!
bounds=[("ReLU", "Sigmoid", "Tanh")] # Correct!

```

Don't expect integers in objective function

```

def objective(X):
    activation = X[0][2]
    # activation is a string, not an integer!
    # Don't do: if activation == 0: # Wrong!
    # Do: if activation == "ReLU": # Correct!

```

Don't manually convert factor variables

```

# SpotOptim handles conversion automatically
# Don't do manual mapping in your objective function

```

6. Factor Variables for Categorical Hyperparameters

Don't use empty tuples

```
# Wrong: Empty tuple
bounds=[()]

# Correct: At least one string
bounds=[("ReLU",)] # Single choice (will be treated as fixed)
```

6.9. Troubleshooting

6.9.1. Common Issues

Issue: Objective function receives integers instead of strings

Solution: Ensure you're using the latest version of SpotOptim with factor variable support. Factor variables are automatically converted before calling the objective function.

Issue: `ValueError: could not convert string to float`

Solution: This occurs if there's a version mismatch. Update SpotOptim to ensure the object array conversion is implemented correctly.

Issue: Results show integers instead of strings

Solution: Check that you're accessing `result.x` (mapped values) instead of internal arrays. The result object automatically maps factor variables to their original strings.

Issue: Single-level factor variables cause dimension reduction

Behavior: If a factor variable has only one choice, e.g., `("ReLU",)`, SpotOptim treats it as a fixed dimension and may reduce the dimensionality. This is expected behavior.

Solution: Use at least two choices for optimization, or remove single-choice dimensions from bounds.

6.10. Summary

Factor variables in SpotOptim enable:

- **Categorical optimization:** Optimize over discrete string choices
- **Automatic conversion:** Seamless integer string mapping
- **Neural network hyperparameters:** Optimize activation functions, optimizers, etc.
- **Mixed variable types:** Combine with continuous and integer variables
- **Clean interface:** Objective functions work with strings directly
- **String results:** Final results contain original string values

Factor variables make categorical hyperparameter optimization as easy as continuous optimization!

6.11. Jupyter Notebook

Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

7. Variable Transformations for Search Space Scaling

SpotOptim supports automatic variable transformations to improve optimization in scaled search spaces. Instead of manually handling transformations (e.g., log-scale for learning rates), you can specify transformations via the `var_trans` parameter, and SpotOptim handles everything internally.

7.1. Overview

What are Variable Transformations?

Variable transformations allow you to specify how search space dimensions should be scaled during optimization:

1. **Original scale** (user interface): Input bounds, output results, plots
2. **Transformed scale** (internal): Surrogate modeling, acquisition optimization
3. **Automatic conversion**: SpotOptim handles all transformations transparently

Module: `spotoptim.SpotOptim`

Key Features:

- Define transformations via `var_trans` parameter: `["log10", "sqrt", None, ...]`
- Optimization occurs in transformed space (better for surrogate models)
- All external interfaces use original scale (intuitive for users)
- Supported transformations: log10, log/ln, sqrt, exp, square, cube, inv/reciprocal
- Mix transformed and non-transformed variables

7.2. Why Use Transformations?

7.2.1. Problem: Poorly Scaled Search Spaces

Some hyperparameters span multiple orders of magnitude:

7. Variable Transformations for Search Space Scaling

- **Learning rates:** 0.0001 to 1.0 (4 orders of magnitude)
- **Regularization:** 0.001 to 100 (5 orders of magnitude)
- **Network sizes:** 10 to 1000 neurons

Direct optimization in these spaces is inefficient because:

1. Surrogate models struggle with extreme scales
2. Uniform sampling wastes evaluations in unimportant regions
3. Acquisition functions behave poorly with skewed distributions

7.2.2. Solution: Logarithmic and Other Transformations

Transform the space for optimization while maintaining user-friendly interfaces:

```
# Without transformations (manual approach)
bounds = [(-4, 0)] # log10(lr): awkward for users
lr = 10 ** params[0] # Manual transformation in objective

# With transformations (automatic)
bounds = [(0.0001, 1.0)] # lr in natural scale
var_trans = ["log10"] # SpotOptim handles transformation
lr = params[0] # Already in original scale!
```

7.3. Quick Start

7.3.1. Basic Log-Scale Transformation

```
from spotoptim import SpotOptim
import numpy as np

def objective_function(X):
    """Objective receives parameters in ORIGINAL scale."""
    results = []
    for params in X:
        lr = params[0] # Already in [0.001, 0.1] - original scale!
        alpha = params[1] # Already in [0.01, 1.0] - original scale!

        # Simulate model training
        score = (lr - 0.01)**2 + (alpha - 0.1)**2 + np.random.normal(0, 0.01)
        results.append(score)
```

```

    return np.array(results)

# Create optimizer with transformations
optimizer = SpotOptim(
    fun=objective_function,
    bounds=[
        (0.001, 0.1),    # learning rate (original scale)
        (0.01, 1.0)      # alpha (original scale)
    ],
    var_trans=["log10", "log10"], # Both use log10 transformation
    var_name=["lr", "alpha"],
    max_iter=20,
    seed=42
)

# Run optimization
result = optimizer.optimize()

print(f"Best lr: {result.x[0]:.6f}")    # In original scale
print(f"Best alpha: {result.x[1]:.6f}") # In original scale
print(f"Best score: {result.fun:.6f}")

```

```

Best lr: 0.038315
Best alpha: 0.113107
Best score: -0.017287

```

7.4. Supported Transformations

SpotOptim supports the following transformations:

Transformation	Forward ($x \rightarrow t$)	Inverse ($t \rightarrow x$)	Use Case
"log10"	$t = \log(x)$	$x = 10^t$	Learning rates, regularization
"log" or "ln"	$t = \ln(x)$	$x = e^t$	Natural exponential scales
"sqrt"	$t = \sqrt{x}$	$x = t^2$	Moderate scaling
"exp"	$t = e^x$	$x = \ln(t)$	Inverse of natural log

7. Variable Transformations for Search Space Scaling

Transformation	Forward ($x \rightarrow t$)	Inverse ($t \rightarrow x$)	Use Case
"square"	$t = x^2$	$x = \sqrt{t}$	Inverse of sqrt
"cube"	$t = x^3$	$x = \sqrt[3]{t}$	Strong scaling
"inv" or "reciprocal"	$t = 1/x$	$x = 1/t$	Reciprocal relationships
"pow(base, x)"	$t = \text{base}^x$	$x = \log_{\text{base}}(t)$	Power scaling (e.g. <code>pow(2, x)</code>)
"log(x, base)"	$t = \log_{\text{base}}(x)$	$x = \text{base}^t$	Log scaling with custom base
None or "id"	$t = x$	$x = t$	No transformation

7.4.1. Transformation Guidelines

When to use "log10" or "log":

- Parameters spanning multiple orders of magnitude
- Learning rates: $(1e-5, 1e-1) \rightarrow$ uniform sampling in log space
- Regularization parameters: $(1e-6, 1e2)$
- Batch sizes, hidden units when range is large

When to use "sqrt":

- Moderate scaling (1-2 orders of magnitude)
- Batch sizes: (16, 512)
- Number of neurons: (32, 256)

When to use "inv" (reciprocal):

- Inverse relationships (e.g., $1/\text{temperature}$)
- When smaller values are more important

Dynamic Transformations:

- "pow(base, x)": Exponential scaling. Example: `transform="pow(2, x)"` with bounds [2, 5] leads to internal search on [4, 32].
- "log(x, base)": Logarithmic scaling with custom base. Example: `transform="log(x, 2)"`.

When to use None:

- Parameters with narrow ranges
- Already well-scaled parameters
- Categorical indices (use with `var_type=["factor"]`)

7.5. Detailed Examples

7.5.1. Example 1: Neural Network Hyperparameter Tuning

```
import torch
import torch.nn as nn
from spotoptim import SpotOptim
from spotoptim.data import get_diabetes_dataloaders
from spotoptim.nn.linear_regressor import LinearRegressor
import numpy as np

def train_neural_network(X):
    """Train neural network with hyperparameters in original scale."""
    results = []

    for params in X:
        # All parameters in original scale
        hidden_size = int(params[0]) # [16, 256]
        num_layers = int(params[1]) # [1, 4]
        lr = params[2] # [0.0001, 0.1]
        weight_decay = params[3] # [1e-6, 0.01]

        print(f"Training: hidden={hidden_size}, layers={num_layers}, "
              f"lr={lr:.6f}, wd={weight_decay:.6f}")

        # Load data
        train_loader, test_loader, _ = get_diabetes_dataloaders(batch_size=32)

        # Create model
        model = LinearRegressor(
            input_dim=10,
            output_dim=1,
            l1=hidden_size,
            num_hidden_layers=num_layers,
            activation="ReLU",
            lr=lr
        )
```

7. Variable Transformations for Search Space Scaling

```
# Get optimizer with weight decay
optimizer = torch.optim.Adam(model.parameters(),
                              lr=lr,
                              weight_decay=weight_decay)

# Train
model.train()
for epoch in range(50):
    for batch_x, batch_y in train_loader:
        optimizer.zero_grad()
        outputs = model(batch_x)
        loss = nn.MSELoss()(outputs, batch_y)
        loss.backward()
        optimizer.step()

# Evaluate
model.eval()
total_loss = 0
with torch.no_grad():
    for batch_x, batch_y in test_loader:
        outputs = model(batch_x)
        loss = nn.MSELoss()(outputs, batch_y)
        total_loss += loss.item()

avg_loss = total_loss / len(test_loader)
results.append(avg_loss)

return np.array(results)

# Create optimizer with appropriate transformations
optimizer = SpotOptim(
    fun=train_neural_network,
    bounds=[
        (16, 256),          # hidden_size: moderate range
        (1, 4),             # num_layers: small range
        (0.0001, 0.1),      # lr: 3 orders of magnitude
        (1e-6, 0.01)        # weight_decay: 4 orders of magnitude
    ],
    var_trans=[
        "sqrt",             # sqrt for hidden_size
        None,               # no transformation for num_layers
        "log10",            # log10 for learning rate
        "log10"             # log10 for weight_decay
    ],
```

```

var_type=["int", "int", "float", "float"],
var_name=["hidden_size", "num_layers", "lr", "weight_decay"],
max_iter=30,
n_initial=10,
seed=42
)

result = optimizer.optimize()

print("\nBest Configuration:")
print(f"  Hidden Size: {int(result.x[0])}")
print(f"  Num Layers: {int(result.x[1])}")
print(f"  Learning Rate: {result.x[2]:.6f}")
print(f"  Weight Decay: {result.x[3]:.8f}")
print(f"  Best Loss: {result.fun:.6f}")

```

```

Training: hidden=225, layers=3, lr=0.001748, wd=0.000001
Training: hidden=81, layers=2, lr=0.003730, wd=0.000003
Training: hidden=25, layers=2, lr=0.000615, wd=0.001695
Training: hidden=49, layers=2, lr=0.000147, wd=0.000204
Training: hidden=196, layers=4, lr=0.007107, wd=0.000009
Training: hidden=121, layers=4, lr=0.025632, wd=0.000017
Training: hidden=100, layers=2, lr=0.072441, wd=0.000096
Training: hidden=169, layers=1, lr=0.000238, wd=0.004102
Training: hidden=100, layers=3, lr=0.001146, wd=0.001331
Training: hidden=36, layers=3, lr=0.021475, wd=0.000340
Training: hidden=100, layers=3, lr=0.001303, wd=0.001200
Training: hidden=81, layers=2, lr=0.009723, wd=0.000004
Training: hidden=100, layers=2, lr=0.006052, wd=0.000002
Training: hidden=25, layers=3, lr=0.049720, wd=0.000793
Training: hidden=225, layers=4, lr=0.004232, wd=0.000001
Training: hidden=36, layers=3, lr=0.083860, wd=0.000386
Training: hidden=100, layers=4, lr=0.001011, wd=0.003569
Training: hidden=16, layers=2, lr=0.002796, wd=0.003487
Training: hidden=81, layers=2, lr=0.005363, wd=0.000001
Training: hidden=225, layers=3, lr=0.002943, wd=0.000001
Training: hidden=100, layers=2, lr=0.004872, wd=0.000004
Training: hidden=100, layers=2, lr=0.004560, wd=0.000005
Training: hidden=100, layers=2, lr=0.003155, wd=0.000003
Training: hidden=100, layers=2, lr=0.005634, wd=0.000003
Training: hidden=100, layers=2, lr=0.004156, wd=0.000003
Training: hidden=100, layers=2, lr=0.004617, wd=0.000001
Training: hidden=100, layers=3, lr=0.001365, wd=0.002581
Training: hidden=81, layers=2, lr=0.004767, wd=0.000002

```

7. Variable Transformations for Search Space Scaling

Training: hidden=81, layers=2, lr=0.004799, wd=0.000002

Training: hidden=81, layers=2, lr=0.010326, wd=0.000005

Best Configuration:

Hidden Size: 225

Num Layers: 3

Learning Rate: 0.001748

Weight Decay: 0.00000132

Best Loss: 2528.505452

7.5.2. Example 2: Physics-Informed Neural Networks (PINNs)

```
from spotoptim import SpotOptim
import numpy as np

def train_pinn(X):
    """Train PINN with hyperparameters in original scale."""
    results = []

    for params in X:
        neurons = int(params[0])      # [16, 128]
        layers = int(params[1])       # [1, 4]
        lr = params[2]                 # [0.1, 10.0]
        alpha = params[3]              # [0.01, 1.0]

        # Simulate PINN training
        # In practice, this would train an actual PINN model
        val_error = 0.1 * (1/lr) + 0.05 * alpha + np.random.normal(0, 0.01)
        results.append(val_error)

    return np.array(results)

# Define optimization with transformations
optimizer = SpotOptim(
    fun=train_pinn,
    bounds=[
        (16, 128),      # neurons
        (1, 4),          # layers
        (0.1, 10.0),    # lr: covers 2 orders of magnitude
        (0.01, 1.0)     # alpha: covers 2 orders of magnitude
    ],
    var_trans=[None, None, "log10", "log10"],
    var_type=["int", "int", "float", "float"],
```



```

var_name=["neurons", "layers", "lr", "alpha"],
max_iter=20,
seed=42
)

result = optimizer.optimize()
optimizer.print_best(result)

```

Best Solution Found:

```

-----
neurons: 74
layers: 2
lr: 8.0660
alpha: 0.0980
Objective Value: 0.0072
Total Evaluations: 20

```

7.5.3. Example 3: Mixing Transformations

```

from spotoptim import SpotOptim
import numpy as np

def complex_objective(X):
    """Objective with multiple transformation types."""
    results = []

    for params in X:
        # All in original scale
        x1 = params[0] # sqrt-transformed: [10, 1000]
        x2 = params[1] # log10-transformed: [0.001, 1.0]
        x3 = params[2] # no transformation: [-5, 5]
        x4 = params[3] # reciprocal: [0.1, 10]

        # Complex function
        result = (
            (x1/100 - 5)**2 +
            (np.log10(x2) + 1.5)**2 +
            x3**2 +
            (1/x4 - 0.2)**2
        )
        results.append(result)

```

7. Variable Transformations for Search Space Scaling

```
        return np.array(results)

optimizer = SpotOptim(
    fun=complex_objective,
    bounds=[
        (10, 1000),      # x1: moderate range
        (0.001, 1.0),    # x2: log scale
        (-5, 5),         # x3: symmetric range
        (0.1, 10)        # x4: for reciprocal
    ],
    var_trans=["sqrt", "log10", None, "inv"],
    var_name=["x1_sqrt", "x2_log", "x3_linear", "x4_inv"],
    max_iter=30,
    seed=42
)

result = optimizer.optimize()
```

7.6. Viewing Transformations in Tables

The transformation type is displayed in the “trans” column of both design and results tables:

7.6.1. Design Table (Before Optimization)

```
from spotoptim import SpotOptim
import numpy as np

optimizer = SpotOptim(
    fun=lambda X: np.sum(X**2, axis=1),
    bounds=[
        (0.001, 1.0),
        (0.01, 10.0),
        (10, 1000),
        (-5, 5)
    ],
    var_trans=["log10", "log", "sqrt", None],
    var_name=["lr", "alpha", "neurons", "bias"],
    max_iter=10
)
```

7.6. Viewing Transformations in Tables

```
# Display design table
print(optimizer.print_design_table())
```

name	type	lower	upper	default	transform
lr	float	0.0010	1.0000	0.5005	log10
alpha	float	0.0100	10.0000	5.0050	log
neurons	float	10.0000	1000.0000	505.0000	sqrt
bias	float	-5.0000	5.0000	0.0000	-
name	type	lower	upper	default	transform
lr	float	0.0010	1.0000	0.5005	log10
alpha	float	0.0100	10.0000	5.0050	log
neurons	float	10.0000	1000.0000	505.0000	sqrt
bias	float	-5.0000	5.0000	0.0000	-

Output:

name	type	lower	upper	default	trans
lr	num	0.0010	1.0000	0.5005	log10
alpha	num	0.0100	10.0000	5.0050	log
neurons	num	10.0000	1000.0000	505.0000	sqrt
bias	num	-5.0000	5.0000	0.0000	-

7.6.2. Results Table (After Optimization)

```
result = optimizer.optimize()

# Display results with transformations
print(optimizer.print_results_table())
```

name	type	default	lower	upper	tuned	transform
lr	float	0.5005	0.0010	1.0000	0.0022	log10
alpha	float	5.0050	0.0100	10.0000	1.0688	log
neurons	float	505.0000	10.0000	1000.0000	20.3597	sqrt
bias	float	0.0000	-5.0000	5.0000	0.5766	-
name	type	default	lower	upper	tuned	transform

7. Variable Transformations for Search Space Scaling

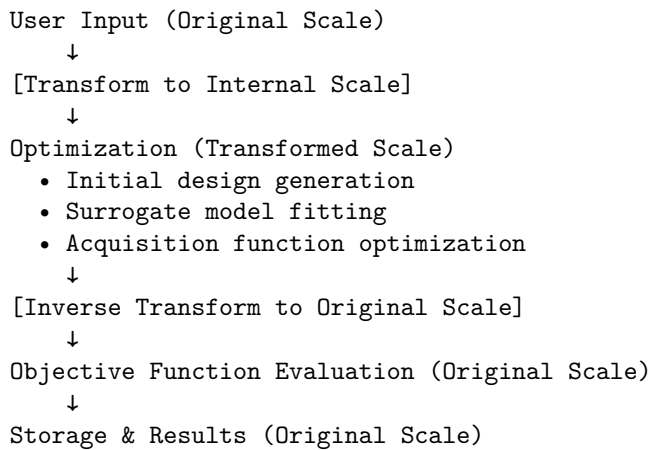
	lr	float	0.5005	0.0010	1.0000	0.0022	log10
	alpha	float	5.0050	0.0100	10.0000	1.0688	log
	neurons	float	505.0000	10.0000	1000.0000	20.3597	sqrt
	bias	float	0.0000	-5.0000	5.0000	0.5766	-

Output shows the “trans” column with transformation types, helping you understand which parameters were optimized in which scale.

7.7. Internal Architecture

Understanding how transformations work internally can help debug issues and understand behavior:

7.7.1. Flow Diagram



7.7.2. Key Components

1. **Bounds Transformation** (`_transform_bounds()`):
 - Called during initialization
 - Transforms `_original_lower` and `_original_upper` → `lower` and `upper`
 - Updates `self.bounds` for internal use
2. **Forward Transformation** (`_transform_X()`):

- Converts from original scale to transformed scale
- Used before surrogate fitting
- Used when comparing distances

3. Inverse Transformation (`_inverse_transform_X()`):

- Converts from transformed scale to original scale
- Used before function evaluation
- Used when storing results

4. Storage:

- `self.X_` stores in **original scale**
- `self.best_x_` stores in **original scale**
- All external-facing data in original scale

7.8. Best Practices

7.8.1. 1. Choose Appropriate Transformations

```
# Good: Log scale for learning rate
bounds = [(1e-5, 1e-1)]
var_trans = ["log10"]

# Bad: No transformation for wide range
bounds = [(1e-5, 1e-1)]
var_trans = [None] # Poor sampling distribution
```

7.8.2. 2. Match Transformation to Range

```
# Wide range (>3 orders of magnitude): use log
bounds = [(1e-6, 1e-2)]
var_trans = ["log10"]

# Moderate range (1-2 orders): use sqrt
bounds = [(10, 500)]
var_trans = ["sqrt"]

# Narrow range (<1 order): no transformation
bounds = [(-1, 1)]
var_trans = [None]
```

7. Variable Transformations for Search Space Scaling

7.8.3. 3. Validate Transformation Choice

```
# Check if transformation makes sense
import numpy as np

# Original space
x_orig = np.linspace(0.001, 1.0, 10)
print("Original:", x_orig)

# Log10 transformed space
x_trans = np.log10(x_orig)
print("Transformed:", x_trans)
print("Range ratio:", np.ptp(x_trans) / np.ptp(x_orig))
# Should be much more uniform distribution
```

7.8.4. 4. Combine with Variable Types

```
# Mix transformations with variable types
optimizer = SpotOptim(
    fun=objective,
    bounds=[
        (10, 200), # int with sqrt
        ("ReLU", "Tanh", "Sigmoid"), # factor (no transform)
        (0.0001, 0.1), # num with log10
        (0.01, 1.0) # num with log10
    ],
    var_type=["int", "factor", "float", "float"],
    var_trans=["sqrt", None, "log10", "log10"],
    var_name=["neurons", "activation", "lr", "dropout"]
)
```

7.9. Troubleshooting

7.9.1. Issue: Values Out of Bounds

Problem: Objective function receives values outside specified bounds.

Solution: This should not happen with transformations. If it does:

```
# Check transformation is applied correctly
print(f"Original bounds: {optimizer._original_lower} to {optimizer._original_upper}")
print(f"Transformed bounds: {optimizer.lower} to {optimizer.upper}")
print(f"Transformations: {optimizer.var_trans}")
```

7.9.2. Issue: Poor Optimization Performance

Problem: Optimization doesn't find good solutions.

Possible causes:

1. Wrong transformation type for the parameter scale
2. Transformation not needed (adding unnecessary complexity)
3. Bounds too wide or too narrow

Solution:

```
# Try different transformations
for trans in [None, "log10", "sqrt"]:
    optimizer = SpotOptim(
        fun=objective,
        bounds=[(0.001, 1.0)],
        var_trans=[trans],
        max_iter=20,
        seed=42
    )
    result = optimizer.optimize()
    print(f"Transformation: {trans}, Best: {result.fun:.6f}")
```

7.9.3. Issue: Transformation Not Applied

Problem: Transformation doesn't seem to affect optimization.

Check:

```
# Verify var_trans length matches dimensions
print(f"Number of dimensions: {len(optimizer.bounds)}")
print(f"Number of transformations: {len(optimizer.var_trans)}")
# These must match!

# Check transformation is not None/"id"
print(f"Transformations: {optimizer.var_trans}")
```

7.10. Comparison: Manual vs Automatic Transformations

7.10.1. Manual Approach (Old Way)

```
def objective_manual(X):
    """Manual transformation - error-prone!"""
    results = []
    for params in X:
        # Must remember to transform
        lr = 10 ** params[0] # Was in log scale
        alpha = 10 ** params[1] # Was in log scale

        # Use parameters
        score = compute_score(lr, alpha)
        results.append(score)
    return np.array(results)

# Bounds in log scale - confusing!
optimizer = SpotOptim(
    fun=objective_manual,
    bounds=[(-4, -1), (-2, 0)], # log10 scale
    var_name=["log10_lr", "log10_alpha"] # Confusing names
)

result = optimizer.optimize()
# Must transform back for interpretation
best_lr = 10 ** result.x[0]
best_alpha = 10 ** result.x[1]
```

7.10.2. Automatic Approach (New Way)

```
def objective_auto(X):
    """Automatic transformation - clean!"""
    results = []
    for params in X:
        # Already in original scale
        lr = params[0]
        alpha = params[1]

        # Use parameters directly
        score = compute_score(lr, alpha)
```



```

        results.append(score)
    return np.array(results)

# Bounds in natural scale - intuitive!
optimizer = SpotOptim(
    fun=objective_auto,
    bounds=[(0.0001, 0.1), (0.01, 1.0)],
    var_trans=["log10", "log10"], # Specify transformation
    var_name=["lr", "alpha"] # Natural names
)

result = optimizer.optimize()
# Results already in original scale
best_lr = result.x[0]
best_alpha = result.x[1]

```

7.11. Summary

Key Takeaways:

1. Use `var_trans` to specify transformations for each dimension
2. Transformations improve optimization for poorly scaled spaces
3. All user interfaces (bounds, results, plots) use original scale
4. Optimization happens internally in transformed space
5. Common transformations: "log10" for learning rates, "sqrt" for moderate scaling
6. View transformations in tables with "trans" column

When to Use:

- Parameters spanning multiple orders of magnitude → "log10" or "log"
- Moderate scaling (1-2 orders) → "sqrt"
- Reciprocal relationships → "inv"
- Well-scaled parameters → None

Benefits:

- Better surrogate model performance
- More efficient sampling
- Improved optimization convergence
- User-friendly interface (no manual transformations in objective function)

7.12. Jupyter Notebook

i Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

Part III.

Visualization

8. Surrogate Model Visualization

This document describes the `plot_surrogate()` method added to the `SpotOptim` class, which provides visualization capabilities similar to the `plotkd()` function in the `spotpython` package.

8.1. Overview

The `plot_surrogate()` method creates a comprehensive 4-panel visualization of the fitted surrogate model, showing both predictions and uncertainty estimates across two selected dimensions.

8.2. Features

- **3D Surface Plots:** Visualize the surrogate's predictions and uncertainty as 3D surfaces
- **Contour Plots:** View 2D contours with overlaid evaluation points
- **Multi-dimensional Support:** Visualize any two dimensions of higher-dimensional problems
- **Customizable Appearance:** Control colors, resolution, transparency, and more

8.3. Usage

8.3.1. Basic Usage

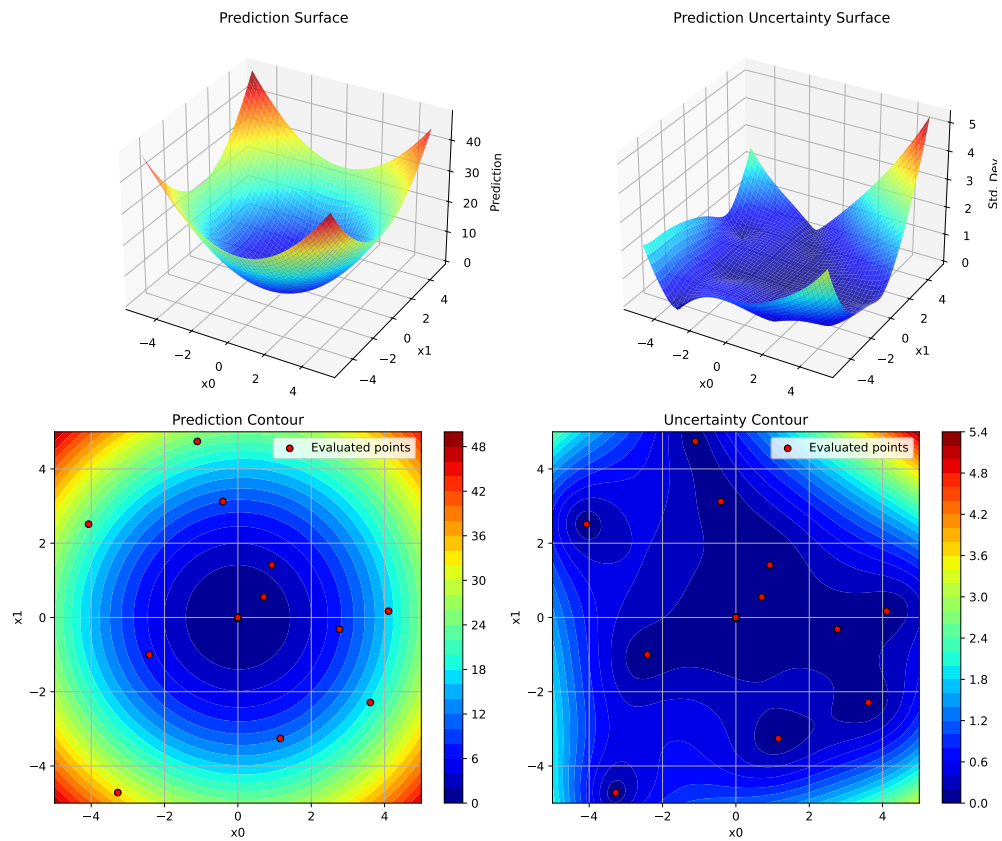
```
import numpy as np
from spotoptim import SpotOptim

# Define objective function
def sphere(X):
    return np.sum(X**2, axis=1)
```

8. Surrogate Model Visualization

```
# Run optimization
optimizer = SpotOptim(fun=sphere, bounds=[(-5, 5), (-5, 5)], max_iter=20)
result = optimizer.optimize()

# Visualize the surrogate model
optimizer.plot_surrogate(i=0, j=1, show=True)
```



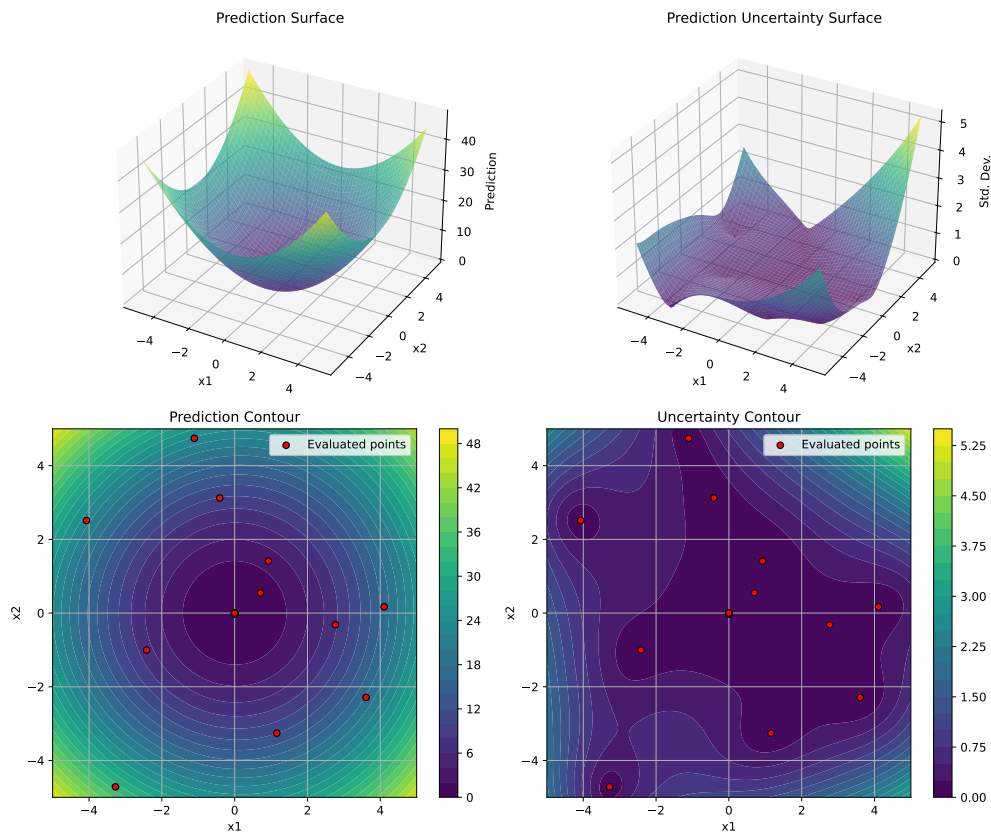
8.3.2. With Custom Parameters

```
optimizer.plot_surrogate(  
    i=0, # First dimension to plot  
    j=1, # Second dimension to plot  
    var_name=['x1', 'x2'], # Variable names for axes  
    add_points=True, # Show evaluated points
```

```

cmap='viridis',          # Colormap
alpha=0.7,               # Surface transparency
num=100,                 # Grid resolution
contour_levels=25,       # Number of contour levels
grid_visible=True,       # Show grid on contours
figsize=(12, 10),        # Figure size
show=True                # Display immediately
)

```



8.3.3. Higher-Dimensional Problems

For problems with more than 2 dimensions, `plot_surrogate()` creates a 2D slice by fixing all other dimensions at their mean values:

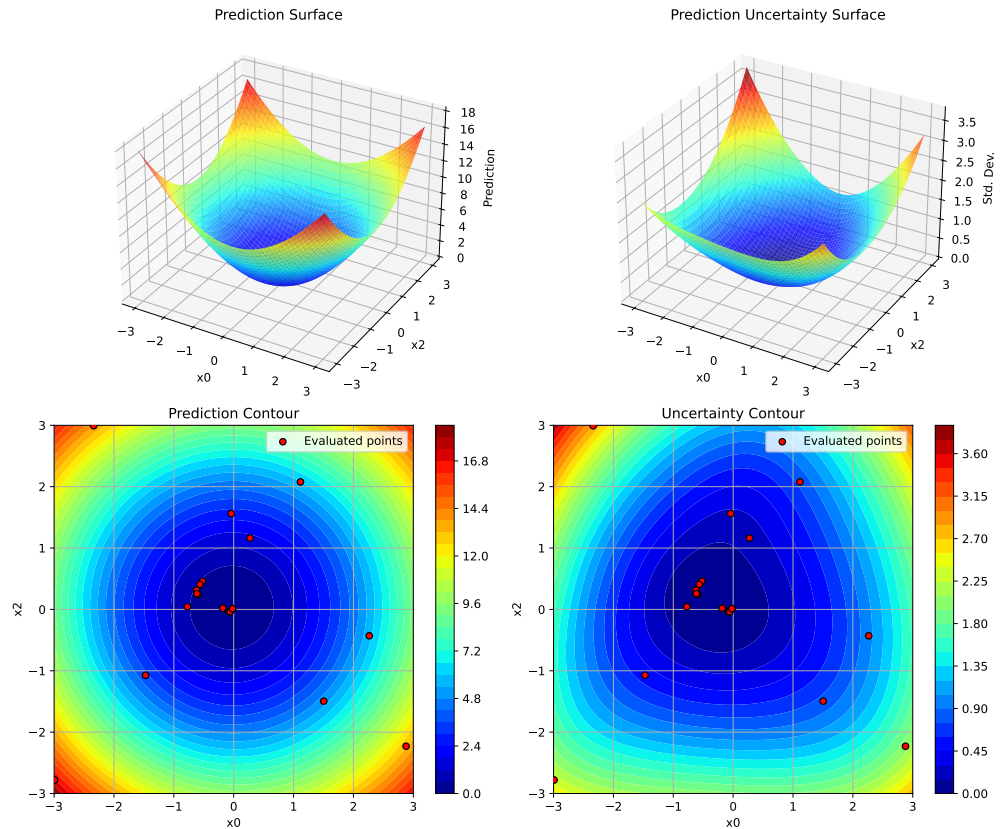
8. Surrogate Model Visualization

```
# 4D optimization problem
def sphere_4d(X):
    return np.sum(X**2, axis=1)

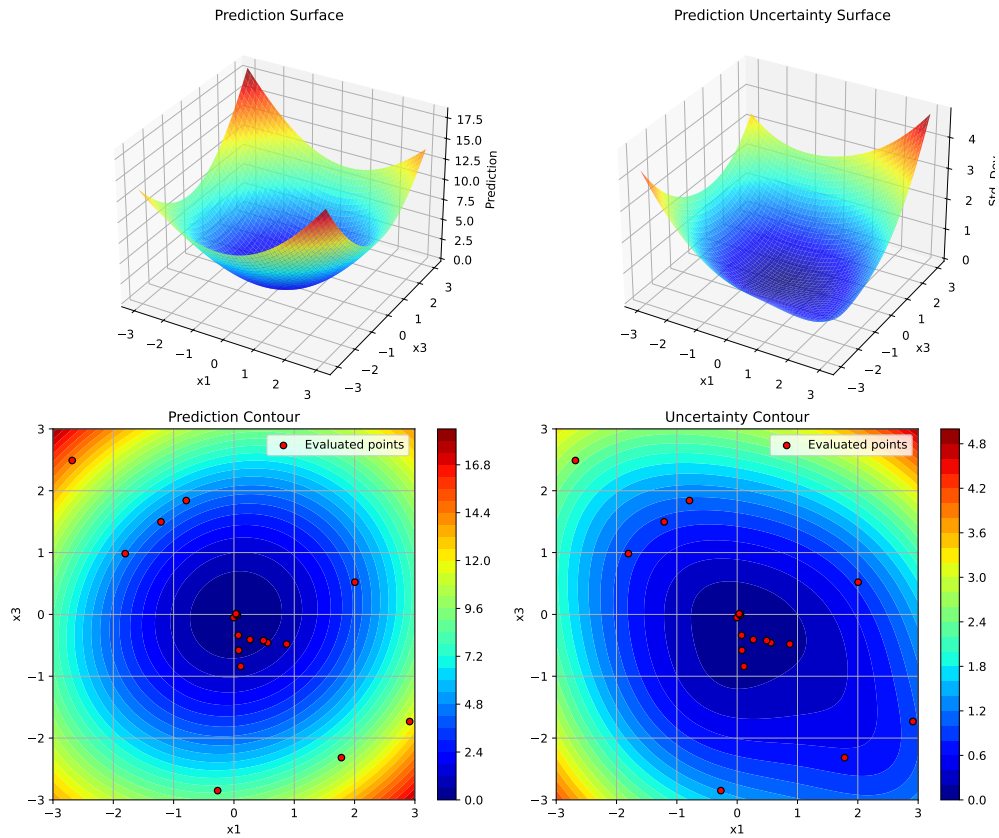
bounds = [(-3, 3)] * 4
optimizer = SpotOptim(fun=sphere_4d, bounds=bounds, max_iter=20)
result = optimizer.optimize()

# Visualize dimensions 0 and 2 (dimensions 1 and 3 fixed at mean)
optimizer.plot_surrogate(
    i=0, j=2,
    var_name=['x0', 'x1', 'x2', 'x3']
)

# Visualize different dimension pair
optimizer.plot_surrogate(i=1, j=3, var_name=['x0', 'x1', 'x2', 'x3'])
```



8.4. Plot Interpretation



8.4. Plot Interpretation

The visualization consists of 4 panels:

8.4.1. Top Left: Prediction Surface

- Shows the surrogate model's predicted function values as a 3D surface
- Helps understand the model's belief about the objective function landscape
- Lower values (blue in default colormap) indicate predicted minima

8.4.2. Top Right: Prediction Uncertainty Surface

- Shows the standard deviation of predictions as a 3D surface
- Indicates where the model is uncertain and might benefit from more samples

8. Surrogate Model Visualization

- Lower values (blue) indicate high confidence, higher values (red) indicate uncertainty

8.4.3. Bottom Left: Prediction Contour with Points

- 2D contour plot of predictions
- Red dots show the actual points evaluated during optimization
- Useful for understanding the exploration-exploitation trade-off

8.4.4. Bottom Right: Uncertainty Contour with Points

- 2D contour plot of prediction uncertainty
- Shows how uncertainty decreases around evaluated points
- Helps identify unexplored regions

8.5. Parameters

8.5.1. Dimension Selection

- `i` (int, default=0): Index of first dimension to plot
- `j` (int, default=1): Index of second dimension to plot

8.5.2. Appearance

- `var_name` (list of str, optional): Names for each dimension
- `cmap` (str, default='jet'): Matplotlib colormap name
- `alpha` (float, default=0.8): Surface transparency (0=transparent, 1=opaque)
- `figsize` (tuple, default=(12, 10)): Figure size in inches (width, height)

8.5.3. Grid and Resolution

- `num` (int, default=100): Number of grid points per dimension
- `contour_levels` (int, default=30): Number of contour levels
- `grid_visible` (bool, default=True): Show grid lines on contour plots

8.5.4. Color Scaling

- `vmin` (float, optional): Minimum value for color scale
- `vmax` (float, optional): Maximum value for color scale

8.5.5. Display

- `show` (bool, default=True): Display plot immediately
- `add_points` (bool, default=True): Overlay evaluated points on contours

8.6. Examples

8.6.1. Example 1: 2D Rosenbrock Function

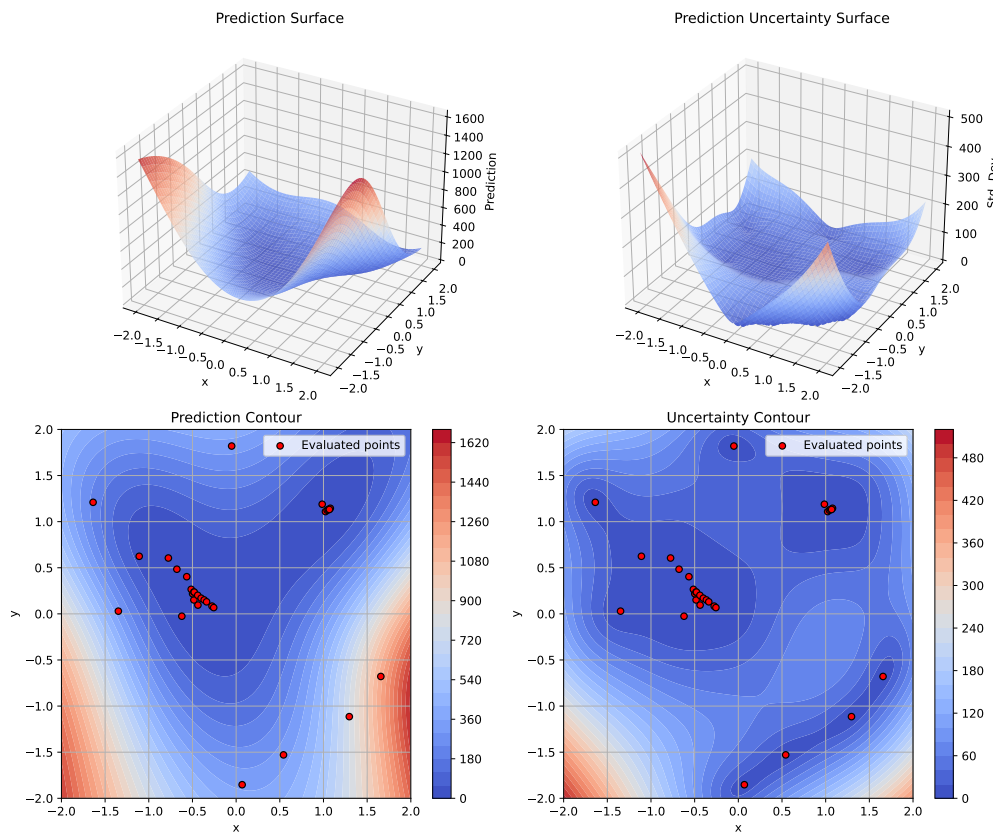
```
import numpy as np
from spotoptim import SpotOptim

def rosenbrock(X):
    X = np.atleast_2d(X)
    x, y = X[:, 0], X[:, 1]
    return (1 - x)**2 + 100 * (y - x**2)**2

optimizer = SpotOptim(
    fun=rosenbrock,
    bounds=[(-2, 2), (-2, 2)],
    max_iter=30,
    seed=42
)
result = optimizer.optimize()

# Visualize with custom colormap
optimizer.plot_surrogate(
    var_name=['x', 'y'],
    cmap='coolwarm',
    add_points=True
)
```

8. Surrogate Model Visualization



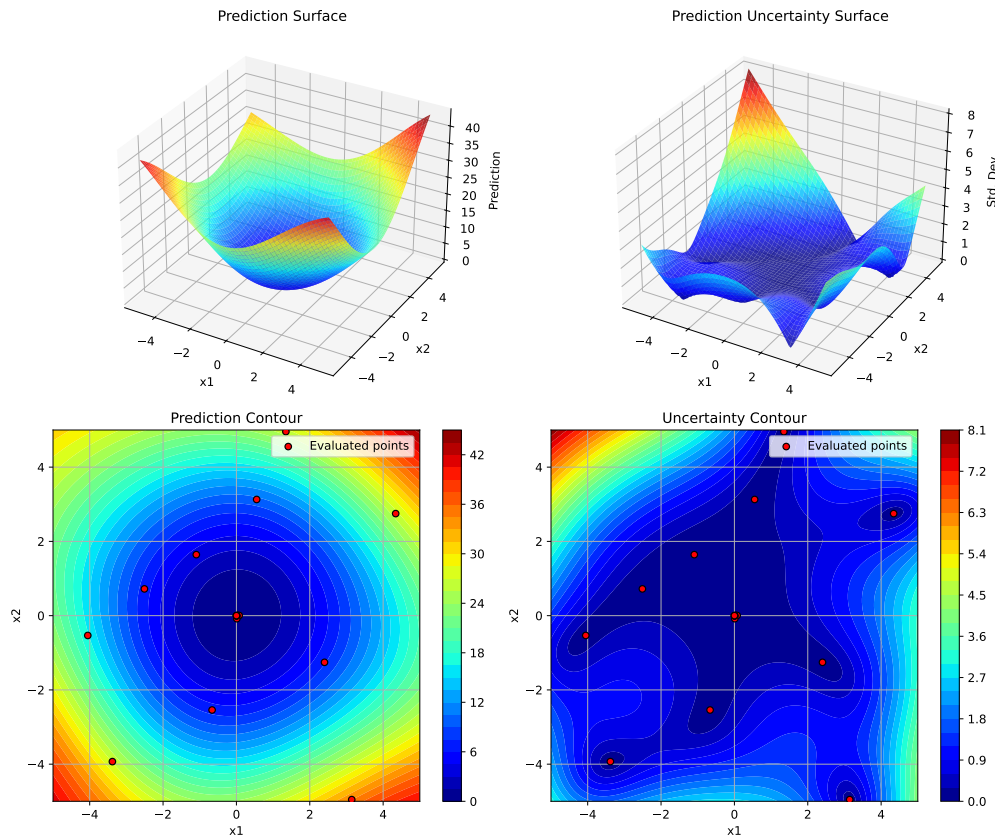
8.6.2. Example 2: Using Kriging Surrogate

```
import numpy as np
from spotoptim import SpotOptim, Kriging

def sphere(X):
    return np.sum(X**2, axis=1)

optimizer = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    surrogate=Kriging(seed=42), # Use Kriging instead of GP
    max_iter=20
)
result = optimizer.optimize()
```

```
# The plotting works the same with any surrogate
optimizer.plot_surrogate(var_name=['x1', 'x2'])
```



8.6.3. Example 3: Comparing Different Dimension Pairs

```
# 3D problem - visualize all dimension pairs
def sphere_3d(X):
    return np.sum(X**2, axis=1)

optimizer = SpotOptim(
    fun=sphere_3d,
    bounds=[(-5, 5)] * 3,
    max_iter=25
)
```

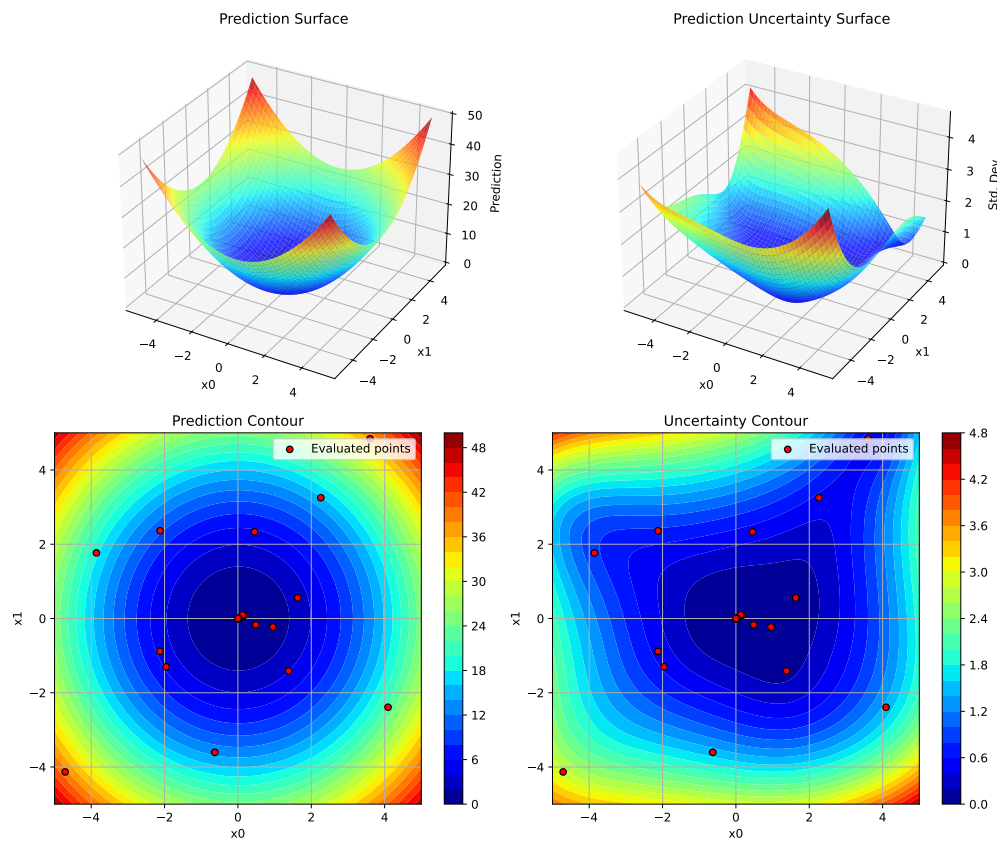
8. Surrogate Model Visualization

```
result = optimizer.optimize()

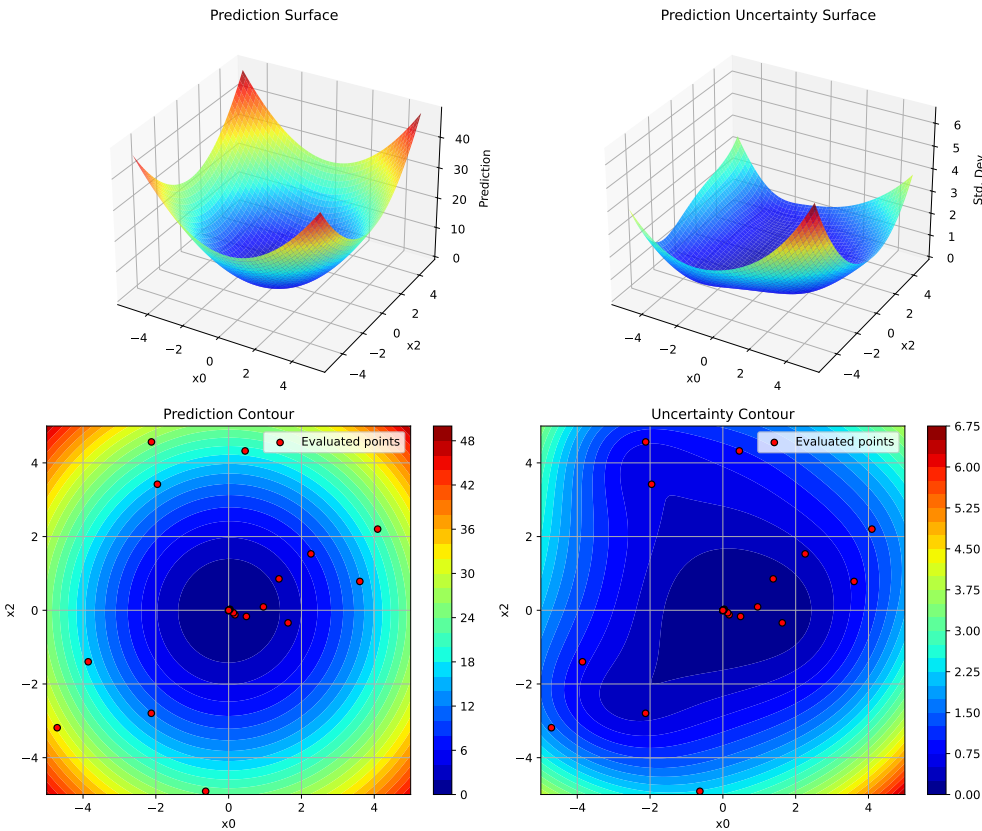
# Dimensions 0 vs 1
optimizer.plot_surrogate(i=0, j=1, var_name=['x0', 'x1', 'x2'])

# Dimensions 0 vs 2
optimizer.plot_surrogate(i=0, j=2, var_name=['x0', 'x1', 'x2'])

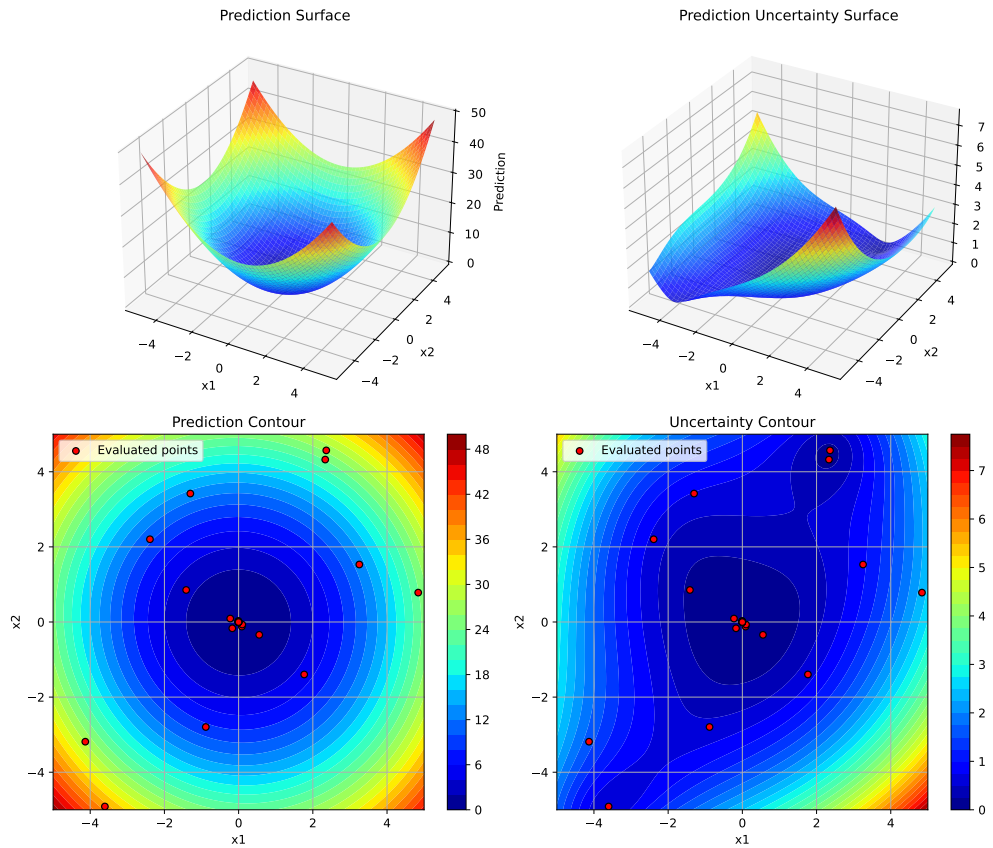
# Dimensions 1 vs 2
optimizer.plot_surrogate(i=1, j=2, var_name=['x0', 'x1', 'x2'])
```



8.6. Examples



8. Surrogate Model Visualization



8.7. Tips and Best Practices

1. **Run Optimization First:** Always call `optimize()` before `plot_surrogate()`
2. **Choose Dimensions Wisely:** For high-dimensional problems, plot dimensions that you suspect are most important or interactive
3. **Adjust Resolution:** Use lower `num` values (e.g., 50) for faster plotting, higher values (e.g., 200) for smoother surfaces
4. **Color Scales:** Set `vmin` and `vmax` explicitly when comparing multiple plots to ensure consistent color scales
5. **Uncertainty Analysis:** High uncertainty areas (bright colors in uncertainty plots) are good candidates for additional sampling
6. **Exploration vs Exploitation:** Red dots clustered in low-prediction areas show exploitation; spread-out dots show exploration

8.8. Comparison with spotpython's plotkd()

The `plot_surrogate()` method is inspired by spotpython's `plotkd()` function but adapted for SpotOptim's simplified interface:

8.8.1. Similarities

- Same 4-panel layout (2 surfaces + 2 contours)
- Visualizes predictions and uncertainty
- Supports dimension selection and customization

8.8.2. Differences

- **Integration:** Method of SpotOptim class (no separate function needed)
- **Simpler:** Fewer parameters, more sensible defaults
- **Automatic:** Uses optimizer's bounds and data automatically
- **Type Handling:** Automatically applies variable type constraints (int/float/factor)

8.9. Error Handling

The method validates inputs and provides clear error messages:

```
# Before optimization runs
optimizer.plot_surrogate() # ValueError: No optimization data available

# Invalid dimension indices
optimizer.plot_surrogate(i=5, j=1) # ValueError: i must be less than n_dim

# Same dimension twice
optimizer.plot_surrogate(i=0, j=0) # ValueError: i and j must be different
```

8.10. Jupyter Notebook

Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

9. TensorBoard Logging in SpotOptim

SpotOptim supports TensorBoard logging for monitoring optimization progress in real-time.

9.1. Quick Start

9.1.1. Enable TensorBoard Logging

```
from spotoptim import SpotOptim
import numpy as np

def sphere(X):
    return np.sum(X**2, axis=1)

optimizer = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=20,
    n_initial=10,
    tensorboard_log=True, # Enable logging
    tensorboard_clean=True,
    verbose=True,
    seed=42
)

result = optimizer.optimize()
print(f"Best value: {result.fun:.6f}")
print(f"Logs saved to: runs/{optimizer.tensorboard_path}")
```

```
Removed old TensorBoard logs: runs/spotoptim_20251220_100850
Cleaned 1 old TensorBoard log directory
TensorBoard logging enabled: runs/spotoptim_20251220_100857
Initial best: f(x) = 2.420807
Iteration 1: New best f(x) = 2.390307
```

9. TensorBoard Logging in SpotOptim

```
Iteration 2: New best f(x) = 1.813139
Iteration 3: New best f(x) = 0.009007
Iteration 4: New best f(x) = 0.001900
Iteration 5: New best f(x) = 0.000973
Iteration 6: New best f(x) = 0.000001
Iteration 7: New best f(x) = 0.000001
Iteration 8: f(x) = 0.000001
Iteration 9: New best f(x) = 0.000001
Iteration 10: f(x) = 0.000001
TensorBoard writer closed. View logs with: tensorboard --logdir=runs/spotoptim_20251220_100857
Best value: 0.000001
Logs saved to: runs/runs/spotoptim_20251220_100857
```

9.1.2. View Logs in TensorBoard

In a separate terminal, run:

```
tensorboard --logdir=runs
```

Then open your browser to <http://localhost:6006>

9.1.3. Cleaning Old Logs

You can automatically remove old TensorBoard logs before starting a new optimization:

```
optimizer = SpotOptim(
    fun=objective,
    bounds=[(-5, 5), (-5, 5)],
    tensorboard_log=True,
    tensorboard_clean=True, # Remove old logs from 'runs' directory
    verbose=True
)
```

Warning

This permanently deletes all subdirectories in the **runs** folder. Make sure to save important logs elsewhere before enabling this feature.

9.1.4. Use Cases

1. **Clean Start** - Remove old logs and create new one:

```
tensorboard_log=True, tensorboard_clean=True
```

2. **Preserve History** - Keep old logs and add new one (default):

```
tensorboard_log=True, tensorboard_clean=False
```

3. **Just Clean** - Remove old logs without new logging:

```
tensorboard_log=False, tensorboard_clean=True
```

9.1.5. Custom Log Directory

Specify a custom path for TensorBoard logs:

```
optimizer = SpotOptim(
    fun=objective,
    bounds=[(-5, 5), (-5, 5)],
    tensorboard_log=True,
    tensorboard_path="my_experiments/run_001",
    ...
)
```

9.1.6. What Gets Logged

9.1.6.1. Scalar Metrics

For Deterministic Functions:

- `y_values/min`: Best (minimum) y value found so far
- `y_values/last`: Most recently evaluated y value
- `X_best/x0`, `X_best/x1`, ...: Coordinates of the best point

For Noisy Functions (repeats > 1):

- `y_values/min`: Best single evaluation
- `y_values/mean_best`: Best mean y value
- `y_values/last`: Most recent evaluation
- `y_variance_at_best`: Variance at the best mean point
- `X_mean_best/x0`, `X_mean_best/x1`, ...: Coordinates of best mean point

9. TensorBoard Logging in SpotOptim

9.1.6.2. Hyperparameters

Each function evaluation is logged with:

- Input coordinates (x0, x1, x2, ...)
- Function value (hp_metric)

This allows you to explore the relationship between hyperparameters and objective values in the HPARAMS tab.

9.1.7. Examples

Example 9.1 (Basic Tensorboard Usage).

```
import numpy as np
from spotoptim import SpotOptim

optimizer = SpotOptim(
    fun=lambda X: np.sum(X**2, axis=1),
    bounds=[(-5, 5), (-5, 5)],
    max_iter=20,
    n_initial=10,
    tensorboard_log=True,
    tensorboard_clean=True,
    verbose=True,
    seed=42
)
result = optimizer.optimize()
print(f"Best value: {result.fun:.6f}")
```

```
Removed old TensorBoard logs: runs/spotoptim_20251220_100857
Cleaned 1 old TensorBoard log directory
TensorBoard logging enabled: runs/spotoptim_20251220_100901
Initial best: f(x) = 2.420807
Iteration 1: New best f(x) = 2.390307
Iteration 2: New best f(x) = 1.813139
Iteration 3: New best f(x) = 0.009007
Iteration 4: New best f(x) = 0.001900
Iteration 5: New best f(x) = 0.000973
Iteration 6: New best f(x) = 0.000001
Iteration 7: New best f(x) = 0.000001
Iteration 8: f(x) = 0.000001
Iteration 9: New best f(x) = 0.000001
Iteration 10: f(x) = 0.000001
```

TensorBoard writer closed. View logs with: `tensorboard --logdir=runs/spotoptim_20251220_100901`
 Best value: 0.000001

Example 9.2 (Noisy Optimization).

```
import numpy as np
from spotoptim import SpotOptim

def noisy_objective(X):
    base = np.sum(X**2, axis=1)
    noise = np.random.normal(0, 0.1, size=base.shape)
    return base + noise

optimizer = SpotOptim(
    fun=noisy_objective,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=20,
    n_initial=10,
    repeats_initial=3,
    repeats_surrogate=2,
    tensorboard_log=True,
    tensorboard_clean=True,
    tensorboard_path="runs/noisy_exp",
    seed=42
)
result = optimizer.optimize()
print(f"Best value: {result.fun:.6f}")
```

Best value: 2.217835

Example 9.3 (With OCBA).

```
import numpy as np
from spotoptim import SpotOptim

def noisy_objective(X):
    base = np.sum(X**2, axis=1)
    noise = np.random.normal(0, 0.1, size=base.shape)
    return base + noise

optimizer = SpotOptim(
    fun=noisy_objective,
    bounds=[(-5, 5), (-5, 5)],
```

9. TensorBoard Logging in SpotOptim

```
max_iter=20,  
n_initial=10,  
repeats_initial=2,  
ocba_delta=3, # Re-evaluate 3 promising points per iteration  
tensorboard_log=True,  
tensorboard_clean=True,  
seed=42  
)  
result = optimizer.optimize()  
print(f"Best value: {result.fun:.6f}")
```

Best value: 2.417607

Example 9.4 (Comparing Multiple Runs). Run multiple optimizations with different settings:

```
# Run 1: Standard  
opt1 = SpotOptim(..., tensorboard_path="runs/standard")  
opt1.optimize()  
  
# Run 2: With OCBA  
opt2 = SpotOptim(..., ocba_delta=3, tensorboard_path="runs/with_ocba")  
opt2.optimize()  
  
# Run 3: More initial points  
opt3 = SpotOptim(..., n_initial=20, tensorboard_path="runs/more_initial")  
opt3.optimize()
```

Then view all runs together:

```
tensorboard --logdir=runs
```

9.2. TensorBoard Features

9.2.1. SCALARS Tab

- View convergence curves
- Compare optimization progress across runs
- Track how metrics change over iterations

9.2.2. HPARAMS Tab

- Explore hyperparameter space
- See which parameter combinations work best
- Identify patterns in successful configurations

9.2.3. Text Tab

- View configuration details
- Check run metadata

Tips

1. **Organize Experiments:** Use descriptive tensorboard_path names:

```
tensorboard_path=f"runs/exp_{date}_{config_name}"
```

2. **Compare Algorithms:** Run multiple optimization strategies and compare:

```
# Different acquisition functions
for acq in ['ei', 'pi', 'y']:
    opt = SpotOptim(..., acquisition=acq, tensorboard_path=f"runs/acq_{acq}")
    opt.optimize()
```

3. **Clean Up Old Runs:** Use tensorboard_clean=True for automatic cleanup, or manually:

```
rm -rf runs/old_experiment
```

4. **Port Conflicts:** If port 6006 is busy, use a different port:

```
tensorboard --logdir=runs --port=6007
```

9.3. TensorBoard Log Cleaning Feature in SpotOptim

Automatic cleaning of old TensorBoard log directories with the `tensorboard_clean` parameter.

9.3.1. Basic Usage

```
import numpy as np
from spotoptim import SpotOptim

def sphere(X):
    """Simple sphere function"""
    return np.sum(X**2, axis=1)

# Remove old logs and create new log directory
optimizer = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=20,
    n_initial=10,
    tensorboard_log=True,
    tensorboard_clean=True, # Removes all subdirectories in 'runs'
    verbose=True,
    seed=42
)

result = optimizer.optimize()
print(f"Best value: {result.fun:.6f}")
print(f"Logs saved to: runs/{optimizer.tensorboard_path}")
```

```
Removed old TensorBoard logs: runs/spotoptim_20251220_100904
Cleaned 1 old TensorBoard log directory
TensorBoard logging enabled: runs/spotoptim_20251220_100904
Initial best: f(x) = 2.420807
Iteration 1: New best f(x) = 2.390307
Iteration 2: New best f(x) = 1.813139
Iteration 3: New best f(x) = 0.009007
Iteration 4: New best f(x) = 0.001900
Iteration 5: New best f(x) = 0.000973
Iteration 6: New best f(x) = 0.000001
Iteration 7: New best f(x) = 0.000001
Iteration 8: f(x) = 0.000001
Iteration 9: New best f(x) = 0.000001
Iteration 10: f(x) = 0.000001
TensorBoard writer closed. View logs with: tensorboard --logdir=runs/spotoptim_20251220_100904
Best value: 0.000001
Logs saved to: runs/runs/spotoptim_20251220_100904
```

9.3.2. Use Cases

tensorboard_log	tensorboard_clean	Behavior
True	True	Clean old logs, create new log directory
True	False	Preserve old logs, create new log directory
False	True	Clean old logs, no new logging
False	False	No logging, no cleaning (default)

9.3.3. Implementation Details

9.3.3.1. Cleaning Method

```
def _clean_tensorboard_logs(self) -> None:
    """Clean old TensorBoard log directories from the runs folder."""
    if self.tensorboard_clean:
        runs_dir = "runs"
        if os.path.exists(runs_dir) and os.path.isdir(runs_dir):
            # Get all subdirectories in runs
            subdirs = [
                os.path.join(runs_dir, d)
                for d in os.listdir(runs_dir)
                if os.path.isdir(os.path.join(runs_dir, d))
            ]

            # Remove each subdirectory
            for subdir in subdirs:
                try:
                    shutil.rmtree(subdir)
                    if self.verbose:
                        print(f"Removed old TensorBoard logs: {subdir}")
                except Exception as e:
                    if self.verbose:
                        print(f"Warning: Could not remove {subdir}: {e}")
```

9.3.4. Execution Flow

1. User creates SpotOptim instance with tensorboard_clean=True
2. During initialization, _clean_tensorboard_logs() is called
3. Method checks if 'runs' directory exists
4. Removes all subdirectories (but preserves files)

9. TensorBoard Logging in SpotOptim

5. If `tensorboard_log=True`, a new log directory is created
6. Optimization proceeds normally

9.4. Safety Features

- Only removes **directories**, not files in ‘runs’ folder
- Handles missing ‘runs’ directory gracefully
- Error handling for permission issues
- Verbose output shows what’s being removed
- Default is `False` to prevent accidental deletion

Warning

- Setting `tensorboard_clean=True` permanently deletes all subdirectories in the ‘runs’ folder. Make sure to save important logs elsewhere before enabling this feature.

9.5. Tensorboard Demo Scripts

Run the comprehensive TensorBoard demo:

```
python demo_tensorboard.py
```

This demonstrates:

- Deterministic optimization (Rosenbrock function)
- Noisy optimization with repeated evaluations
- OCBA for intelligent re-evaluation

Run the log cleaning demo:

```
python demo_tensorboard_clean.py
```

This demonstrates:

- Creating multiple log directories
- Preserving old logs (default behavior)
- Cleaning old logs automatically
- Cleaning without creating new logs

9.6. Troubleshooting

Q: TensorBoard shows “No dashboards are active” A: Make sure you’ve run an optimization with `tensorboard_log=True` first.

Q: Can’t see my latest run A: Refresh TensorBoard (click the reload button in the upper right).

Q: How do I stop TensorBoard? A: Press Ctrl+C in the terminal where TensorBoard is running.

Q: Logs taking up too much space? A: Use `tensorboard_clean=True` to automatically remove old logs, or manually delete old run directories.

Q: How do I remove all old logs at once? A: Set `tensorboard_clean=True` when creating your optimizer. This will remove all subdirectories in the `runs` folder.

9.7. Related Parameters

- `tensorboard_log` (bool): Enable/disable logging (default: False)
- `tensorboard_path` (str): Custom log directory (default: auto-generated with timestamp)
- `tensorboard_clean` (bool): Remove old logs from ‘runs’ directory before starting (default: False)
- `verbose` (bool): Print progress to console (default: False)
- `var_name` (list): Custom names for variables (used in TensorBoard labels)

9.8. Performance Notes

TensorBoard logging has minimal overhead:

- Event files are efficiently buffered and written
- Writer is properly closed after optimization completes

9.9. Jupyter Notebook

Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

Part IV.

Saving and Loading Models

10. Save and Load in SpotOptim

SpotOptim provides comprehensive save and load functionality for serializing optimization configurations and results. This enables distributed workflows where experiments are defined locally, executed remotely, and analyzed back on the local machine.

10.1. Key Concepts

10.1.1. Experiments vs Results

SpotOptim distinguishes between two types of saved data:

- **Experiment** (*_exp.pkl): Configuration only, excluding the objective function and results. Used to transfer optimization setup to remote machines.
- **Result** (*_res.pkl): Complete optimization state including configuration, all evaluations, and results. Used to save and analyze completed optimizations.

10.1.2. What Gets Saved

Component	Experiment	Result
Configuration (bounds, parameters)		
Objective function		
Evaluations (X, y)		
Best solution		
Surrogate model	Excluded*	
TensorBoard writer		

*Surrogate model is excluded from experiments and automatically recreated when loaded.

10.2. Quick Start

10.2.1. Basic Save and Load

```
import numpy as np
from spotoptim import SpotOptim

def sphere(X):
    """Simple sphere function"""
    return np.sum(X**2, axis=1)

# Create and configure optimizer
optimizer = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=20,
    n_initial=10,
    seed=42
)

# Run optimization
result = optimizer.optimize()
print(f"Best value: {result.fun:.6f}")

# Save complete results
optimizer.save_result(prefix="sphere_opt")
# Creates: sphere_opt_res.pkl

# Later: load and analyze results
loaded_opt = SpotOptim.load_result("sphere_opt_res.pkl")
print(f"Loaded best value: {loaded_opt.best_y_:.6f}")
print(f"Total evaluations: {loaded_opt.counter}")
```

```
Best value: 0.000001
Experiment saved to sphere_opt_res.pkl
Result saved to sphere_opt_res.pkl
Loaded result from sphere_opt_res.pkl
Loaded best value: 0.000001
Total evaluations: 20
```

10.3. Distributed Workflow

The save/load functionality enables a powerful workflow for distributed optimization:

10.3.1. Step 1: Define Experiment Locally

```
import numpy as np
from spotoptim import SpotOptim

# Define a placeholder function (will be replaced on remote machine)
def placeholder(X):
    """Placeholder function - will be replaced remotely"""
    return np.sum(X**2, axis=1)

# Define configuration locally
optimizer = SpotOptim(
    fun=placeholder, # Temporary function
    bounds=[(-10, 10), (-10, 10), (-10, 10)],
    max_iter=20,
    n_initial=10,
    seed=42,
    verbose=True
)

# Save experiment configuration
optimizer.save_experiment(prefix="remote_job_001")
# Creates: remote_job_001_exp.pkl

print("Experiment saved. Transfer remote_job_001_exp.pkl to remote machine.")
print("The objective function will be replaced on the remote machine.")
```

```
TensorBoard logging disabled
Experiment saved to remote_job_001_exp.pkl
Experiment saved. Transfer remote_job_001_exp.pkl to remote machine.
The objective function will be replaced on the remote machine.
```

10.3.2. Step 2: Execute on Remote Machine

10. Save and Load in SpotOptim

```
from spotoptim import SpotOptim
import numpy as np

# Define objective function on remote machine
def expensive_function(X):
    """Expensive simulation or computation"""
    # Your expensive computation here
    return np.sum(X**2, axis=1) + 0.1 * np.sum(np.sin(10 * X), axis=1)

# Load experiment configuration
optimizer = SpotOptim.load_experiment("remote_job_001_exp.pkl")
print("Experiment loaded successfully")

# Attach objective function (must be done after loading)
optimizer.fun = expensive_function

# Run optimization
result = optimizer.optimize()
print(f"Optimization complete. Best value: {result.fun:.6f}")

# Save results
optimizer.save_result(prefix="remote_job_001")
# Creates: remote_job_001_res.pkl

print("Results saved. Transfer remote_job_001_res.pkl back to local machine.")
```

```
Loaded experiment from remote_job_001_exp.pkl
Experiment loaded successfully
Initial best: f(x) = 46.587622
Iteration 1: f(x) = 57.963676
Iteration 2: f(x) = 49.624579
Iteration 3: New best f(x) = 45.184938
Iteration 4: New best f(x) = 41.587740
Iteration 5: New best f(x) = 3.846258
Iteration 6: New best f(x) = 1.484084
Iteration 7: New best f(x) = 0.167743
Iteration 8: f(x) = 0.210425
Iteration 9: f(x) = 0.364727
Iteration 10: f(x) = 0.271020
Optimization complete. Best value: 0.167743
Experiment saved to remote_job_001_res.pkl
Result saved to remote_job_001_res.pkl
Results saved. Transfer remote_job_001_res.pkl back to local machine.
```

10.3.3. Step 3: Analyze Results Locally

```

from spotoptim import SpotOptim
import matplotlib.pyplot as plt

# Load results from remote execution
optimizer = SpotOptim.load_result("remote_job_001_res.pkl")

# Access all optimization data
print(f"Best value found: {optimizer.best_y_:.6f}")
print(f"Best point: {optimizer.best_x_}")
print(f"Total evaluations: {optimizer.counter}")
print(f"Number of iterations: {optimizer.n_iter_}")

# Analyze convergence
plt.figure(figsize=(10, 6))
plt.plot(optimizer.y_, 'o-', alpha=0.6, label='Evaluations')
plt.plot(range(len(optimizer.y_)),
         [optimizer.y_[:i+1].min() for i in range(len(optimizer.y_))],
         'r-', linewidth=2, label='Best so far')
plt.xlabel('Iteration')
plt.ylabel('Objective Value')
plt.title('Optimization Progress')
plt.legend()
plt.grid(True)
plt.show()

# Access all evaluated points
print(f"\nAll evaluated points shape: {optimizer.X_.shape}")
print(f"All objective values shape: {optimizer.y_.shape}")

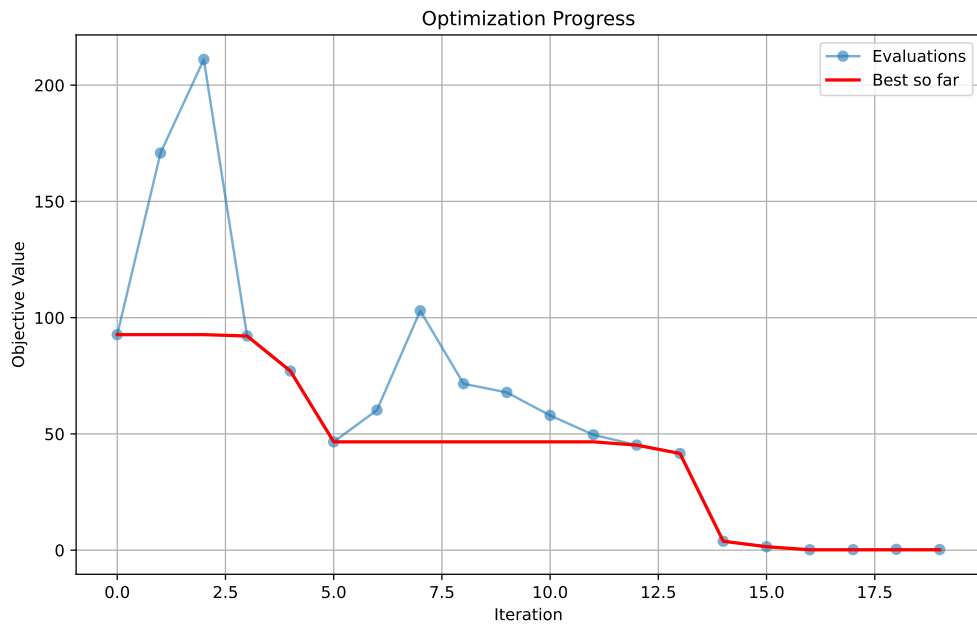
```

```

Loaded result from remote_job_001_res.pkl
Best value found: 0.167743
Best point: [ 0.17516921  0.34614876 -0.05559165]
Total evaluations: 20
Number of iterations: 10

```

10. Save and Load in SpotOptim



All evaluated points shape: (20, 3)
All objective values shape: (20,)

10.4. Advanced Usage

10.4.1. Custom Filenames and Paths

```
import os
from spotoptim import SpotOptim
import numpy as np

def objective(X):
    return np.sum(X**2, axis=1)

optimizer = SpotOptim(
    fun=objective,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=30,
    seed=42
```

```

)

# Save with custom filename
optimizer.save_experiment(
    filename="custom_name.pkl",
    verbosity=1
)

# Save to specific directory
os.makedirs("experiments/batch_001", exist_ok=True)
optimizer.save_experiment(
    prefix="exp_001",
    path="experiments/batch_001",
    verbosity=1
)
# Creates: experiments/batch_001/exp_001_exp.pkl

```

Experiment saved to custom_name.pkl

Experiment saved to experiments/batch_001/exp_001_exp.pkl

10.4.2. Overwrite Protection

```

from spotoptim import SpotOptim
import numpy as np

def sphere(X):
    return np.sum(X**2, axis=1)

optimizer = SpotOptim(fun=sphere, bounds=[(-5, 5), (-5, 5)], max_iter=20)
result = optimizer.optimize()

# First save
optimizer.save_result(prefix="my_result")

# Try to save again - raises FileExistsError by default
try:
    optimizer.save_result(prefix="my_result")
except FileExistsError as e:
    print(f"File already exists: {e}")

# Explicitly allow overwriting

```

10. Save and Load in SpotOptim

```
optimizer.save_result(prefix="my_result", overwrite=True)
print("File overwritten successfully")
```

```
Experiment saved to my_result_res.pkl
Result saved to my_result_res.pkl
Experiment saved to my_result_res.pkl
Result saved to my_result_res.pkl
Experiment saved to my_result_res.pkl
Result saved to my_result_res.pkl
File overwritten successfully
```

10.4.3. Loading and Continuing Optimization

```
from spotoptim import SpotOptim
import numpy as np

def objective(X):
    return np.sum(X**2, axis=1)

# Initial optimization
opt1 = SpotOptim(
    fun=objective,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=20,
    seed=42
)
result1 = opt1.optimize()
opt1.save_result(prefix="checkpoint")

print(f"Initial optimization: {result1.nfev} evaluations, best={result1.fun:.6f}")

# Load and continue
opt2 = SpotOptim.load_result("checkpoint_res.pkl")
opt2.fun = objective # Re-attach function
opt2.max_iter = 25 # Increase budget

# Continue optimization
result2 = opt2.optimize()
print(f"After continuation: {result2.nfev} evaluations, best={result2.fun:.6f}")
```

```
Experiment saved to checkpoint_res.pkl
```


Result saved to checkpoint_res.pkl
 Initial optimization: 20 evaluations, best=0.000001
 Loaded result from checkpoint_res.pkl
 After continuation: 25 evaluations, best=0.000002

10.5. Working with Noisy Functions

Save and load preserves noise statistics for reproducible analysis:

```
import numpy as np
from spotoptim import SpotOptim

def noisy_objective(X):
    """Objective with measurement noise"""
    true_value = np.sum(X**2, axis=1)
    noise = np.random.normal(0, 0.1, X.shape[0])
    return true_value + noise

# Optimize noisy function with repeated evaluations
optimizer = SpotOptim(
    fun=noisy_objective,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=20,
    n_initial=10,
    repeats_initial=3,    # Repeat initial points
    repeats_surrogate=2,  # Repeat surrogate points
    seed=42,
    verbose=True
)

result = optimizer.optimize()

# Save results (includes noise statistics)
optimizer.save_result(prefix="noisy_opt")

# Load and analyze noise statistics
loaded_opt = SpotOptim.load_result("noisy_opt_res.pkl")

print(f"Noise handling enabled: {loaded_opt.noise}")
print(f"Best mean value: {loaded_opt.best_y_:.6f}")

if loaded_opt.mean_y is not None:
```

10. Save and Load in SpotOptim

```
print(f"Mean values available: {len(loader.mean_y)}")
print(f"Variance values available: {len(loader.var_y)}")
```

```
TensorBoard logging disabled
Initial best: f(x) = 2.286218, mean best: f(x) = 2.373803
Experiment saved to noisy_opt_res.pkl
Result saved to noisy_opt_res.pkl
Loaded result from noisy_opt_res.pkl
Noise handling enabled: True
Best mean value: 2.286218
Mean values available: 10
Variance values available: 10
```

10.6. Working with Different Variable Types

Save and load preserves variable type information:

```
import numpy as np
from spotoptim import SpotOptim

def mixed_objective(X):
    """Objective with mixed variable types"""
    return np.sum(X**2, axis=1)

# Create optimizer with mixed variable types
optimizer = SpotOptim(
    fun=mixed_objective,
    bounds=[(-5, 5), (-5, 5), (-5, 5), (-5, 5)],
    var_type=["float", "int", "factor", "float"],
    var_name=["continuous", "integer", "categorical", "another_cont"],
    max_iter=20,
    n_initial=10,
    seed=42
)

result = optimizer.optimize()

# Save results
optimizer.save_result(prefix="mixed_vars")

# Load results
loaded_opt = SpotOptim.load_result("mixed_vars_res.pkl")
```

```

print("Variable types preserved:")
print(f"  var_type: {loaded_opt.var_type}")
print(f"  var_name: {loaded_opt.var_name}")

# Verify integer variables are still integers
print(f"\nInteger variable (dim 1) values:")
print(loaded_opt.X_[:5, 1]) # Should be integers

```

```

Experiment saved to mixed_vars_res.pkl
Result saved to mixed_vars_res.pkl
Loaded result from mixed_vars_res.pkl
Variable types preserved:
  var_type: ['float', 'int', 'factor', 'float']
  var_name: ['continuous', 'integer', 'categorical', 'another_cont']

Integer variable (dim 1) values:
[ 3. -2. -0. -3.  4.]

```

10.7. Best Practices

10.7.1. 1. Always Re-attach the Objective Function

After loading an experiment, you **must** re-attach the objective function:

```

# Load experiment
optimizer = SpotOptim.load_experiment("experiment_exp.pkl")

# REQUIRED: Re-attach function
optimizer.fun = your_objective_function

# Now you can optimize
result = optimizer.optimize()

```

10.7.2. 2. Use Meaningful Prefixes

Organize your experiments with descriptive prefixes:

10. Save and Load in SpotOptim

```
# Good practice: descriptive prefixes
optimizer.save_experiment(prefix="sphere_d10_seed42")
optimizer.save_experiment(prefix="rosenbrock_n100_lhs")
optimizer.save_result(prefix="final_run_2024_11_15")

# Avoid: generic names
optimizer.save_experiment(prefix="exp1") # Not descriptive
optimizer.save_result(prefix="result")   # Hard to track
```

10.7.3. 3. Save Experiments Before Remote Execution

```
# Define locally
optimizer = SpotOptim(bounds=bounds, max_iter=20, seed=42)
optimizer.save_experiment(prefix="remote_job")

# Transfer file to remote machine
# Execute remotely
# Transfer results back
# Analyze locally
```

10.7.4. 4. Version Your Experiments

```
import datetime

# Add timestamp to prefix
timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
prefix = f"experiment_{timestamp}"

optimizer.save_experiment(prefix=prefix)
# Creates: experiment_20241115_143022_exp.pkl
```

10.7.5. 5. Handle File Paths Robustly

```
import os

# Create directory structure
exp_dir = "experiments/batch_001"
os.makedirs(exp_dir, exist_ok=True)
```

```
# Save with full path
optimizer.save_experiment(
    prefix="exp_001",
    path=exp_dir
)

# Load with full path
exp_file = os.path.join(exp_dir, "exp_001_exp.pkl")
loaded_opt = SpotOptim.load_experiment(exp_file)
```

10.8. Complete Example: Multi-Machine Workflow

Here's a complete example demonstrating the entire workflow:

10.8.1. Local Machine (Setup)

```
# setup_experiment.py
import numpy as np
from spotoptim import SpotOptim
import os

# Placeholder function for experiment setup
def placeholder_func(X):
    return np.sum(X**2, axis=1)

# Create experiments directory
os.makedirs("experiments", exist_ok=True)

# Define multiple experiments
experiments = [
    {"seed": 42, "max_iter": 20, "prefix": "exp_seed42"},
    {"seed": 123, "max_iter": 20, "prefix": "exp_seed123"},
    {"seed": 999, "max_iter": 20, "prefix": "exp_seed999"},
]

for exp_config in experiments:
    optimizer = SpotOptim(
        fun=placeholder_func, # Placeholder - will be replaced remotely
        bounds=[(-10, 10), (-10, 10), (-10, 10)],
        max_iter=exp_config["max_iter"],
```

10. Save and Load in SpotOptim

```
        n_initial=10,
        seed=exp_config["seed"],
        verbose=False
    )

    optimizer.save_experiment(
        prefix=exp_config["prefix"],
        path="experiments"
    )

    print(f"Created: experiments/{exp_config['prefix']}_exp.pkl")

print("\nAll experiments created. Transfer 'experiments' folder to remote machine.")
```

```
Experiment saved to experiments/exp_seed42_exp.pkl
Created: experiments/exp_seed42_exp.pkl
Experiment saved to experiments/exp_seed123_exp.pkl
Created: experiments/exp_seed123_exp.pkl
Experiment saved to experiments/exp_seed999_exp.pkl
Created: experiments/exp_seed999_exp.pkl
```

All experiments created. Transfer 'experiments' folder to remote machine.

10.8.2. Remote Machine (Execution)

```
# run_experiments.py
import numpy as np
from spotoptim import SpotOptim
import os
import glob

def complex_objective(X):
    """Complex multimodal objective function"""
    term1 = np.sum(X**2, axis=1)
    term2 = 10 * np.sum(np.cos(2 * np.pi * X), axis=1)
    term3 = 0.1 * np.sum(np.sin(5 * np.pi * X), axis=1)
    return term1 - term2 + term3

# Find all experiment files
exp_files = glob.glob("experiments/*_exp.pkl")
print(f"Found {len(exp_files)} experiments to run")
```

10.8. Complete Example: Multi-Machine Workflow

```
# Run each experiment
for exp_file in exp_files:
    print(f"\nProcessing: {exp_file}")

    # Load experiment
    optimizer = SpotOptim.load_experiment(exp_file)

    # Attach objective
    optimizer.fun = complex_objective

    # Run optimization
    result = optimizer.optimize()
    print(f"  Best value: {result.fun:.6f}")

    # Save result (same prefix, different suffix)
    prefix = os.path.basename(exp_file).replace("_exp.pkl", "")
    optimizer.save_result(
        prefix=prefix,
        path="experiments"
    )
    print(f"  Saved: experiments/{prefix}_res.pkl")

print("\nAll experiments completed. Transfer results back to local machine.")
```

Found 3 experiments to run

Processing: experiments/exp_seed999_exp.pkl
Loaded experiment from experiments/exp_seed999_exp.pkl
 Best value: -9.121049
Experiment saved to experiments/exp_seed999_res.pkl
Result saved to experiments/exp_seed999_res.pkl
 Saved: experiments/exp_seed999_res.pkl

Processing: experiments/exp_seed42_exp.pkl
Loaded experiment from experiments/exp_seed42_exp.pkl
 Best value: -5.807748
Experiment saved to experiments/exp_seed42_res.pkl
Result saved to experiments/exp_seed42_res.pkl
 Saved: experiments/exp_seed42_res.pkl

Processing: experiments/exp_seed123_exp.pkl
Loaded experiment from experiments/exp_seed123_exp.pkl
 Best value: -8.050942
Experiment saved to experiments/exp_seed123_res.pkl

10. Save and Load in SpotOptim

```
Result saved to experiments/exp_seed123_res.pkl  
Saved: experiments/exp_seed123_res.pkl
```

All experiments completed. Transfer results back to local machine.

10.8.3. Local Machine (Analysis)

```
# analyze_results.py  
import numpy as np  
from spotoptim import SpotOptim  
import glob  
import matplotlib.pyplot as plt  
  
# Find all result files  
result_files = glob.glob("experiments/*_res.pkl")  
print(f"Found {len(result_files)} results to analyze")  
  
# Load and compare results  
results = []  
for res_file in result_files:  
    opt = SpotOptim.load_result(res_file)  
    results.append({  
        "file": res_file,  
        "best_value": opt.best_y_,  
        "best_point": opt.best_x_,  
        "n_evals": opt.counter,  
        "seed": opt.seed  
    })  
    print(f"{res_file}: best={opt.best_y_:.6f}, evals={opt.counter}")  
  
# Find best overall result  
best = min(results, key=lambda x: x["best_value"])  
print(f"\nBest result:")  
print(f"  File: {best['file']}")  
print(f"  Value: {best['best_value']:.6f}")  
print(f"  Point: {best['best_point']}")  
print(f"  Seed: {best['seed']}")  
  
# Plot convergence comparison  
plt.figure(figsize=(12, 6))  
  
for res_file in result_files:  
    opt = SpotOptim.load_result(res_file)
```


10.8. Complete Example: Multi-Machine Workflow

```
seed = opt.seed
cummin = [opt.y_[i+1].min() for i in range(len(opt.y_))]
plt.plot(cummin, label=f"Seed {seed}", linewidth=2, alpha=0.7)

plt.xlabel("Iteration", fontsize=12)
plt.ylabel("Best Value Found", fontsize=12)
plt.title("Optimization Progress Comparison", fontsize=14)
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig("experiments/convergence_comparison.png", dpi=150)
print("\nConvergence plot saved to: experiments/convergence_comparison.png")
```

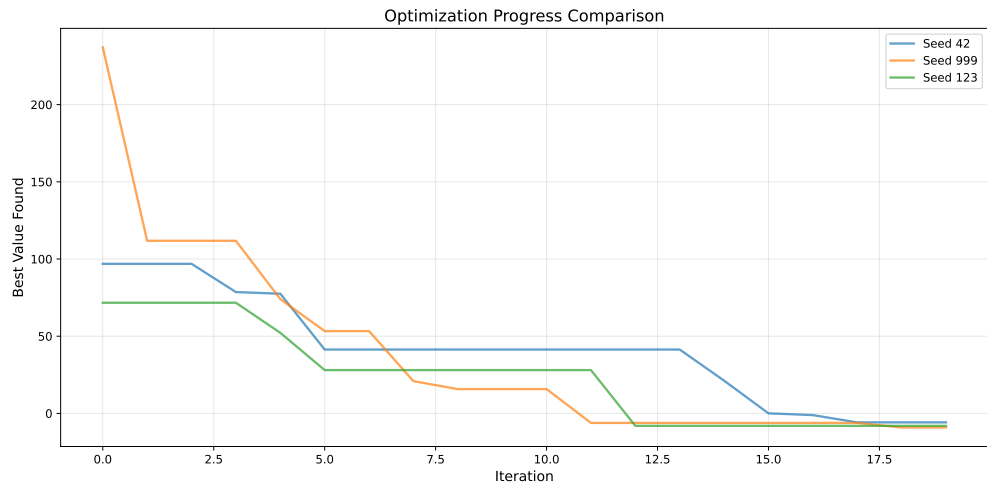
```
Found 3 results to analyze
Loaded result from experiments/exp_seed42_res.pkl
experiments/exp_seed42_res.pkl: best=-5.807748, evals=20
Loaded result from experiments/exp_seed999_res.pkl
experiments/exp_seed999_res.pkl: best=-9.121049, evals=20
Loaded result from experiments/exp_seed123_res.pkl
experiments/exp_seed123_res.pkl: best=-8.050942, evals=20
```

Best result:

```
File: experiments/exp_seed999_res.pkl
Value: -9.121049
Point: [-1.05626884 -0.90533225  0.363701  ]
Seed: 999
Loaded result from experiments/exp_seed42_res.pkl
Loaded result from experiments/exp_seed999_res.pkl
Loaded result from experiments/exp_seed123_res.pkl
```

Convergence plot saved to: experiments/convergence_comparison.png

10. Save and Load in SpotOptim



10.9. Technical Details

10.9.1. Serialization Method

SpotOptim uses Python's built-in `pickle` module for serialization. This provides:

- **Standard library:** No additional dependencies required
- **Compatibility:** Works with numpy arrays, sklearn models, scipy functions
- **Performance:** Efficient serialization of large datasets

10.9.2. Component Reinitialization

When loading experiments, certain components are automatically recreated:

- **Surrogate model:** Gaussian Process with default kernel
- **LHS sampler:** Latin Hypercube Sampler with original seed

This ensures loaded experiments can continue optimization without manual configuration.

10.9.3. Excluded Components

Some components cannot be pickled and are automatically excluded:

- **Objective function (fun):** Lambda functions and local functions cannot be reliably pickled

- **TensorBoard writer** (tb_writer): File handles cannot be serialized
- **Surrogate model** (experiments only): Recreated on load for experiments

10.9.4. File Format

Files are saved using pickle's highest protocol:

```
with open(filename, "wb") as handle:
    pickle.dump(optimizer_state, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

10.10. Troubleshooting

10.10.1. Issue: “AttributeError: ‘SpotOptim’ object has no attribute ‘fun’”

Cause: Objective function not re-attached after loading experiment.

Solution: Always re-attach the function after loading:

```
opt = SpotOptim.load_experiment("exp.pkl")
opt.fun = your_objective_function # Add this line
result = opt.optimize()
```

10.10.2. Issue: “FileNotFoundError: Experiment file not found”

Cause: Incorrect file path or file doesn't exist.

Solution: Check file path and ensure file exists:

```
import os

filename = "experiment_exp.pkl"
if os.path.exists(filename):
    opt = SpotOptim.load_experiment(filename)
else:
    print(f"File not found: {filename}")
```

10. Save and Load in SpotOptim

10.10.3. Issue: “FileExistsError: File already exists”

Cause: Attempting to save over an existing file without `overwrite=True`.

Solution: Either use a different prefix or enable overwriting:

```
# Option 1: Use different prefix
optimizer.save_result(prefix="my_result_v2")

# Option 2: Enable overwriting
optimizer.save_result(prefix="my_result", overwrite=True)
```

10.10.4. Issue: Results differ after loading

Cause: Random state not preserved or function behavior changed.

Solution: Ensure you’re using the same seed and function definition:

```
# When saving
optimizer = SpotOptim(..., seed=42) # Use fixed seed

# When loading and continuing
loaded_opt = SpotOptim.load_result("result_res.pkl")
loaded_opt.fun = same_objective_function # Same function definition
```

10.11. Jupyter Notebook

Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

11. Reproducibility in SpotOptim

11.1. Introduction

SpotOptim provides full support for reproducible optimization runs through the `seed` parameter. This is essential for:

- **Scientific research:** Ensuring experiments can be replicated
- **Debugging:** Reproducing specific optimization behaviors
- **Benchmarking:** Fair comparison between different configurations
- **Production:** Consistent results in deployed applications

When you specify a seed, SpotOptim guarantees that running the same optimization multiple times will produce identical results. Without a seed, each run explores the search space differently, which can be useful for robustness testing.

11.2. Basic Usage

11.2.1. Making Optimization Reproducible

To ensure reproducible results, simply specify the `seed` parameter when creating the optimizer:

```
import numpy as np
from spotoptim import SpotOptim

def sphere(X):
    """Simple sphere function:  $f(x) = \sum(x^2)$ """
    return np.sum(X**2, axis=1)

# Reproducible optimization
optimizer = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=30,
    n_initial=15,
```

11. Reproducibility in SpotOptim

```
seed=42, # This ensures reproducibility
verbose=True
)

result = optimizer.optimize()
print(f"Best solution: {result.x}")
print(f"Best value: {result.fun}")
```

```
TensorBoard logging disabled
Initial best: f(x) = 5.542803
Iteration 1: New best f(x) = 0.001070
Iteration 2: New best f(x) = 0.000089
Iteration 3: New best f(x) = 0.000066
Iteration 4: New best f(x) = 0.000036
Iteration 5: New best f(x) = 0.000001
Iteration 6: New best f(x) = 0.000000
Iteration 7: f(x) = 0.000000
Iteration 8: f(x) = 0.000000
Iteration 9: f(x) = 0.000000
Iteration 10: f(x) = 0.000000
Iteration 11: f(x) = 0.000000
Iteration 12: f(x) = 0.000000
Iteration 13: f(x) = 0.000000
Iteration 14: f(x) = 0.000000
Iteration 15: New best f(x) = 0.000000
Best solution: [3.31436760e-04 4.18312302e-05]
Best value: 1.1160017787260647e-07
```

Key Point: Running this code multiple times (even on different days or machines) will always produce the same result.

11.2.2. Running Independent Experiments

If you don't specify a seed, each optimization run will explore the search space differently:

```
# Non-reproducible: different results each time
optimizer = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=30,
    n_initial=15
```

```

    # No seed specified
)

result = optimizer.optimize()
# Results will vary between runs

```

This is useful when you want to: - Explore different regions of the search space - Test the robustness of your results - Run multiple independent optimization attempts

11.3. Practical Examples

11.3.1. Example 1: Comparing Different Configurations

When comparing different optimizer settings, use the same seed for fair comparison:

```

import numpy as np
from spotoptim import SpotOptim

def rosenbrock(X):
    """Rosenbrock function"""
    x = X[:, 0]
    y = X[:, 1]
    return (1 - x)**2 + 100 * (y - x**2)**2

# Configuration 1: More initial points
opt1 = SpotOptim(
    fun=rosenbrock,
    bounds=[(-2, 2), (-2, 2)],
    max_iter=50,
    n_initial=20,
    seed=42 # Same seed for fair comparison
)
result1 = opt1.optimize()

# Configuration 2: Fewer initial points, more iterations
opt2 = SpotOptim(
    fun=rosenbrock,
    bounds=[(-2, 2), (-2, 2)],
    max_iter=50,
    n_initial=10,
    seed=42 # Same seed
)

```

11. Reproducibility in SpotOptim

```
result2 = opt2.optimize()

print(f"Config 1 (more initial): {result1.fun:.6f}")
print(f"Config 2 (fewer initial): {result2.fun:.6f}")
```

```
Config 1 (more initial): 0.002494
Config 2 (fewer initial): 0.011538
```

11.3.2. Example 2: Reproducible Research Experiment

For scientific papers or reports, always use a fixed seed and document it:

```
import numpy as np
from spotoptim import SpotOptim

def rastrigin(X):
    """Rastrigin function (multimodal)"""
    A = 10
    n = X.shape[1]
    return A * n + np.sum(X**2 - A * np.cos(2 * np.pi * X), axis=1)

# Documented seed for reproducibility
RANDOM_SEED = 12345

optimizer = SpotOptim(
    fun=rastrigin,
    bounds=[(-5.12, 5.12), (-5.12, 5.12), (-5.12, 5.12)],
    max_iter=100,
    n_initial=30,
    seed=RANDOM_SEED,
    verbose=True
)

result = optimizer.optimize()

print(f"\nExperiment Results (seed={RANDOM_SEED}):")
print(f"Best solution: {result.x}")
print(f"Best value: {result.fun}")
print(f"Iterations: {result.nit}")
print(f"Function evaluations: {result.nfev}")

# These results can now be cited in a paper
```


11.3. Practical Examples

```
TensorBoard logging disabled
Initial best: f(x) = 20.392774
Iteration 1: f(x) = 31.367736
Iteration 2: f(x) = 30.822371
Iteration 3: f(x) = 28.761963
Iteration 4: f(x) = 25.355390
Iteration 5: f(x) = 22.087398
Iteration 6: f(x) = 23.609642
Iteration 7: f(x) = 27.024927
Iteration 8: f(x) = 23.154774
Iteration 9: New best f(x) = 8.927550
Iteration 10: New best f(x) = 8.899624
Iteration 11: f(x) = 19.831800
Iteration 12: f(x) = 19.323148
Iteration 13: New best f(x) = 8.692652
Iteration 14: New best f(x) = 8.244681
Iteration 15: f(x) = 8.404445
Iteration 16: f(x) = 17.268172
Iteration 17: f(x) = 17.250262
Iteration 18: f(x) = 16.989215
Iteration 19: f(x) = 22.061741
Iteration 20: New best f(x) = 8.227940
Iteration 21: New best f(x) = 2.732909
Iteration 22: New best f(x) = 2.236454
Iteration 23: New best f(x) = 2.001414
Iteration 24: f(x) = 2.001821
Iteration 25: f(x) = 2.002706
Iteration 26: f(x) = 24.677487
Iteration 27: f(x) = 25.445950
Iteration 28: f(x) = 22.323886
Iteration 29: New best f(x) = 1.990146
Iteration 30: New best f(x) = 1.989946
Iteration 31: f(x) = 30.157363
Iteration 32: f(x) = 13.980500
Iteration 33: f(x) = 26.257931
Iteration 34: f(x) = 26.086756
Iteration 35: f(x) = 16.955502
Iteration 36: f(x) = 86.774141
Iteration 37: f(x) = 42.005362
Iteration 38: New best f(x) = 1.989929
Iteration 39: f(x) = 1.989930
Iteration 40: New best f(x) = 1.989929
Iteration 41: f(x) = 69.717061
Iteration 42: New best f(x) = 1.989929
Iteration 43: f(x) = 66.932192
```

11. Reproducibility in SpotOptim

```
Iteration 44: f(x) = 50.905256
Iteration 45: f(x) = 1.989930
Iteration 46: f(x) = 46.800517
Iteration 47: f(x) = 67.429406
Iteration 48: f(x) = 60.775651
Iteration 49: f(x) = 1.989930
Iteration 50: f(x) = 1.989929
Iteration 51: f(x) = 1.989930
Iteration 52: f(x) = 1.989929
Iteration 53: f(x) = 1.989930
Iteration 54: f(x) = 1.989929
Iteration 55: f(x) = 1.989931
Iteration 56: f(x) = 1.989930
Iteration 57: f(x) = 1.989929
Iteration 58: f(x) = 16.986975
Iteration 59: f(x) = 14.520119
Iteration 60: f(x) = 86.774141
Iteration 61: f(x) = 1.989931
Iteration 62: f(x) = 76.830975
Iteration 63: f(x) = 59.126187
Iteration 64: f(x) = 1.989929
Iteration 65: f(x) = 13.047213
Iteration 66: New best f(x) = 1.989929
Iteration 67: New best f(x) = 1.989928
Iteration 68: f(x) = 1.989931
Iteration 69: f(x) = 1.989930
Iteration 70: f(x) = 1.989928
```

```
Experiment Results (seed=12345):
Best solution: [ 9.94876607e-01 -9.94866368e-01  1.89299459e-04]
Best value: 1.9899282459113223
Iterations: 70
Function evaluations: 100
```

11.3.3. Example 3: Multiple Independent Runs

To test robustness, run the same optimization with different seeds:

```
import numpy as np
from spotoptim import SpotOptim

def ackley(X):
    """Ackley function"""
    a = 20
```

```

b = 0.2
c = 2 * np.pi
n = X.shape[1]

sum_sq = np.sum(X**2, axis=1)
sum_cos = np.sum(np.cos(c * X), axis=1)

return -a * np.exp(-b * np.sqrt(sum_sq / n)) - np.exp(sum_cos / n) + a + np.e

# Run 5 independent optimizations
results = []
seeds = [42, 123, 456, 789, 1011]

for seed in seeds:
    optimizer = SpotOptim(
        fun=ackley,
        bounds=[(-5, 5), (-5, 5)],
        max_iter=40,
        n_initial=20,
        seed=seed,
        verbose=False
    )
    result = optimizer.optimize()
    results.append(result.fun)
    print(f"Run with seed {seed:4d}: f(x) = {result.fun:.6f}")

# Analyze robustness
print(f"\nBest result: {min(results):.6f}")
print(f"Worst result: {max(results):.6f}")
print(f"Mean: {np.mean(results):.6f}")
print(f"Std dev: {np.std(results):.6f}")

```

```

Run with seed 42: f(x) = 0.000907
Run with seed 123: f(x) = 0.001394
Run with seed 456: f(x) = 0.001941
Run with seed 789: f(x) = 0.000616
Run with seed 1011: f(x) = 0.003029

```

```

Best result: 0.000616
Worst result: 0.003029
Mean: 0.001578
Std dev: 0.000854

```

11.3.4. Example 4: Reproducible Initial Design

The seed ensures that even the initial design points are reproducible:

```
import numpy as np
from spotoptim import SpotOptim

def simple_quadratic(X):
    return np.sum((X - 1)**2, axis=1)

# Create two optimizers with same seed
opt1 = SpotOptim(
    fun=simple_quadratic,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=25,
    n_initial=10,
    seed=999
)

opt2 = SpotOptim(
    fun=simple_quadratic,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=25,
    n_initial=10,
    seed=999 # Same seed
)

# Run both optimizations
result1 = opt1.optimize()
result2 = opt2.optimize()

# Verify identical results
print("Initial design points are identical:",
      np.allclose(opt1.X_[:10], opt2.X_[:10]))
print("All evaluated points are identical:",
      np.allclose(opt1.X_, opt2.X_))
print("All function values are identical:",
      np.allclose(opt1.y_, opt2.y_))
print("Best solutions are identical:",
      np.allclose(result1.x, result2.x))
```

```
Initial design points are identical: True
All evaluated points are identical: True
All function values are identical: True
Best solutions are identical: True
```

11.3.5. Example 5: Custom Initial Design with Seed

Even when providing a custom initial design, the seed ensures reproducible subsequent iterations:

```
import numpy as np
from spotoptim import SpotOptim

def beale(X):
    """Beale function"""
    x = X[:, 0]
    y = X[:, 1]
    term1 = (1.5 - x + x * y)**2
    term2 = (2.25 - x + x * y**2)**2
    term3 = (2.625 - x + x * y**3)**2
    return term1 + term2 + term3

# Custom initial design (e.g., from previous knowledge)
X_start = np.array([
    [0.0, 0.0],
    [1.0, 1.0],
    [2.0, 2.0],
    [-1.0, -1.0]
])

# Run twice with same seed and initial design
opt1 = SpotOptim(
    fun=beale,
    bounds=[(-4.5, 4.5), (-4.5, 4.5)],
    max_iter=30,
    n_initial=10,
    seed=777
)
result1 = opt1.optimize(X0=X_start)

opt2 = SpotOptim(
    fun=beale,
    bounds=[(-4.5, 4.5), (-4.5, 4.5)],
    max_iter=30,
    n_initial=10,
    seed=777 # Same seed
)
result2 = opt2.optimize(X0=X_start)
```

11. Reproducibility in SpotOptim

```
print("Results are identical:", np.allclose(result1.x, result2.x))
print(f"Best value: {result1.fun:.6f}")
```

```
Results are identical: True
Best value: 3.201102
```

11.4. Advanced Topics

11.4.1. Seed and Noisy Functions

When optimizing noisy functions with repeated evaluations, the seed ensures reproducible noise:

```
import numpy as np
from spotoptim import SpotOptim

def noisy_sphere(X):
    """Sphere function with Gaussian noise"""
    base = np.sum(X**2, axis=1)
    noise = np.random.normal(0, 0.1, size=base.shape)
    return base + noise

optimizer = SpotOptim(
    fun=noisy_sphere,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=40,
    n_initial=20,
    repeats_initial=3, # 3 evaluations per point
    repeats_surrogate=2,
    seed=42 # Ensures same noise pattern
)

result = optimizer.optimize()
print(f"Best mean value: {optimizer.min_mean_y:.6f}")
print(f"Variance at best: {optimizer.min_var_y:.6f}")
```

```
Best mean value: 0.165136
Variance at best: 0.000940
```

Important: With the same seed, even the noise will be identical across runs!

11.4.2. Different Seeds for Different Exploration

Use different seeds to explore different regions systematically:

```
import numpy as np
from spotoptim import SpotOptim

def griewank(X):
    """Griewank function"""
    sum_sq = np.sum(X**2 / 4000, axis=1)
    prod_cos = np.prod(np.cos(X / np.sqrt(np.arange(1, X.shape[1] + 1))), axis=1)
    return sum_sq - prod_cos + 1

# Systematic exploration with different seeds
best_overall = float('inf')
best_seed = None

for seed in range(10, 20): # Seeds 10-19
    optimizer = SpotOptim(
        fun=griewank,
        bounds=[(-600, 600), (-600, 600)],
        max_iter=50,
        n_initial=25,
        seed=seed
    )
    result = optimizer.optimize()

    if result.fun < best_overall:
        best_overall = result.fun
        best_seed = seed

    print(f"Seed {seed}: f(x) = {result.fun:.6f}")

print(f"\nBest result with seed {best_seed}: {best_overall:.6f}")
```

```
Seed 10: f(x) = 4.597121
Seed 11: f(x) = 1.946733
Seed 12: f(x) = 6.030579
Seed 13: f(x) = 0.851170
Seed 14: f(x) = 0.154832
Seed 15: f(x) = 0.044427
Seed 16: f(x) = 8.343419
Seed 17: f(x) = 0.547492
Seed 18: f(x) = 2.121439
```

11. Reproducibility in SpotOptim

Seed 19: $f(x) = 0.257388$

Best result with seed 15: 0.044427

11.5. Best Practices

11.5.1. 1. Always Use Seeds for Production Code

```
# Good: Reproducible
optimizer = SpotOptim(fun=objective, bounds=bounds, seed=42)

# Risky: Non-reproducible
optimizer = SpotOptim(fun=objective, bounds=bounds)
```

11.5.2. 2. Document Your Seeds

```
# Configuration for experiment reported in Section 4.2
EXPERIMENT_SEED = 2024
MAX_ITERATIONS = 100

optimizer = SpotOptim(
    fun=my_objective,
    bounds=my_bounds,
    max_iter=MAX_ITERATIONS,
    seed=EXPERIMENT_SEED
)
```

11.5.3. 3. Use Different Seeds for Different Experiments

```
# Different experiments should use different seeds
BASELINE_SEED = 100
EXPERIMENT_A_SEED = 200
EXPERIMENT_B_SEED = 300
```


11.5.4. 4. Test Robustness Across Multiple Seeds

```
# Run same optimization with multiple seeds
for seed in [42, 123, 456, 789, 1011]:
    optimizer = SpotOptim(fun=objective, bounds=bounds, seed=seed)
    result = optimizer.optimize()
# Analyze results
```

11.6. What the Seed Controls

The `seed` parameter ensures reproducibility by controlling:

1. **Initial Design Generation:** Latin Hypercube Sampling produces the same initial points
2. **Surrogate Model:** Gaussian Process random initialization is identical
3. **Acquisition Optimization:** Differential evolution explores the same candidates
4. **Random Sampling:** Any random exploration uses the same random numbers

This guarantees that the entire optimization pipeline is deterministic and reproducible.

11.7. Common Questions

Q: Can I use `seed=0`?

A: Yes, any integer (including 0) is a valid seed.

Q: Will different Python versions give the same results?

A: Generally yes, but minor numerical differences may occur due to underlying library changes. Use the same environment for exact reproducibility.

Q: Does the seed affect the objective function?

A: No, the seed only affects SpotOptim's internal random processes. If your objective function has its own randomness, you'll need to control that separately.

Q: How do I choose a good seed value?

A: Any integer works. Common choices are 42, 123, or dates (e.g., 20241112). What matters is consistency, not the specific value.

11.8. Summary

- Use `seed` parameter for reproducible optimization
- Same seed → identical results (every time)
- No seed → different results (random exploration)
- Essential for research, debugging, and production
- Document your seeds for transparency
- Test robustness with multiple different seeds

11.9. Jupyter Notebook

Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

Part V.

Surrogate Handling

12. Surrogate Model Selection in SpotOptim

Note

- This section demonstrates how to select and configure different surrogate models in SpotOptim
- We compare various surrogate options:
 - **Gaussian Process** with different kernels (Matern, RBF, Rational Quadratic)
 - **SpotOptim Kriging** model
 - **Random Forest** regressor
 - **XGBoost** regressor
 - **Support Vector Regression** (SVR)
 - **Gradient Boosting** regressor
- All methods are evaluated on the Aircraft Wing Weight Example (AWWE) function
- We visualize the fitted surrogates and compare optimization performance

12.1. Introduction

Surrogate models are the heart of Bayesian optimization. SpotOptim supports any scikit-learn compatible regressor, allowing you to choose the surrogate that best fits your problem characteristics:

- **Gaussian Processes:** Provide uncertainty estimates, smooth interpolation, good for continuous functions. Support Expected Improvement (EI) acquisition.
- **Kriging:** Similar to GP but with customizable correlation functions. Supports EI acquisition.
- **Random Forests:** Robust to noise, handle discontinuities. Don't provide uncertainty, so use `acquisition='y'` (greedy).
- **XGBoost:** Excellent for high-dimensional problems, fast training and prediction. Use `acquisition='y'`.

12. Surrogate Model Selection in SpotOptim

- **SVR:** Good for high-dimensional problems with smooth structure. Use `acquisition='y'`.
- **Gradient Boosting:** Strong performance on structured problems. Use `acquisition='y'`.

! Acquisition Functions and Uncertainty

Models that provide uncertainty estimates (Gaussian Process, Kriging) work with all acquisition functions: 'ei' (Expected Improvement), 'pi' (Probability of Improvement), and 'y' (greedy).

Tree-based and other models (Random Forest, XGBoost, SVR, Gradient Boosting) don't provide uncertainty estimates by default, so they should use `acquisition='y'` for greedy optimization. SpotOptim automatically handles this gracefully.

12.2. Setup and Imports

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from spotoptim import SpotOptim
from spotoptim.surrogate import Kriging
import time
import warnings
warnings.filterwarnings('ignore')

# Scikit-learn models
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import (
    Matern, RBF, ConstantKernel, WhiteKernel, RationalQuadratic
)
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.svm import SVR
```

- XGBoost (if available)

```
try:
    import xgboost as xgb
    XGBOOST_AVAILABLE = True
except ImportError:
    XGBOOST_AVAILABLE = False
```

```
print("XGBoost not available. Install with: pip install xgboost")

print("Libraries imported successfully!")
```

Libraries imported successfully!

12.3. The AWE Objective Function

We use the Aircraft Wing Weight Example function, which models the weight of an unpainted light aircraft wing. The function accepts inputs in the unit cube $[0, 1]^9$ and returns the wing weight in pounds.

```
def wingwt(x):
    """
    Aircraft Wing Weight function.

    Args:
        x: array-like of 9 values in [0,1]
           [Sw, Wfw, A, L, q, l, Rtc, Nz, Wdg]

    Returns:
        Wing weight (scalar)
    """
    # Ensure x is a 2D array for batch evaluation
    x = np.atleast_2d(x)

    # Transform from unit cube to natural scales
    Sw = x[:, 0] * (200 - 150) + 150      # Wing area (ft2)
    Wfw = x[:, 1] * (300 - 220) + 220      # Fuel weight (lb)
    A = x[:, 2] * (10 - 6) + 6             # Aspect ratio
    L = (x[:, 3] * (10 - (-10)) - 10) * np.pi/180 # Sweep angle (rad)
    q = x[:, 4] * (45 - 16) + 16           # Dynamic pressure (lb/ft2)
    l = x[:, 5] * (1 - 0.5) + 0.5          # Taper ratio
    Rtc = x[:, 6] * (0.18 - 0.08) + 0.08   # Root thickness/chord
    Nz = x[:, 7] * (6 - 2.5) + 2.5          # Ultimate load factor
    Wdg = x[:, 8] * (2500 - 1700) + 1700    # Design gross weight (lb)

    # Calculate weight on natural scale
    W = 0.036 * Sw**0.758 * Wfw**0.0035 * (A/np.cos(L)**2)**0.6 * q**0.006
    W = W * l**0.04 * (100*Rtc/np.cos(L))**(-0.3) * (Nz*Wdg)**(0.49)

    return W.ravel()
```

12. Surrogate Model Selection in SpotOptim

```
# Problem setup
bounds = [(0, 1)] * 9
param_names = ['Sw', 'Wfw', 'A', 'L', 'q', 'l', 'Rtc', 'Nz', 'Wdg']
max_iter = 30
n_initial = 10
seed = 42

print(f"Problem dimension: {len(bounds)}")
print(f"Optimization budget: {max_iter} evaluations")
```

Problem dimension: 9
Optimization budget: 30 evaluations

12.4. 1. Default Surrogate: Gaussian Process with Matern Kernel

SpotOptim's default surrogate is a Gaussian Process with a Matern kernel ($\nu=2.5$), which provides twice-differentiable sample paths and good performance for most optimization problems.

```
print("=" * 80)
print("1. DEFAULT: Gaussian Process with Matern  $\nu=2.5$  Kernel")
print("=" * 80)

start_time = time.time()

# Default GP (no surrogate specified)
optimizer_default = SpotOptim(
    fun=wingwt,
    bounds=bounds,
    max_iter=max_iter,
    n_initial=n_initial,
    var_name=param_names,
    acquisition='ei',
    seed=seed,
    verbose=False
)

result_default = optimizer_default.optimize()
time_default = time.time() - start_time
```


12.4. 1. Default Surrogate: Gaussian Process with Matern Kernel

```
print(f"\nResults:")
print(f"  Best weight: {result_default.fun:.4f} lb")
print(f"  Function evaluations: {result_default.nfev}")
print(f"  Time: {time_default:.2f}s")
print(f"  Success: {result_default.success}")

# Store for comparison
results_comparison = [{
    'Surrogate': 'GP Matern =2.5 (Default)',
    'Best Weight': result_default.fun,
    'Evaluations': result_default.nfev,
    'Time (s)': time_default,
    'Success': result_default.success
}]
```

```
=====
1. DEFAULT: Gaussian Process with Matern =2.5 Kernel
=====
```

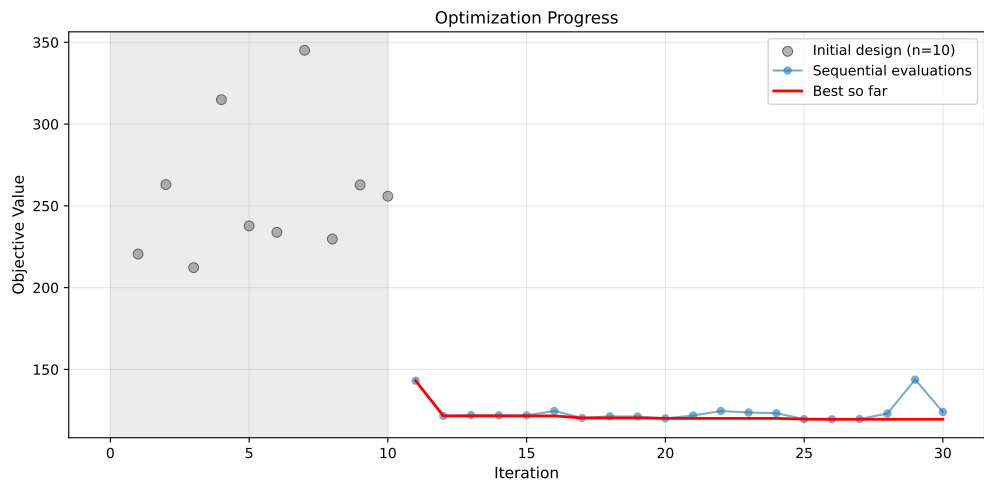
Results:

```
Best weight: 119.5041 lb
Function evaluations: 30
Time: 41.05s
Success: True
```

12.4.1. Visualization: Default Surrogate

```
# Plot convergence
optimizer_default.plot_progress(log_y=False, figsize=(10, 5))
```

12. Surrogate Model Selection in SpotOptim



```
# Plot most important hyperparameters
optimizer_default.plot_important_hyperparameter_contour(max_imp=3)
plt.suptitle('Default GP Matern =2.5: Most Important Parameters', y=1.02)
plt.show()
```

Plotting surrogate contours for top 3 most important parameters:

A: importance = 19.62% (type: float)

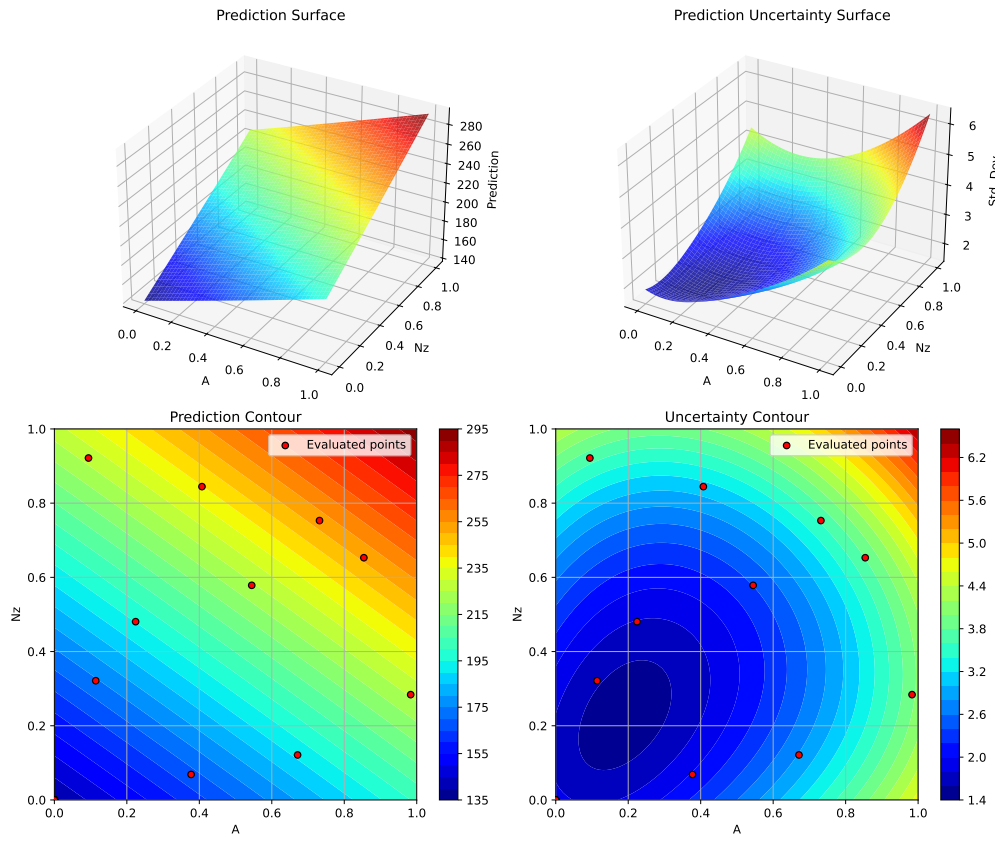
Nz: importance = 19.50% (type: float)

Rtc: importance = 19.29% (type: float)

Generating 3 surrogate plots...

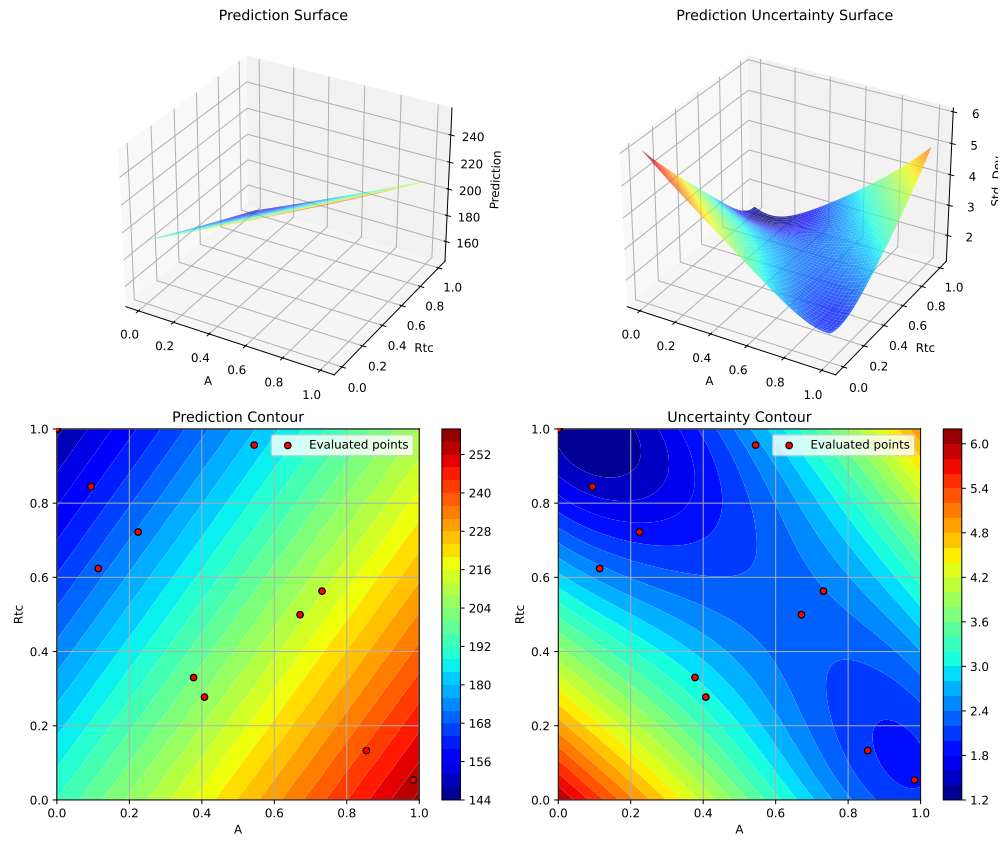
Plotting A vs Nz

12.4. 1. Default Surrogate: Gaussian Process with Matern Kernel



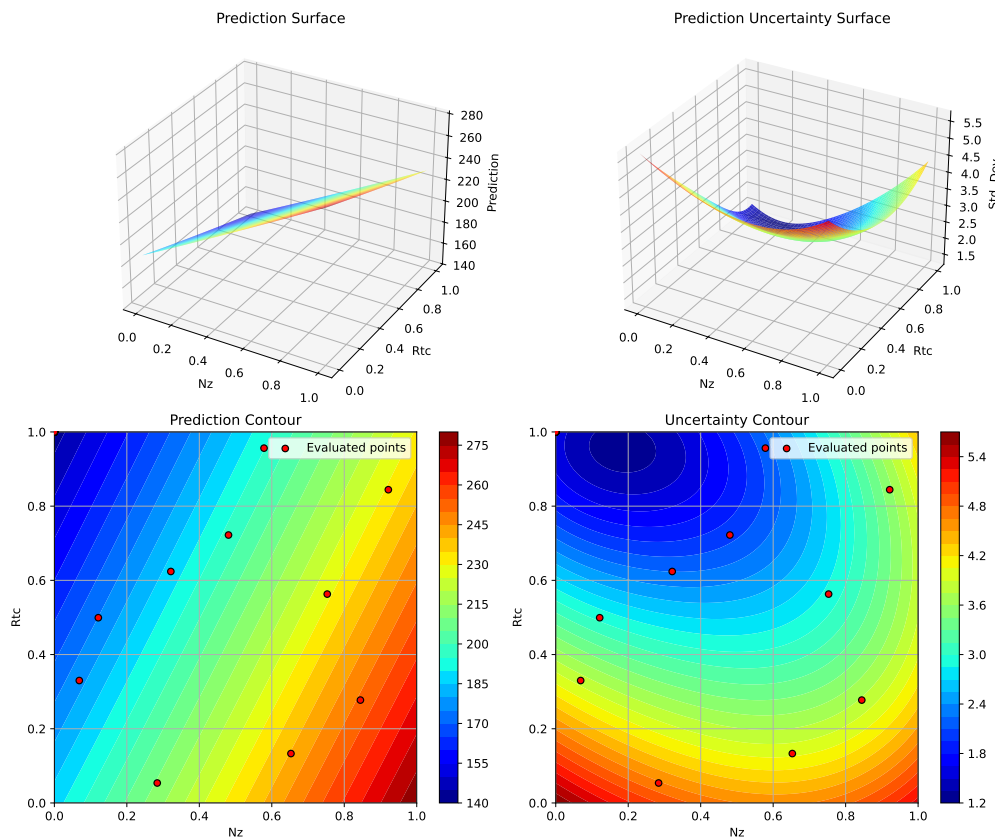
Plotting A vs Rtc

12. Surrogate Model Selection in SpotOptim



Plotting N_z vs R_{tc}

12.5. 2. Gaussian Process with RBF (Radial Basis Function) Kernel



<Figure size 1650x1050 with 0 Axes>

12.5. 2. Gaussian Process with RBF (Radial Basis Function) Kernel

The RBF kernel (also called squared exponential) produces infinitely differentiable sample paths, resulting in very smooth predictions.

```
print("=" * 80)
print("2. Gaussian Process with RBF Kernel")
print("=" * 80)

start_time = time.time()
```

12. Surrogate Model Selection in SpotOptim

```
# Configure GP with RBF kernel
kernel_rbf = ConstantKernel(1.0, (1e-3, 1e3)) * RBF(
    length_scale=1.0,
    length_scale_bounds=(1e-2, 1e2)
)

gp_rbf = GaussianProcessRegressor(
    kernel=kernel_rbf,
    n_restarts_optimizer=10,
    normalize_y=True,
    random_state=seed
)

optimizer_rbf = SpotOptim(
    fun=wingwt,
    bounds=bounds,
    surrogate=gp_rbf,
    max_iter=max_iter,
    n_initial=n_initial,
    var_name=param_names,
    acquisition='ei',
    seed=seed,
    verbose=False
)

result_rbf = optimizer_rbf.optimize()
time_rbf = time.time() - start_time

print(f"\nResults:")
print(f"  Best weight: {result_rbf.fun:.4f} lb")
print(f"  Function evaluations: {result_rbf.nfev}")
print(f"  Time: {time_rbf:.2f}s")
print(f"  Success: {result_rbf.success}")

results_comparison.append({
    'Surrogate': 'GP RBF',
    'Best Weight': result_rbf.fun,
    'Evaluations': result_rbf.nfev,
    'Time (s)': time_rbf,
    'Success': result_rbf.success
})
```

2. Gaussian Process with RBF Kernel

12.5. 2. Gaussian Process with RBF (Radial Basis Function) Kernel

Results:

Best weight: 119.5037 lb

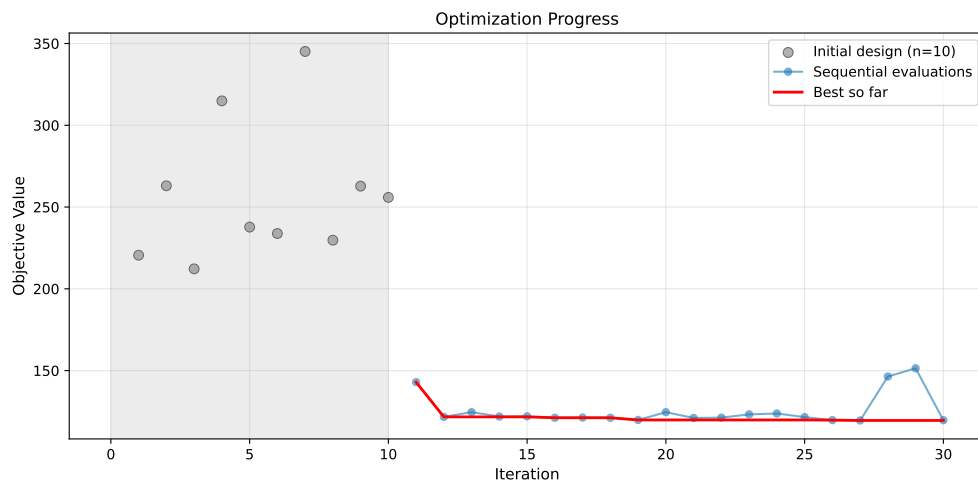
Function evaluations: 30

Time: 51.00s

Success: True

12.5.1. Visualization: RBF Kernel

```
optimizer_rbf.plot_progress(log_y=False, figsize=(10, 5))
```



```
optimizer_rbf.plot_important_hyperparameter_contour(max_imp=3)
plt.suptitle('GP RBF Kernel: Most Important Parameters', y=1.02)
plt.show()
```

Plotting surrogate contours for top 3 most important parameters:

A: importance = 19.40% (type: float)

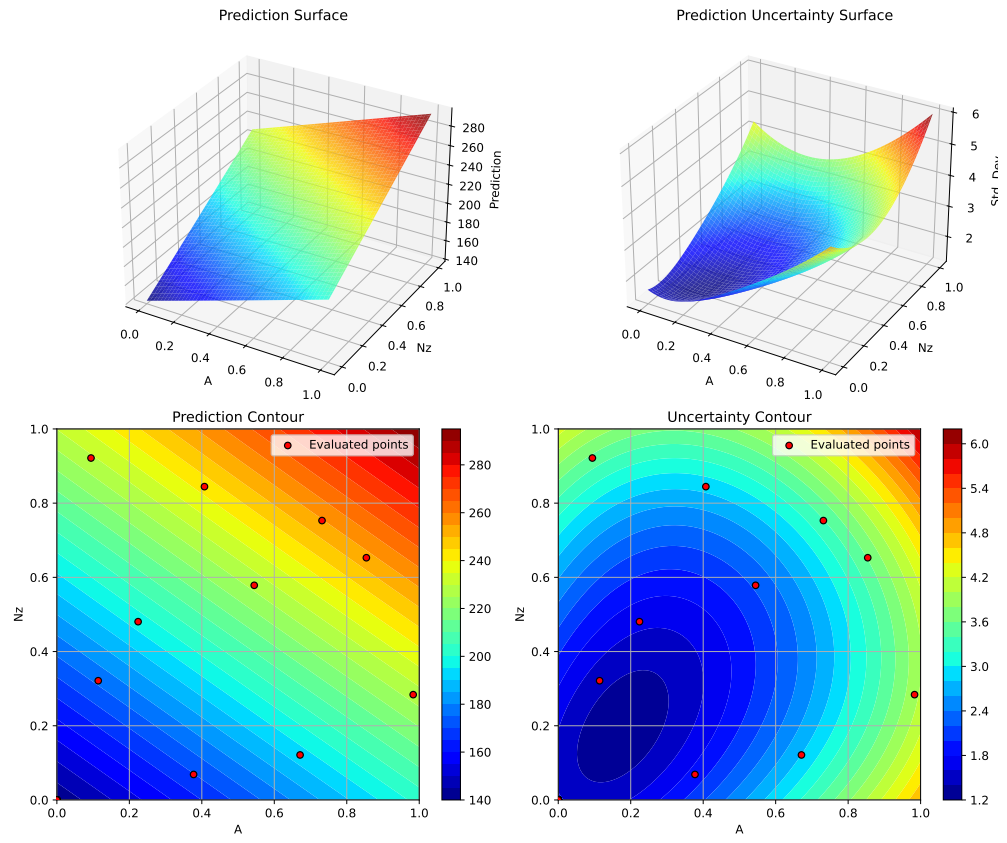
Nz: importance = 19.27% (type: float)

Rtc: importance = 19.06% (type: float)

Generating 3 surrogate plots...

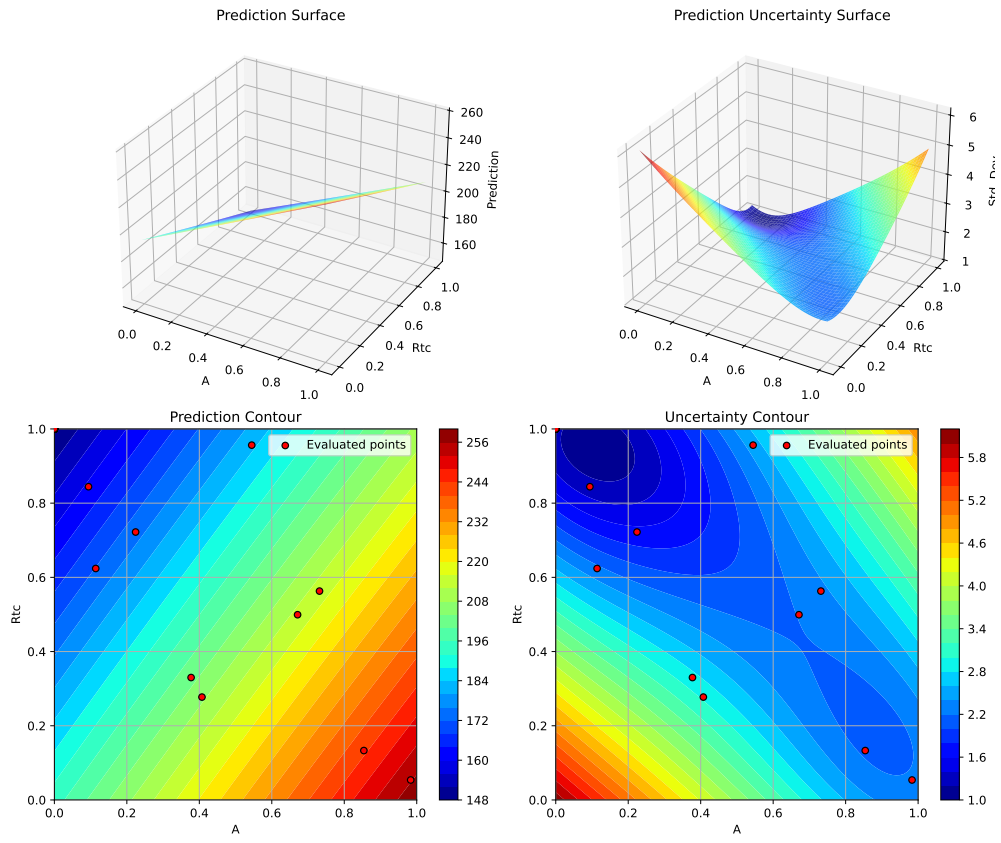
Plotting A vs Nz

12. Surrogate Model Selection in SpotOptim



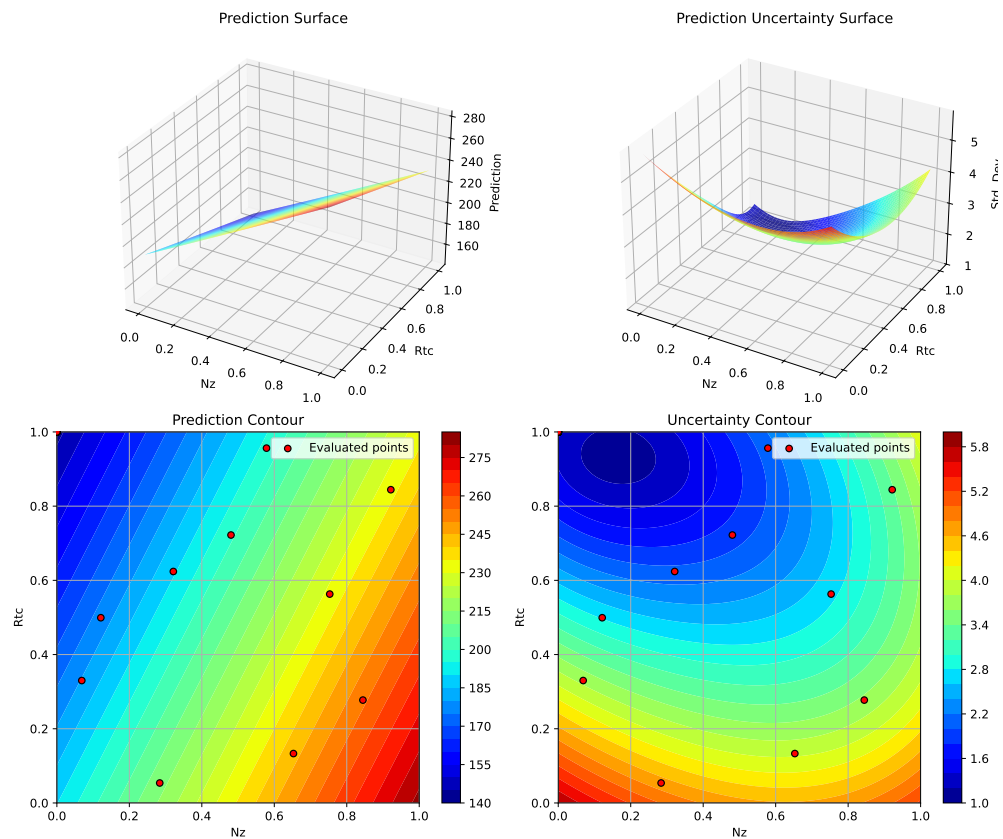
Plotting A vs R_{tc}

12.5. 2. Gaussian Process with RBF (Radial Basis Function) Kernel



Plotting N_z vs R_{tc}

12. Surrogate Model Selection in SpotOptim



<Figure size 1650x1050 with 0 Axes>

12.6. 3. Gaussian Process with Matern $\nu=1.5$ Kernel

The Matern $\nu=1.5$ kernel produces once-differentiable sample paths, allowing for more flexible (less smooth) fits than $\nu=2.5$.

```
print("=" * 80)
print("3. Gaussian Process with Matern  $\nu=1.5$  Kernel")
print("=" * 80)

start_time = time.time()

# Configure GP with Matern  $\nu=1.5$ 
kernel_matern15 = ConstantKernel(1.0, (1e-3, 1e3)) * Matern(
```

```

    length_scale=1.0,
    length_scale_bounds=(1e-2, 1e2),
    nu=1.5
)

gp_matern15 = GaussianProcessRegressor(
    kernel=kernel_matern15,
    n_restarts_optimizer=10,
    normalize_y=True,
    random_state=seed
)

optimizer_matern15 = SpotOptim(
    fun=wingwt,
    bounds=bounds,
    surrogate=gp_matern15,
    max_iter=max_iter,
    n_initial=n_initial,
    var_name=param_names,
    acquisition='ei',
    seed=seed,
    verbose=False
)

result_matern15 = optimizer_matern15.optimize()
time_matern15 = time.time() - start_time

print(f"\nResults:")
print(f"  Best weight: {result_matern15.fun:.4f} lb")
print(f"  Function evaluations: {result_matern15.nfev}")
print(f"  Time: {time_matern15:.2f}s")
print(f"  Success: {result_matern15.success}")

results_comparison.append({
    'Surrogate': 'GP Matern  $\nu=1.5$ ',
    'Best Weight': result_matern15.fun,
    'Evaluations': result_matern15.nfev,
    'Time (s)': time_matern15,
    'Success': result_matern15.success
})

```

```

=====
3. Gaussian Process with Matern  $\nu=1.5$  Kernel
=====

```

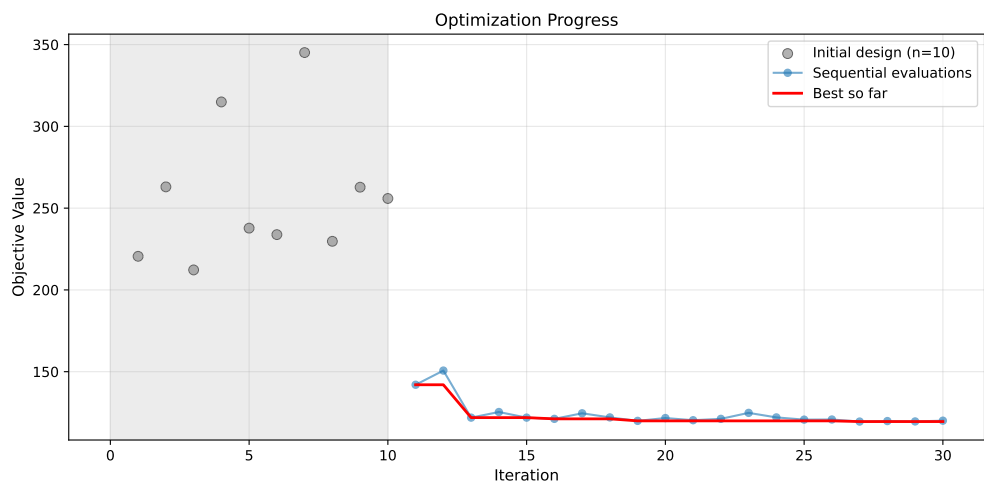
12. Surrogate Model Selection in SpotOptim

Results:

Best weight: 119.5056 lb
Function evaluations: 30
Time: 39.28s
Success: True

12.6.1. Visualization: Matern $\nu=1.5$ Kernel

```
optimizer_matern15.plot_progress(log_y=False, figsize=(10, 5))
```



```
optimizer_matern15.plot_important_hyperparameter_contour(max_imp=3)  
plt.suptitle('GP Matern  $\nu=1.5$  Kernel: Most Important Parameters', y=1.02)  
plt.show()
```

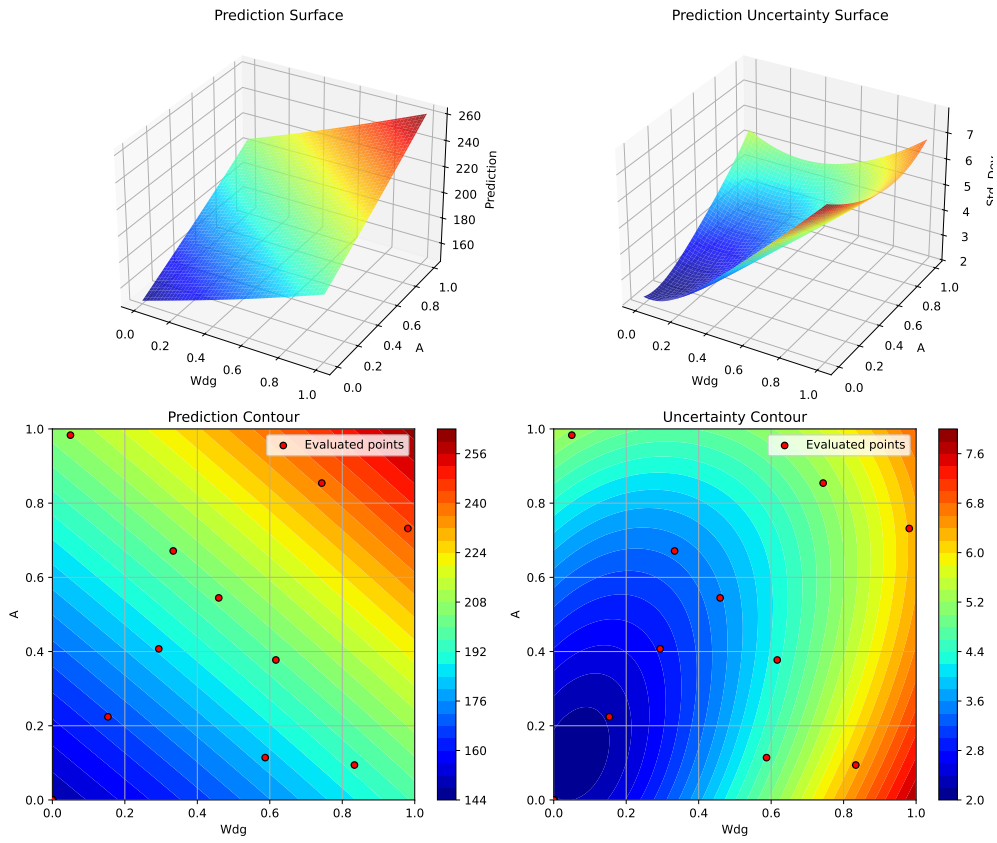
Plotting surrogate contours for top 3 most important parameters:

Wdg: importance = 18.62% (type: float)
A: importance = 18.41% (type: float)
Nz: importance = 18.30% (type: float)

Generating 3 surrogate plots...

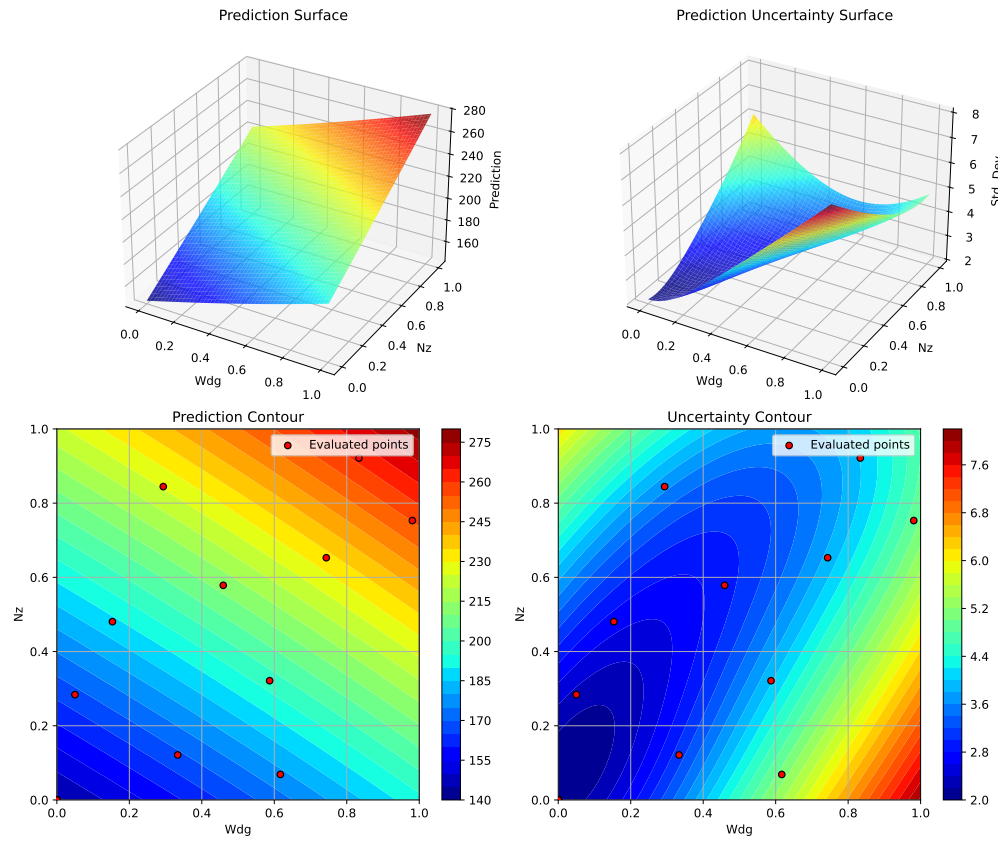
Plotting Wdg vs A

12.6. 3. Gaussian Process with Matern $\nu=1.5$ Kernel



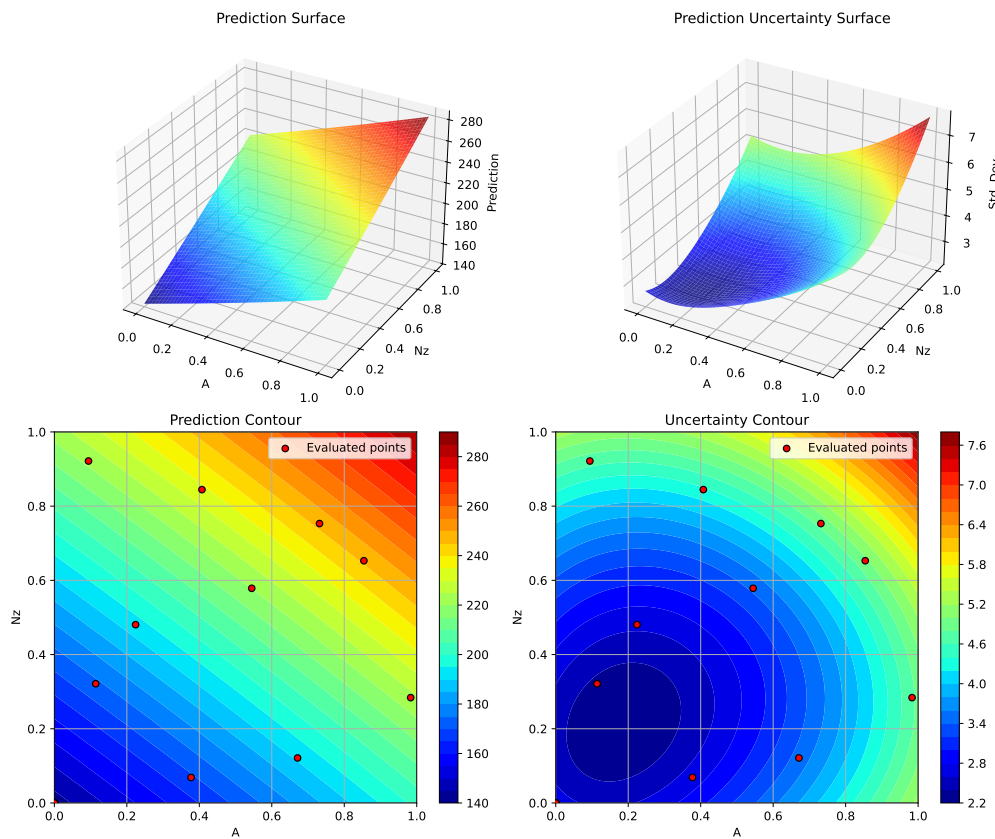
Plotting Wdg vs Nz

12. Surrogate Model Selection in SpotOptim



Plotting A vs Nz

12.7. 4. Gaussian Process with Rational Quadratic Kernel



<Figure size 1650x1050 with 0 Axes>

12.7. 4. Gaussian Process with Rational Quadratic Kernel

The Rational Quadratic kernel is a scale mixture of RBF kernels with different length scales, providing more flexibility than a single RBF.

```
print("=" * 80)
print("4. Gaussian Process with Rational Quadratic Kernel")
print("=" * 80)

start_time = time.time()
```

12. Surrogate Model Selection in SpotOptim

```
# Configure GP with Rational Quadratic kernel
kernel_rq = ConstantKernel(1.0, (1e-3, 1e3)) * RationalQuadratic(
    length_scale=1.0,
    alpha=1.0,
    length_scale_bounds=(1e-2, 1e2),
    alpha_bounds=(1e-2, 1e2)
)

gp_rq = GaussianProcessRegressor(
    kernel=kernel_rq,
    n_restarts_optimizer=10,
    normalize_y=True,
    random_state=seed
)

optimizer_rq = SpotOptim(
    fun=wingwt,
    bounds=bounds,
    surrogate=gp_rq,
    max_iter=max_iter,
    n_initial=n_initial,
    var_name=param_names,
    acquisition='ei',
    seed=seed,
    verbose=False
)

result_rq = optimizer_rq.optimize()
time_rq = time.time() - start_time

print(f"\nResults:")
print(f"  Best weight: {result_rq.fun:.4f} lb")
print(f"  Function evaluations: {result_rq.nfev}")
print(f"  Time: {time_rq:.2f}s")
print(f"  Success: {result_rq.success}")

results_comparison.append({
    'Surrogate': 'GP Rational Quadratic',
    'Best Weight': result_rq.fun,
    'Evaluations': result_rq.nfev,
    'Time (s)': time_rq,
    'Success': result_rq.success
})
```

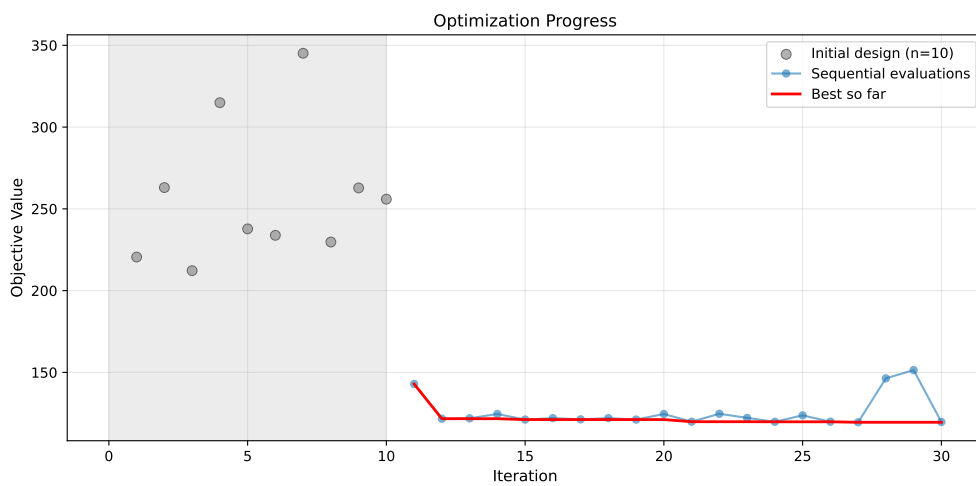
4. Gaussian Process with Rational Quadratic Kernel

Results:

Best weight: 119.5061 lb
 Function evaluations: 30
 Time: 46.18s
 Success: True

12.7.1. Visualization: Rational Quadratic Kernel

```
optimizer_rq.plot_progress(log_y=False, figsize=(10, 5))
```



```
optimizer_rq.plot_important_hyperparameter_contour(max_imp=3)
plt.suptitle('GP Rational Quadratic Kernel: Most Important Parameters', y=1.02)
plt.show()
```

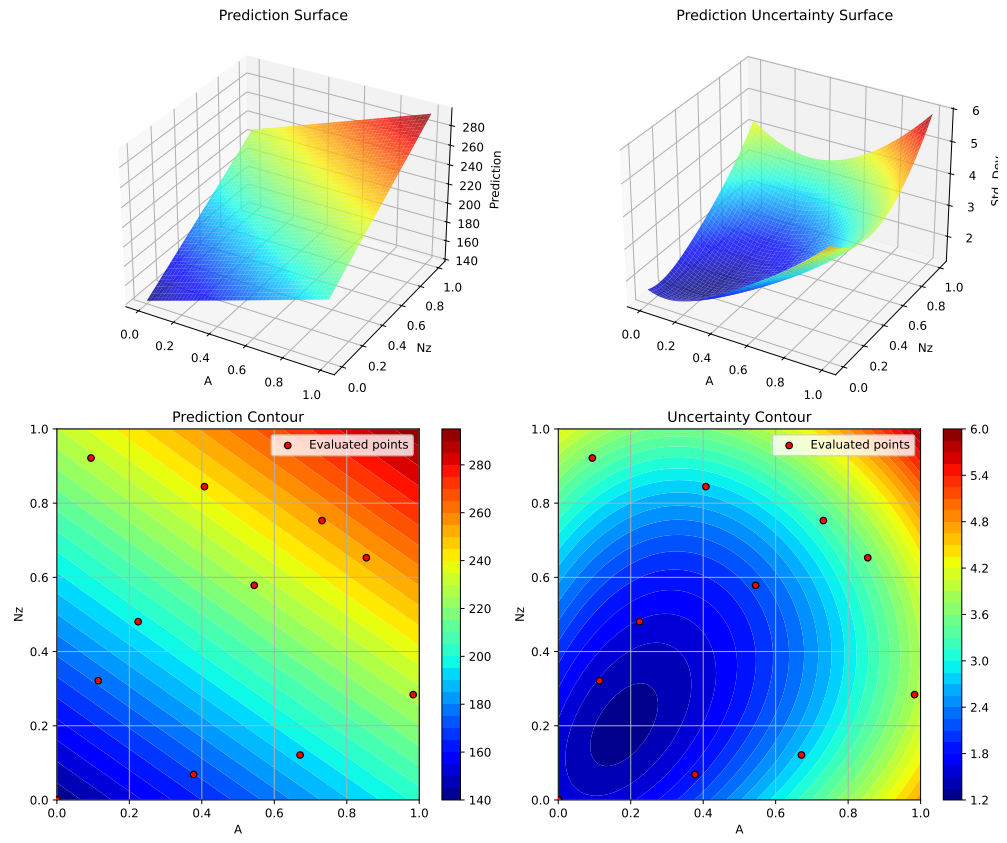
Plotting surrogate contours for top 3 most important parameters:

A: importance = 18.94% (type: float)
 Nz: importance = 18.82% (type: float)
 Rtc: importance = 18.61% (type: float)

Generating 3 surrogate plots...

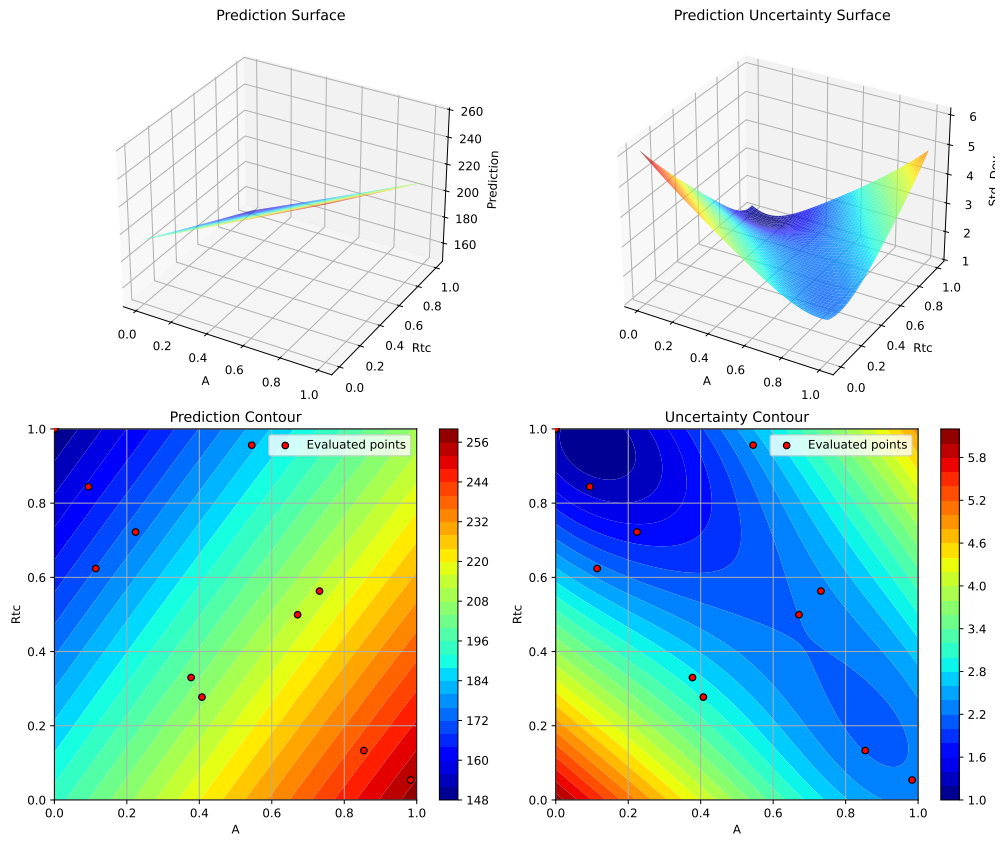
Plotting A vs Nz

12. Surrogate Model Selection in SpotOptim



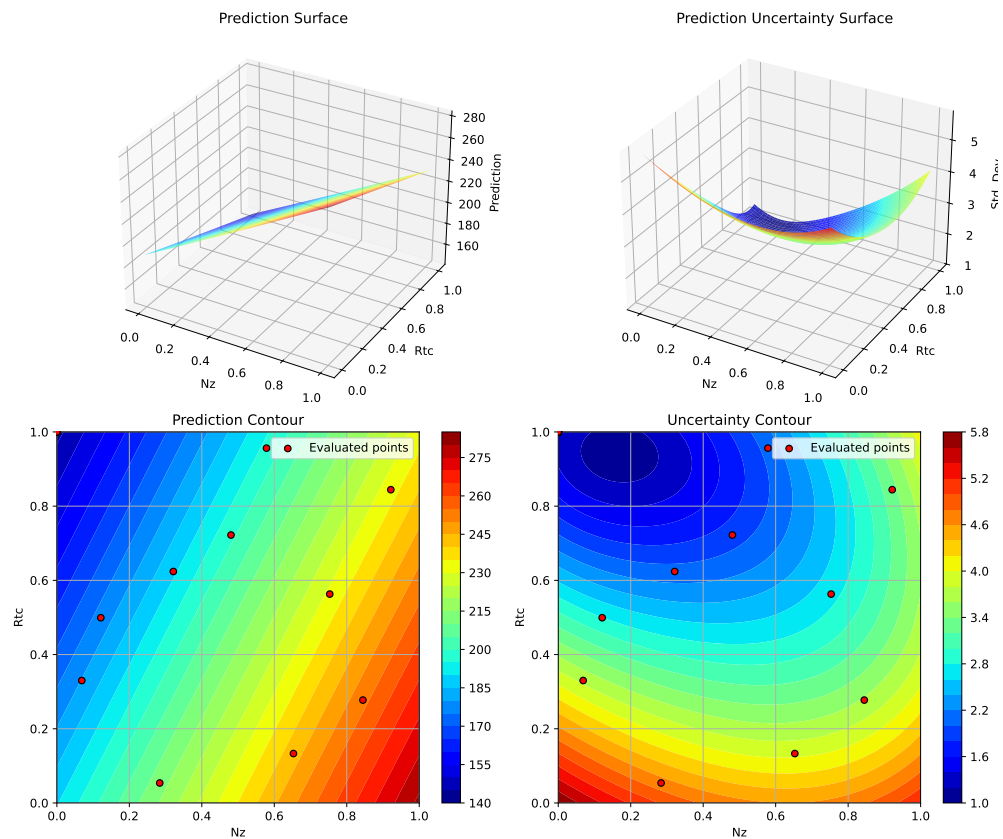
Plotting A vs R_{tc}

12.7. 4. Gaussian Process with Rational Quadratic Kernel



Plotting N_z vs R_{tc}

12. Surrogate Model Selection in SpotOptim



<Figure size 1650x1050 with 0 Axes>

12.8. 5. SpotOptim Kriging Model

SpotOptim includes its own Kriging implementation optimized for sequential design. It uses Gaussian correlation function and optimizes hyperparameters via differential evolution.

```
print("=" * 80)
print("5. SpotOptim Kriging Model")
print("=" * 80)

start_time = time.time()

# Configure Kriging model
```

```

kriging_model = Kriging(
    noise=1e-10,      # Regularization parameter
    kernel='gauss',   # Gaussian/RBF kernel
    n_theta=None,     # Auto: use number of dimensions
    min_theta=-3.0,   # Min log10(theta) bound
    max_theta=2.0,    # Max log10(theta) bound
    seed=seed
)

optimizer_kriging = SpotOptim(
    fun=wingwt,
    bounds=bounds,
    surrogate=kriging_model,
    max_iter=max_iter,
    n_initial=n_initial,
    var_name=param_names,
    acquisition='ei',
    seed=seed,
    verbose=False
)

result_kriging = optimizer_kriging.optimize()
time_kriging = time.time() - start_time

print(f"\nResults:")
print(f"  Best weight: {result_kriging.fun:.4f} lb")
print(f"  Function evaluations: {result_kriging.nfev}")
print(f"  Time: {time_kriging:.2f}s")
print(f"  Success: {result_kriging.success}")

results_comparison.append({
    'Surrogate': 'SpotOptim Kriging',
    'Best Weight': result_kriging.fun,
    'Evaluations': result_kriging.nfev,
    'Time (s)': time_kriging,
    'Success': result_kriging.success
})

```

5. SpotOptim Kriging Model

Results:

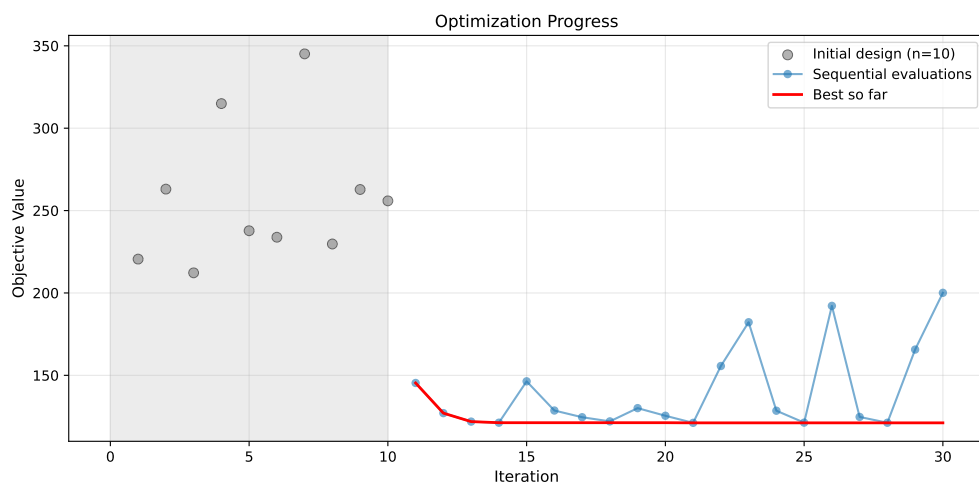
Best weight: 121.1616 lb

12. Surrogate Model Selection in SpotOptim

Function evaluations: 30
Time: 61.31s
Success: True

12.8.1. Visualization: Kriging Model

```
optimizer_kriging.plot_progress(log_y=False, figsize=(10, 5))
```



```
optimizer_kriging.plot_important_hyperparameter_contour(max_imp=3)
plt.suptitle('SpotOptim Kriging: Most Important Parameters', y=1.02)
plt.show()
```

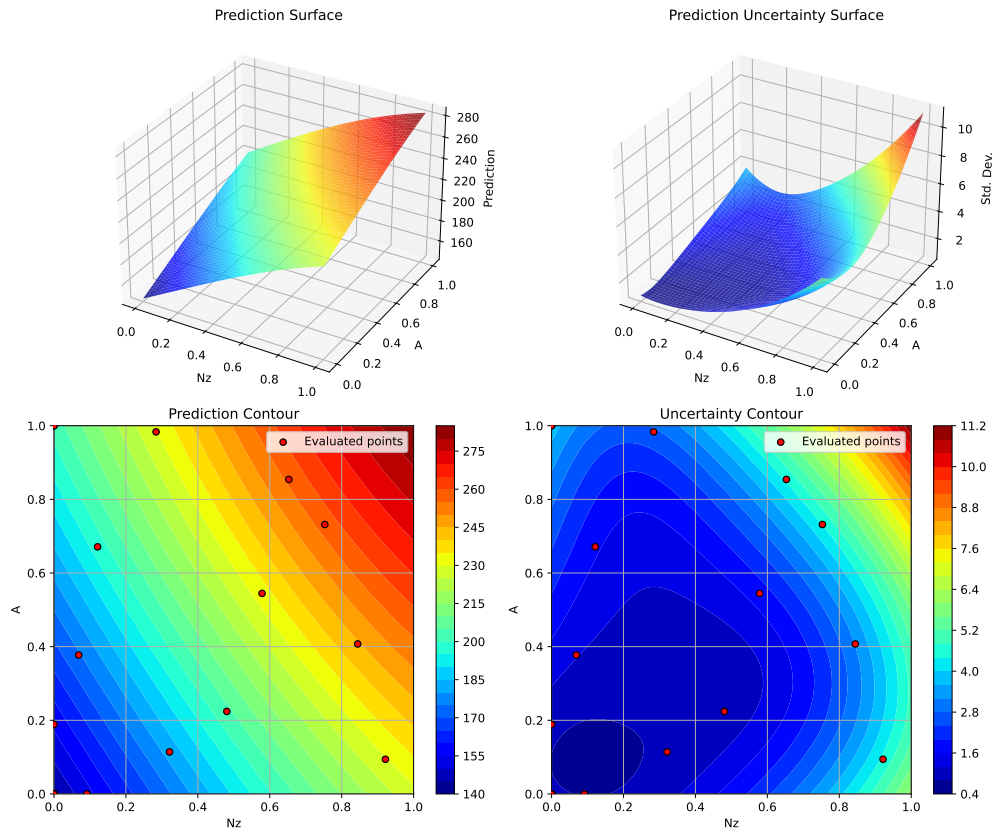
Plotting surrogate contours for top 3 most important parameters:

Nz: importance = 19.13% (type: float)
A: importance = 16.01% (type: float)
Wdg: importance = 14.70% (type: float)

Generating 3 surrogate plots...

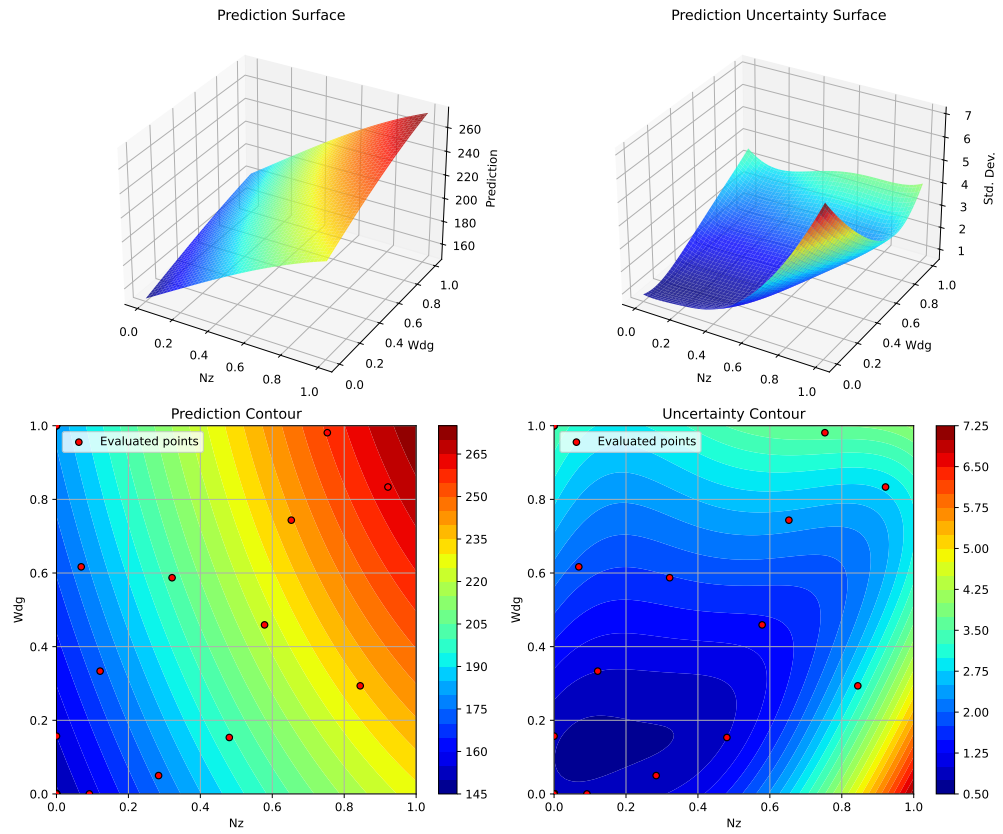
Plotting Nz vs A

12.8. 5. SpotOptim Kriging Model



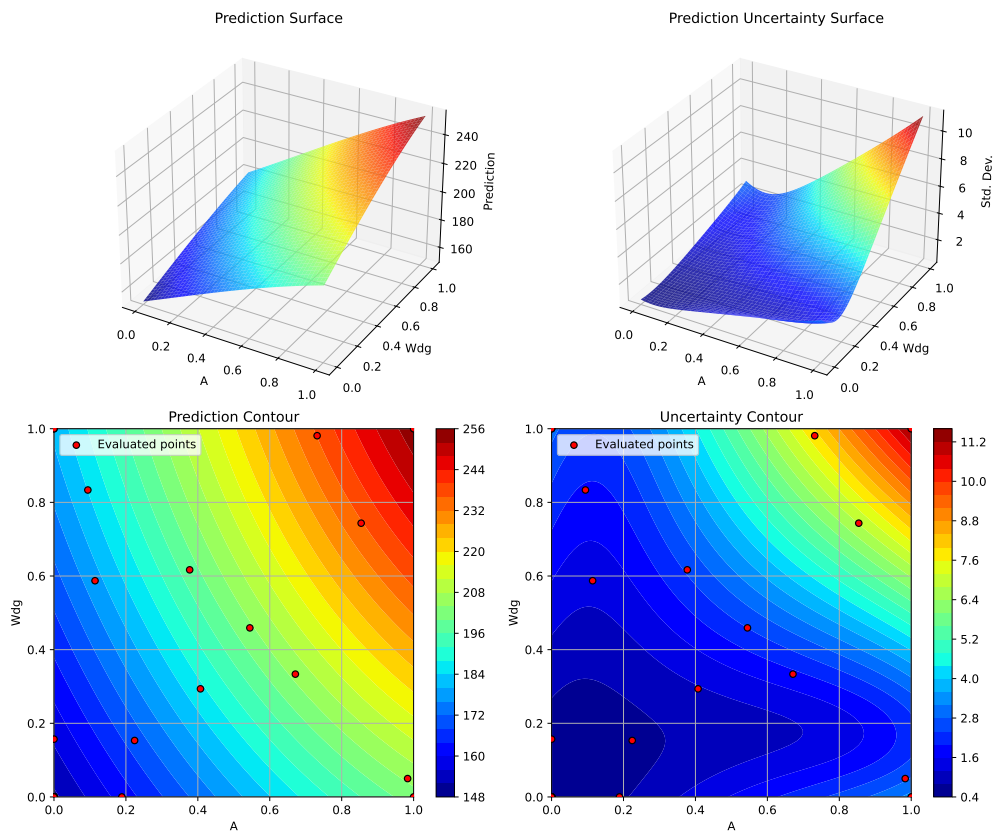
Plotting N_z vs Wdg

12. Surrogate Model Selection in SpotOptim



Plotting A vs Wdg

12.9. 6. Random Forest Regressor



<Figure size 1650x1050 with 0 Axes>

12.9. 6. Random Forest Regressor

Random Forests are ensemble methods that handle noise well and can model discontinuities. They don't naturally provide uncertainty estimates, so the acquisition function uses predictions only.

```
print("=" * 80)
print("6. Random Forest Regressor")
print("=" * 80)

start_time = time.time()

# Configure Random Forest
```

12. Surrogate Model Selection in SpotOptim

```
rf_model = RandomForestRegressor(  
    n_estimators=100,  
    max_depth=15,  
    min_samples_split=2,  
    min_samples_leaf=1,  
    random_state=seed,  
    n_jobs=-1  
)  
  
optimizer_rf = SpotOptim(  
    fun=wingwt,  
    bounds=bounds,  
    surrogate=rf_model,  
    max_iter=max_iter,  
    n_initial=n_initial,  
    var_name=param_names,  
    acquisition='y', # Use 'y' (greedy) since RF doesn't provide std  
    seed=seed,  
    verbose=False  
)  
  
result_rf = optimizer_rf.optimize()  
time_rf = time.time() - start_time  
  
print(f"\nResults:")  
print(f"  Best weight: {result_rf.fun:.4f} lb")  
print(f"  Function evaluations: {result_rf.nfev}")  
print(f"  Time: {time_rf:.2f}s")  
print(f"  Success: {result_rf.success}")  
print(f"  Note: Using acquisition='y' (greedy) since RF doesn't provide uncertainty")  
  
results_comparison.append({  
    'Surrogate': 'Random Forest',  
    'Best Weight': result_rf.fun,  
    'Evaluations': result_rf.nfev,  
    'Time (s)': time_rf,  
    'Success': result_rf.success  
})
```

6. Random Forest Regressor

Results:

Best weight: 152.6161 lb

Function evaluations: 30

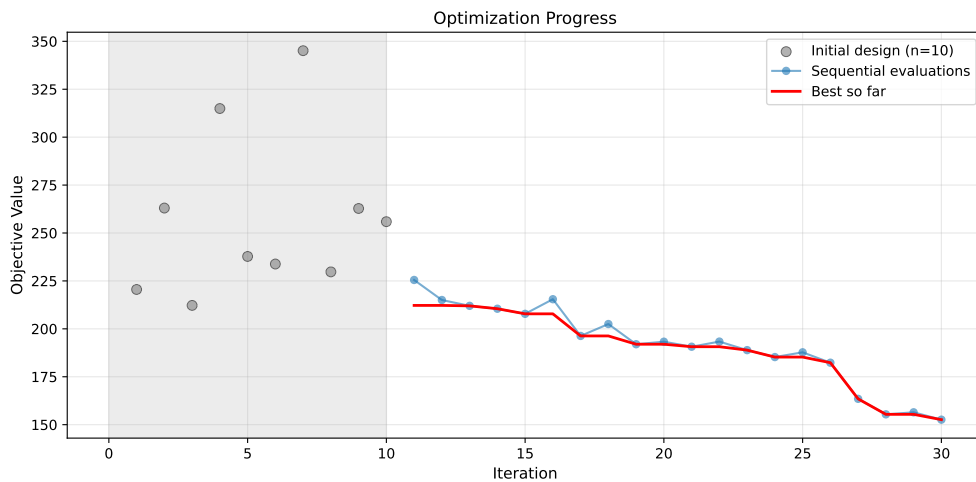
Time: 996.31s

Success: True

Note: Using acquisition='y' (greedy) since RF doesn't provide uncertainty

12.9.1. Visualization: Random Forest

```
optimizer_rf.plot_progress(log_y=False, figsize=(10, 5))
```



```
optimizer_rf.plot_important_hyperparameter_contour(max_imp=3)
plt.suptitle('Random Forest: Most Important Parameters', y=1.02)
plt.show()
```

Plotting surrogate contours for top 3 most important parameters:

Wdg: importance = 14.76% (type: float)

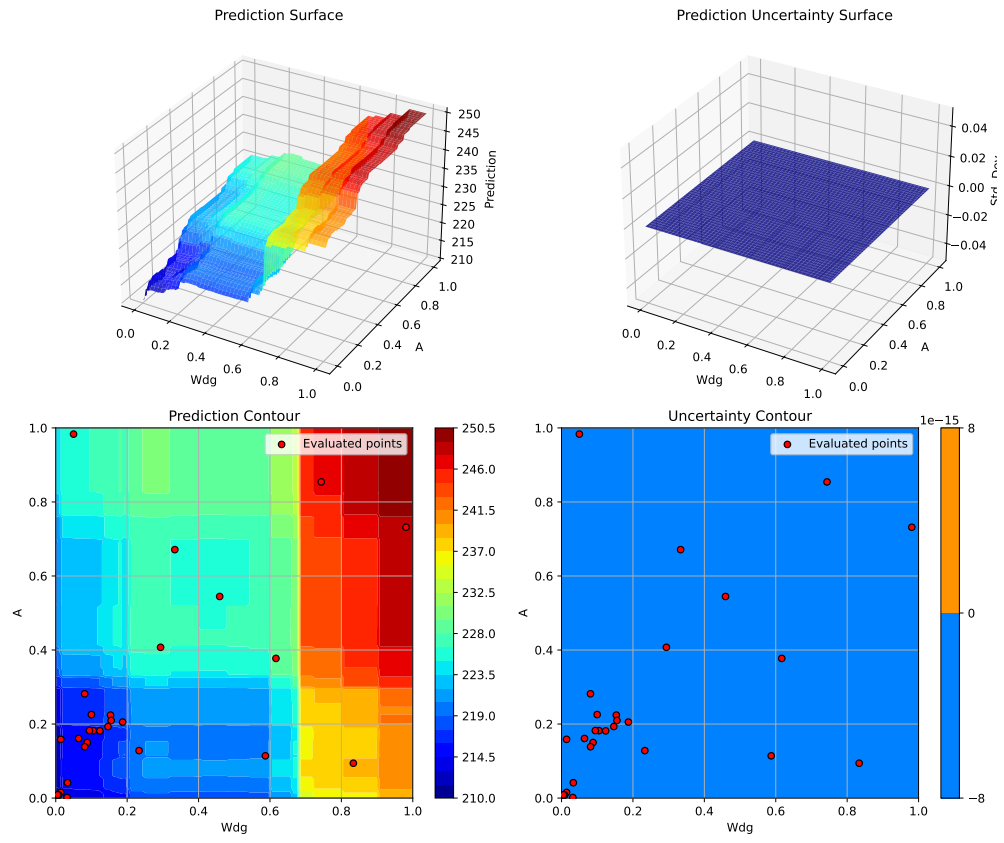
A: importance = 14.44% (type: float)

Rtc: importance = 13.91% (type: float)

Generating 3 surrogate plots...

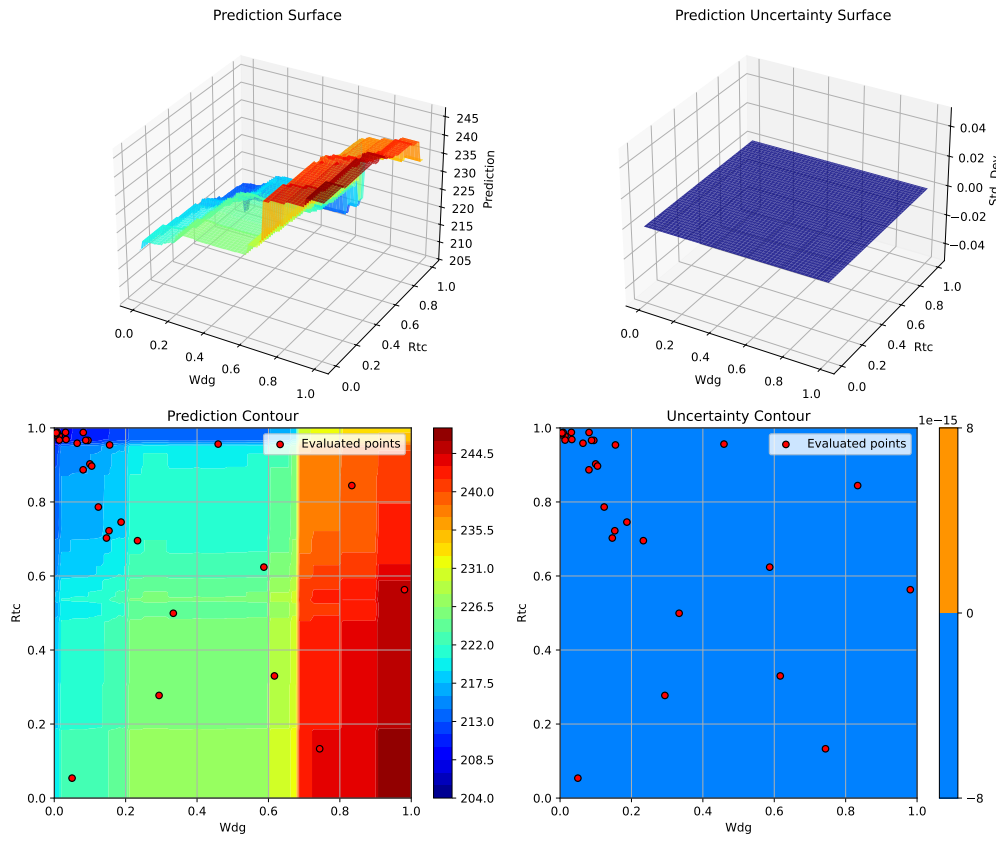
Plotting Wdg vs A

12. Surrogate Model Selection in SpotOptim



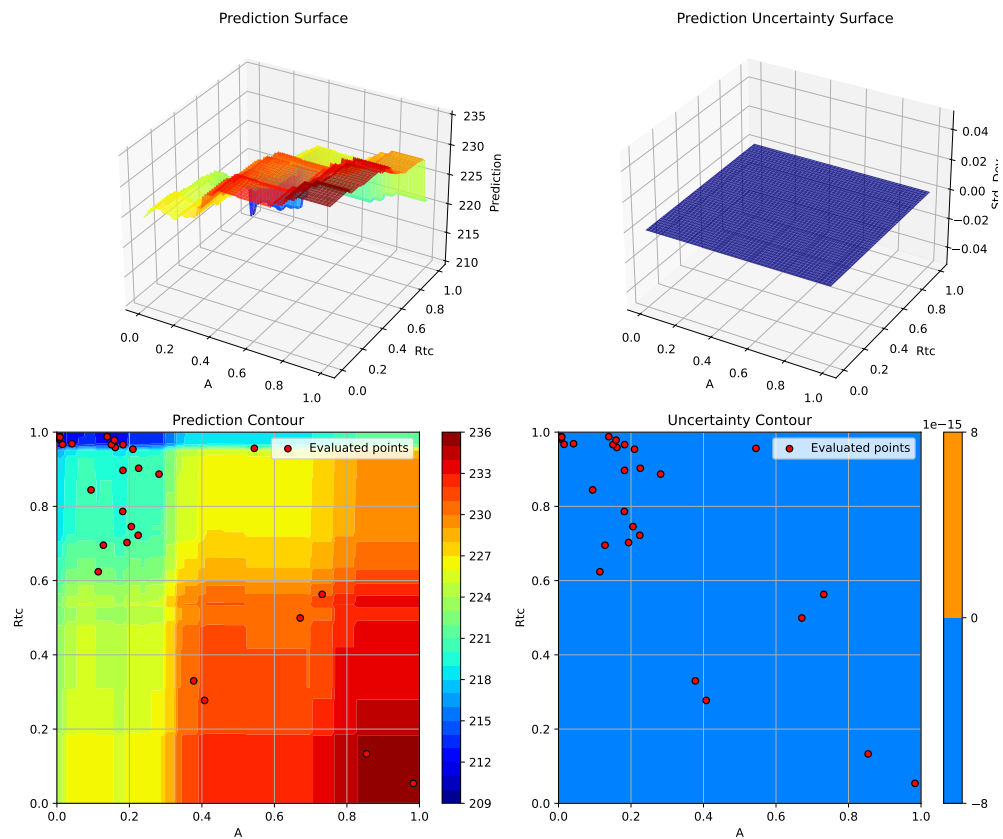
Plotting Wdg vs Rtc

12.9. 6. Random Forest Regressor



Plotting A vs Rtc

12. Surrogate Model Selection in SpotOptim



<Figure size 1650x1050 with 0 Axes>

12.10. 7. XGBoost Regressor

XGBoost is a gradient boosting implementation known for excellent performance on structured data and fast training/prediction times.

```
if XGBOOST_AVAILABLE:
    print("=" * 80)
    print("7. XGBoost Regressor")
    print("=" * 80)

    start_time = time.time()

    # Configure XGBoost
```

```

xgb_model = xgb.XGBRegressor(
    n_estimators=100,
    max_depth=6,
    learning_rate=0.1,
    subsample=0.8,
    colsample_bytree=0.8,
    random_state=seed,
    n_jobs=-1
)

optimizer_xgb = SpotOptim(
    fun=wingwt,
    bounds=bounds,
    surrogate=xgb_model,
    max_iter=max_iter,
    n_initial=n_initial,
    var_name=param_names,
    acquisition='y', # Use 'y' (greedy) since XGBoost doesn't provide std
    seed=seed,
    verbose=False
)

result_xgb = optimizer_xgb.optimize()
time_xgb = time.time() - start_time

print(f"\nResults:")
print(f" Best weight: {result_xgb.fun:.4f} lb")
print(f" Function evaluations: {result_xgb.nfev}")
print(f" Time: {time_xgb:.2f}s")
print(f" Success: {result_xgb.success}")
print(f" Note: Using acquisition='y' (greedy) since XGBoost doesn't provide uncertainty")

results_comparison.append({
    'Surrogate': 'XGBoost',
    'Best Weight': result_xgb.fun,
    'Evaluations': result_xgb.nfev,
    'Time (s)': time_xgb,
    'Success': result_xgb.success
})

# Visualization
optimizer_xgb.plot_progress(log_y=False, figsize=(10, 5))
plt.title('XGBoost: Convergence')
plt.show()

```

12. Surrogate Model Selection in SpotOptim

```
optimizer_xgb.plot_important_hyperparameter_contour(max_imp=3)
plt.suptitle('XGBoost: Most Important Parameters', y=1.02)
plt.show()
else:
    print("=" * 80)
    print("7. XGBoost Regressor - SKIPPED (not installed)")
    print("=" * 80)
    print("Install XGBoost with: pip install xgboost")
```

7. XGBoost Regressor

Results:

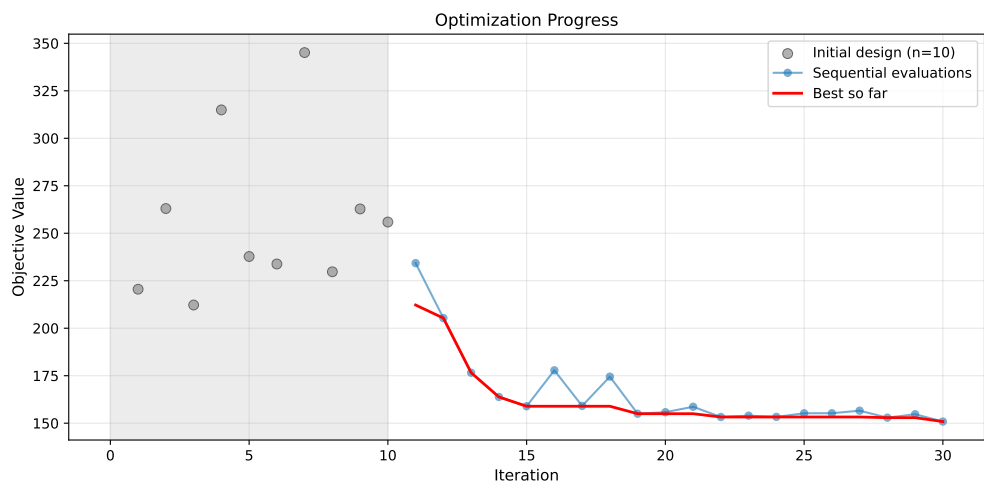
Best weight: 150.8548 lb

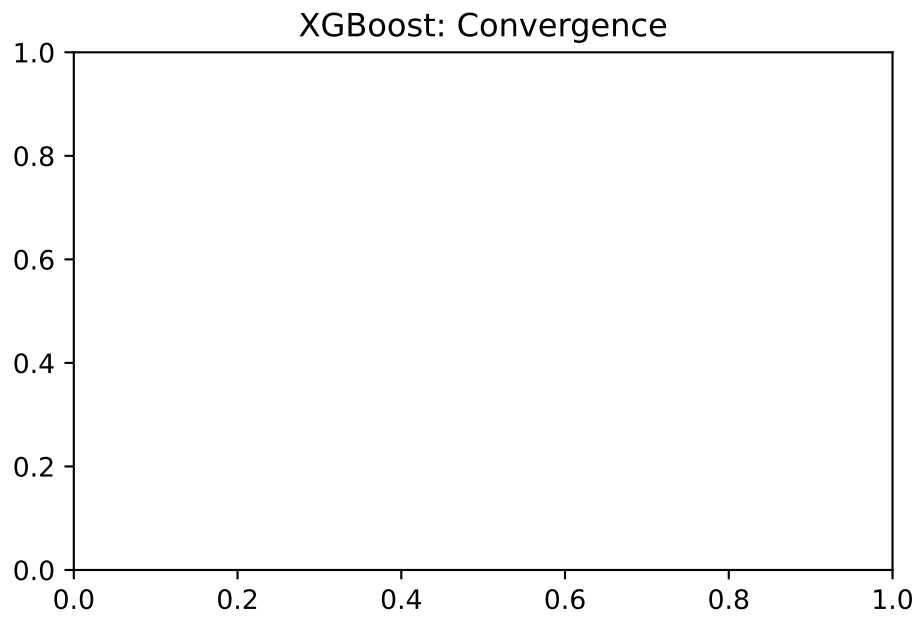
Function evaluations: 30

Time: 19.18s

Success: True

Note: Using acquisition='y' (greedy) since XGBoost doesn't provide uncertainty





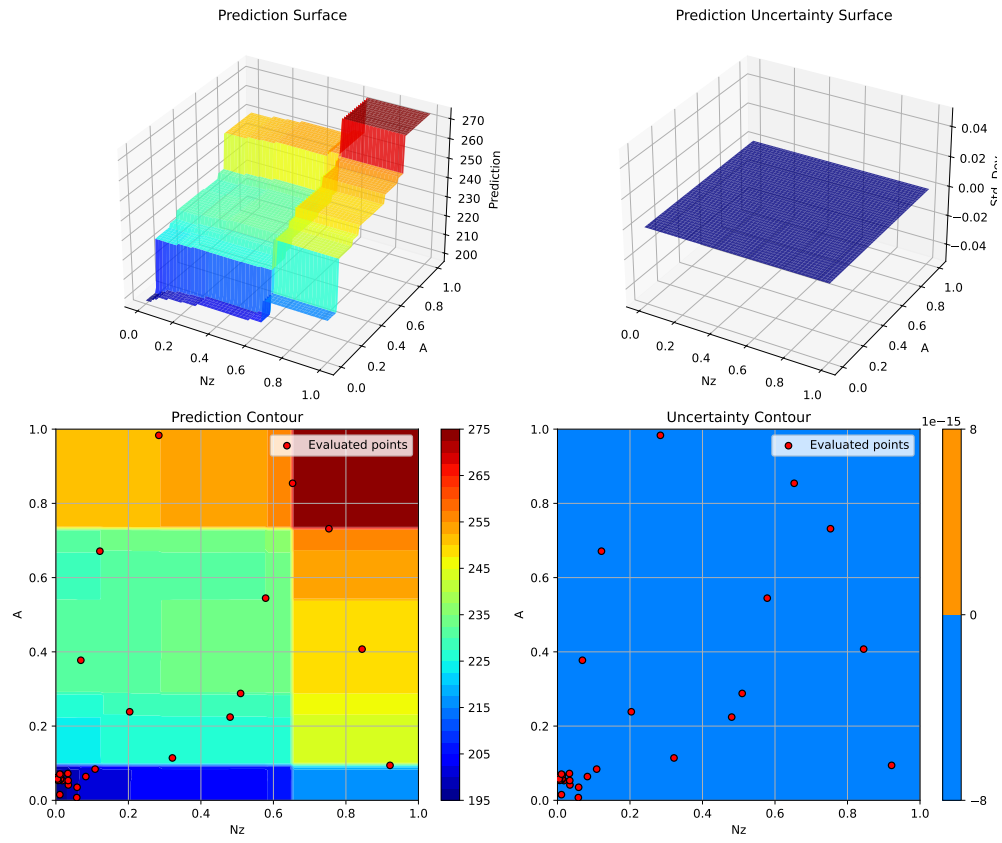
Plotting surrogate contours for top 3 most important parameters:

Nz: importance = 15.18% (type: float)
A: importance = 15.04% (type: float)
Wdg: importance = 14.49% (type: float)

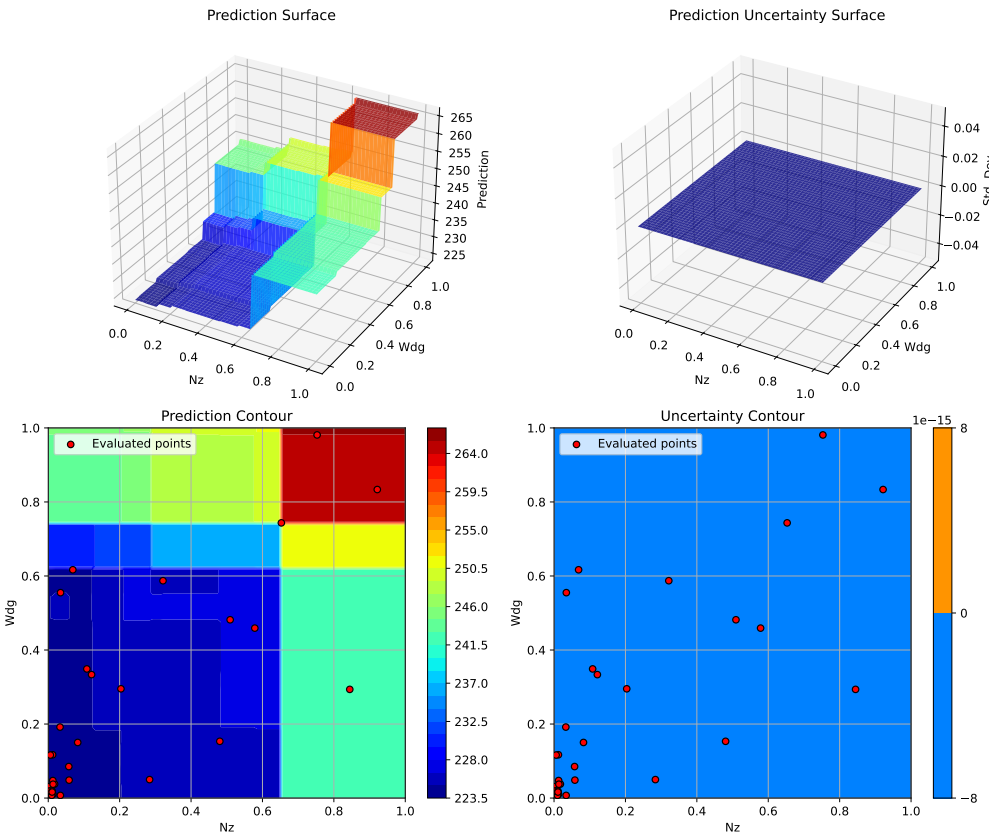
Generating 3 surrogate plots...

Plotting Nz vs A

12. Surrogate Model Selection in SpotOptim

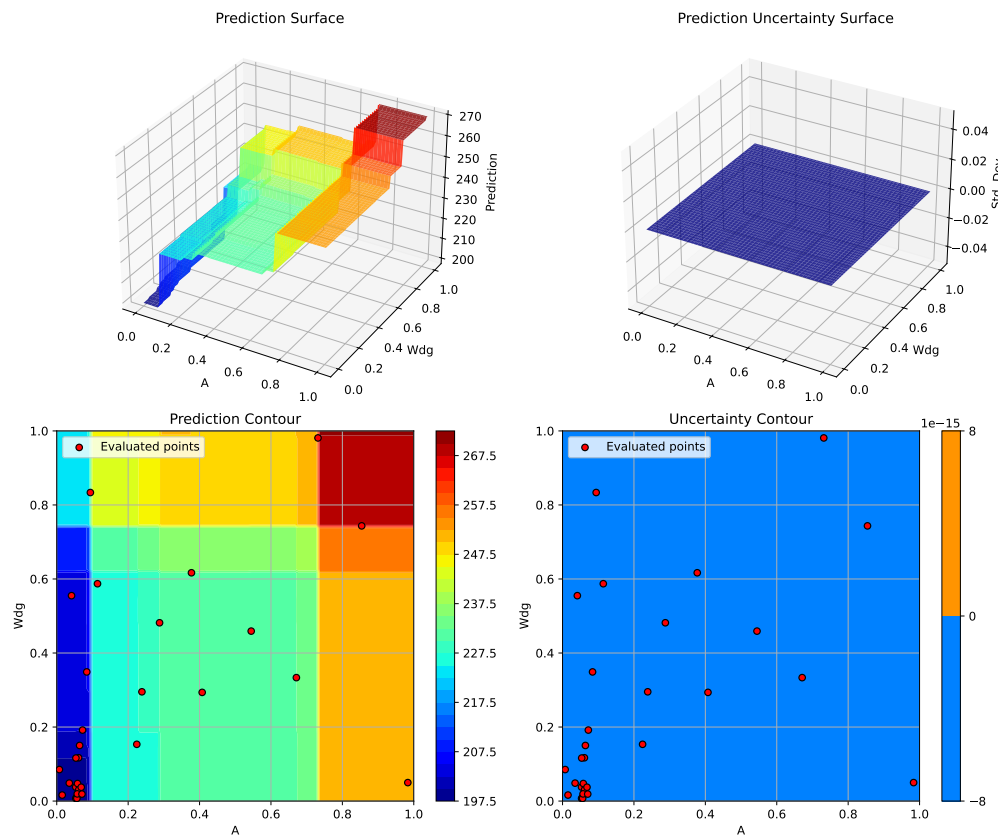


Plotting N_z vs Wdg



Plotting A vs Wdg

12. Surrogate Model Selection in SpotOptim



<Figure size 1650x1050 with 0 Axes>

12.11. 8. Support Vector Regression (SVR)

SVR with RBF kernel can model complex non-linear relationships. It's particularly good for high-dimensional problems with smooth structure.

```
print("=" * 80)
print("8. Support Vector Regression (SVR)")
print("=" * 80)

start_time = time.time()

# Configure SVR
svr_model = SVR(
```

12.11. 8. Support Vector Regression (SVR)

```
kernel='rbf',
C=100.0,
epsilon=0.1,
gamma='scale'
)

optimizer_svr = SpotOptim(
    fun=wingwt,
    bounds=bounds,
    surrogate=svr_model,
    max_iter=max_iter,
    n_initial=n_initial,
    var_name=param_names,
    acquisition='y', # Use 'y' (greedy) since SVR doesn't provide std by default
    seed=seed,
    verbose=False
)

result_svr = optimizer_svr.optimize()
time_svr = time.time() - start_time

print(f"\nResults:")
print(f"  Best weight: {result_svr.fun:.4f} lb")
print(f"  Function evaluations: {result_svr.nfev}")
print(f"  Time: {time_svr:.2f}s")
print(f"  Success: {result_svr.success}")

results_comparison.append({
    'Surrogate': 'SVR (RBF)',
    'Best Weight': result_svr.fun,
    'Evaluations': result_svr.nfev,
    'Time (s)': time_svr,
    'Success': result_svr.success
})
```

8. Support Vector Regression (SVR)

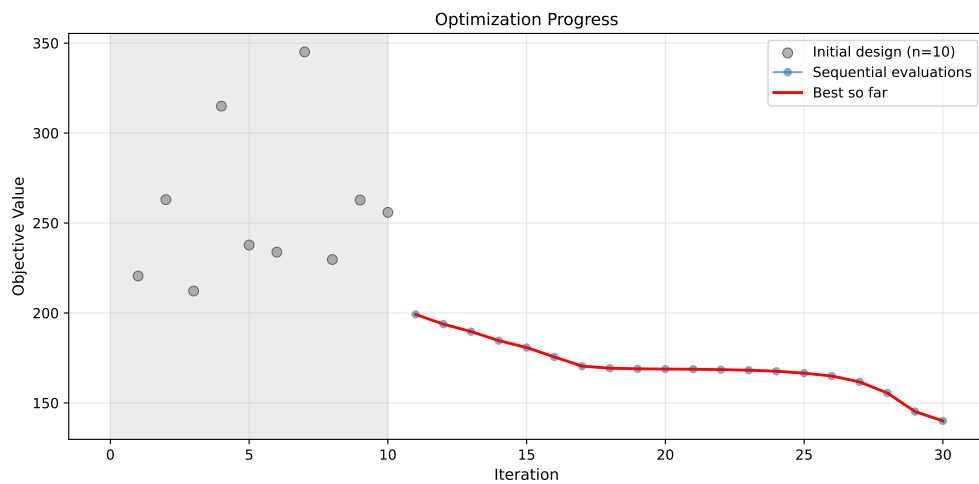
Results:

```
Best weight: 140.0109 lb
Function evaluations: 30
Time: 1.74s
Success: True
```

12. Surrogate Model Selection in SpotOptim

12.11.1. Visualization: SVR

```
optimizer_svr.plot_progress(log_y=False, figsize=(10, 5))
```



```
optimizer_svr.plot_important_hyperparameter_contour(max_imp=3)
plt.suptitle('Support Vector Regression: Most Important Parameters', y=1.02)
plt.show()
```

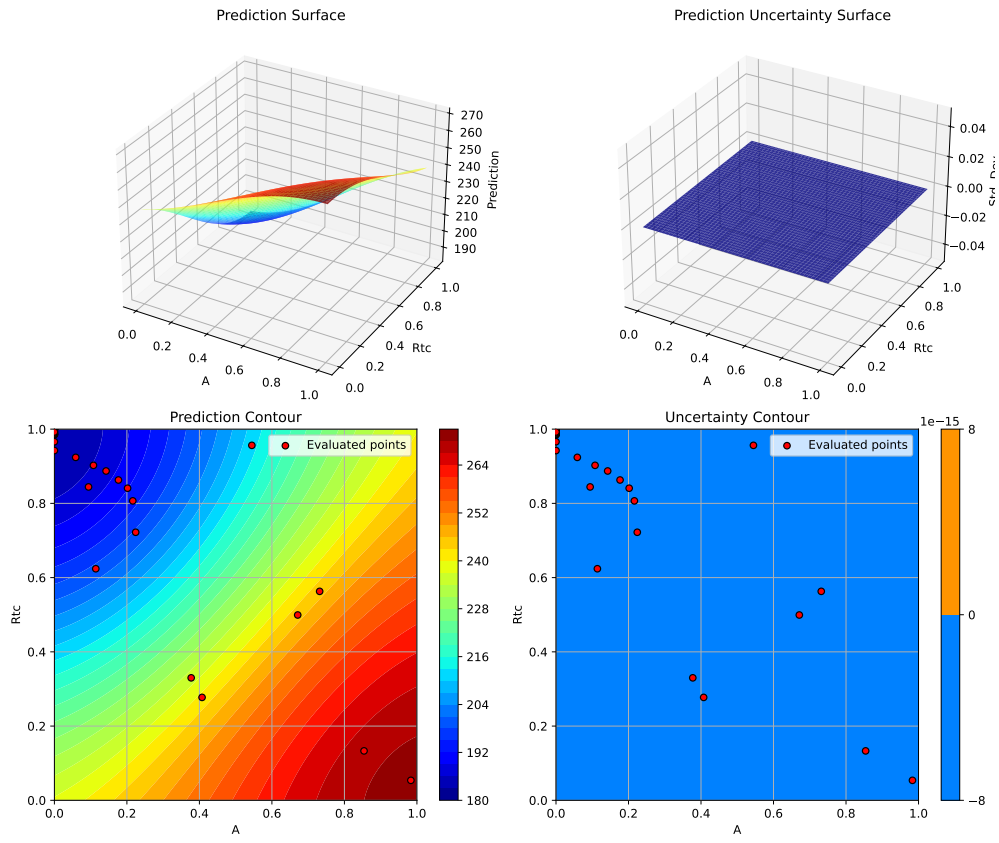
Plotting surrogate contours for top 3 most important parameters:

A: importance = 18.74% (type: float)
Rtc: importance = 18.17% (type: float)
Wdg: importance = 18.14% (type: float)

Generating 3 surrogate plots...

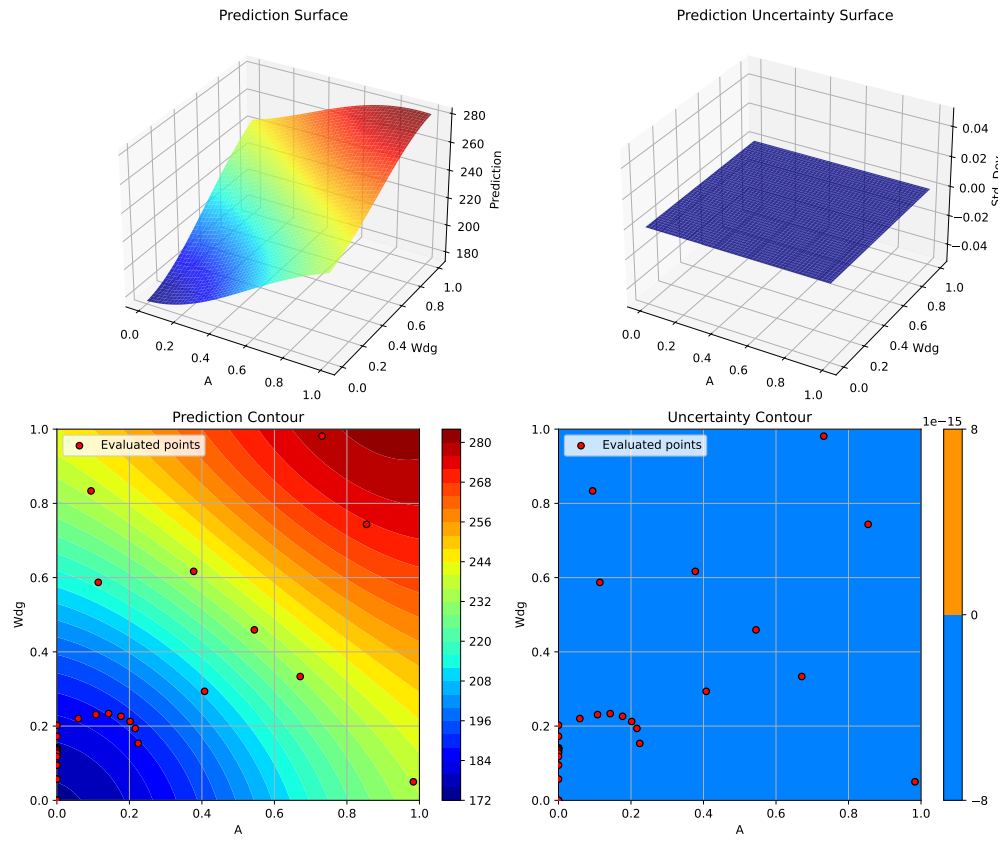
Plotting A vs Rtc

12.11. 8. Support Vector Regression (SVR)



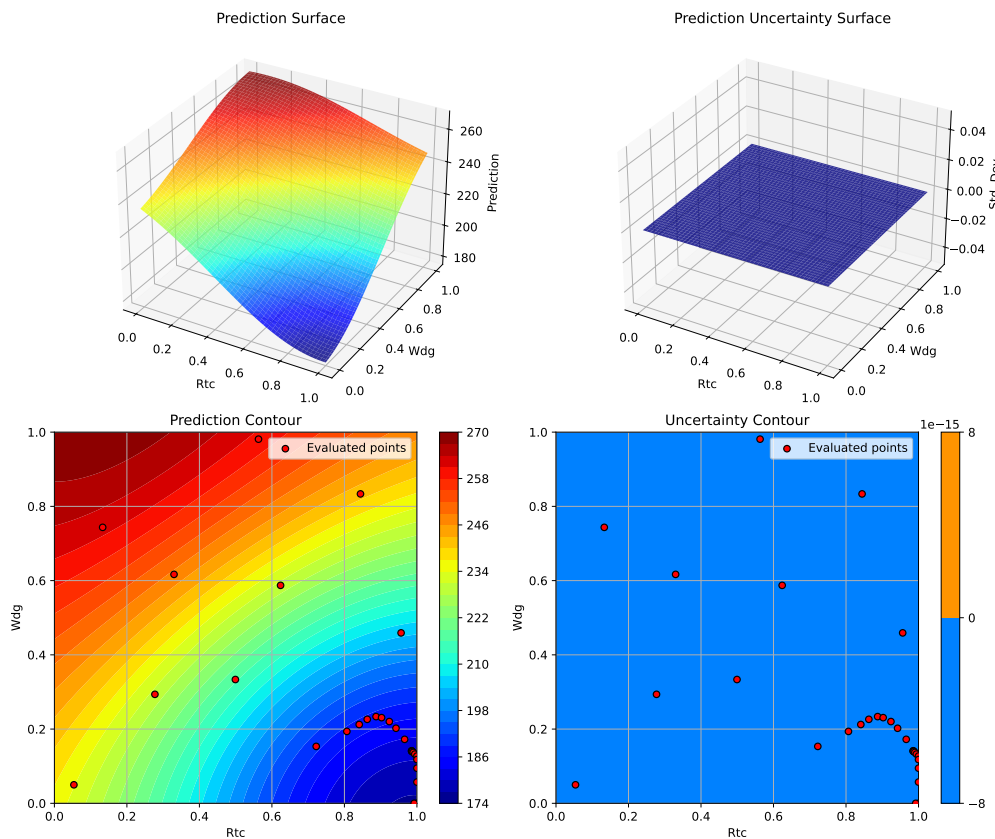
Plotting A vs W_{dg}

12. Surrogate Model Selection in SpotOptim



Plotting Rtc vs Wdg

12.12. 9. Gradient Boosting Regressor



<Figure size 1650x1050 with 0 Axes>

12.12. 9. Gradient Boosting Regressor

Gradient Boosting from scikit-learn is another ensemble method that builds trees sequentially, with each tree correcting errors of the previous ones.

```
print("=" * 80)
print("9. Gradient Boosting Regressor")
print("=" * 80)

start_time = time.time()

# Configure Gradient Boosting
gb_model = GradientBoostingRegressor(
```

12. Surrogate Model Selection in SpotOptim

```
n_estimators=100,
max_depth=5,
learning_rate=0.1,
subsample=0.8,
random_state=seed
)

optimizer_gb = SpotOptim(
    fun=wingwt,
    bounds=bounds,
    surrogate=gb_model,
    max_iter=max_iter,
    n_initial=n_initial,
    var_name=param_names,
    acquisition='y', # Use 'y' (greedy) since GB doesn't provide std
    seed=seed,
    verbose=False
)

result_gb = optimizer_gb.optimize()
time_gb = time.time() - start_time

print(f"\nResults:")
print(f"  Best weight: {result_gb.fun:.4f} lb")
print(f"  Function evaluations: {result_gb.nfev}")
print(f"  Time: {time_gb:.2f}s")
print(f"  Success: {result_gb.success}")

results_comparison.append({
    'Surrogate': 'Gradient Boosting',
    'Best Weight': result_gb.fun,
    'Evaluations': result_gb.nfev,
    'Time (s)': time_gb,
    'Success': result_gb.success
})
```

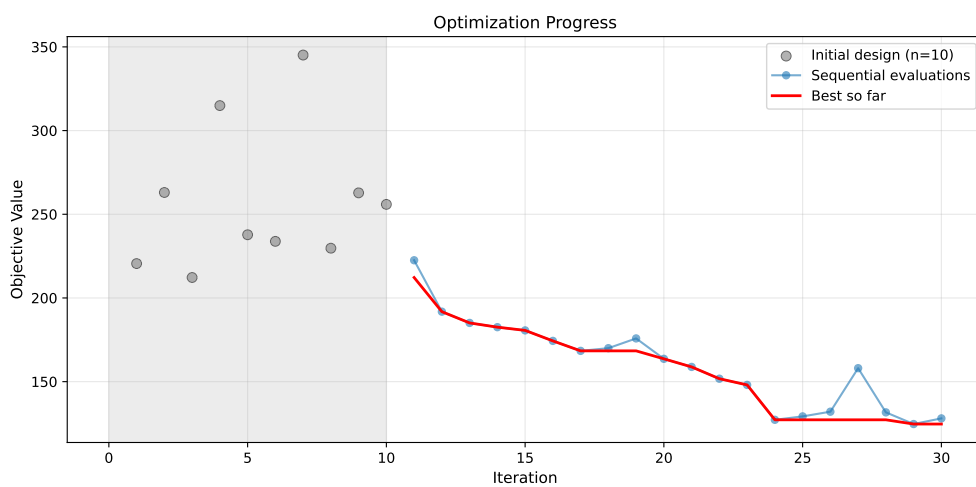
9. Gradient Boosting Regressor

Results:
Best weight: 124.6772 lb
Function evaluations: 30
Time: 7.38s

Success: True

12.12.1. Visualization: Gradient Boosting

```
optimizer_gb.plot_progress(log_y=False, figsize=(10, 5))
```



```
optimizer_gb.plot_important_hyperparameter_contour(max_imp=3)
plt.suptitle('Gradient Boosting: Most Important Parameters', y=1.02)
plt.show()
```

Plotting surrogate contours for top 3 most important parameters:

Wdg: importance = 16.80% (type: float)

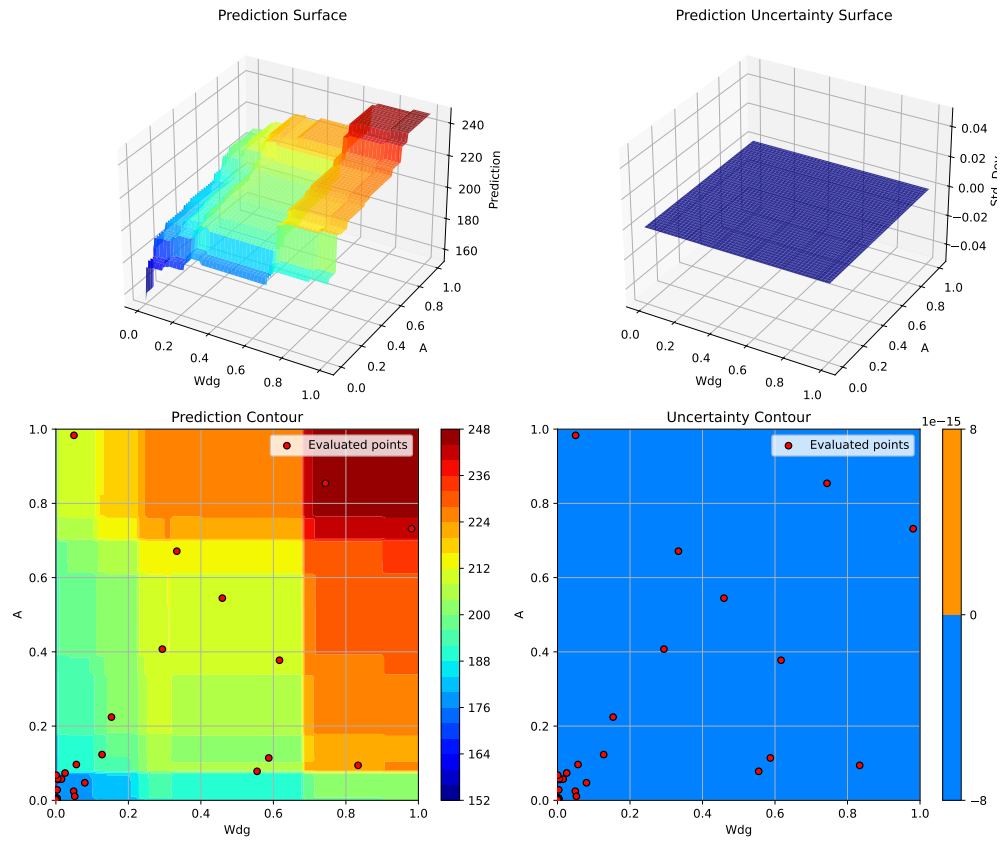
A: importance = 16.28% (type: float)

Rtc: importance = 15.69% (type: float)

Generating 3 surrogate plots...

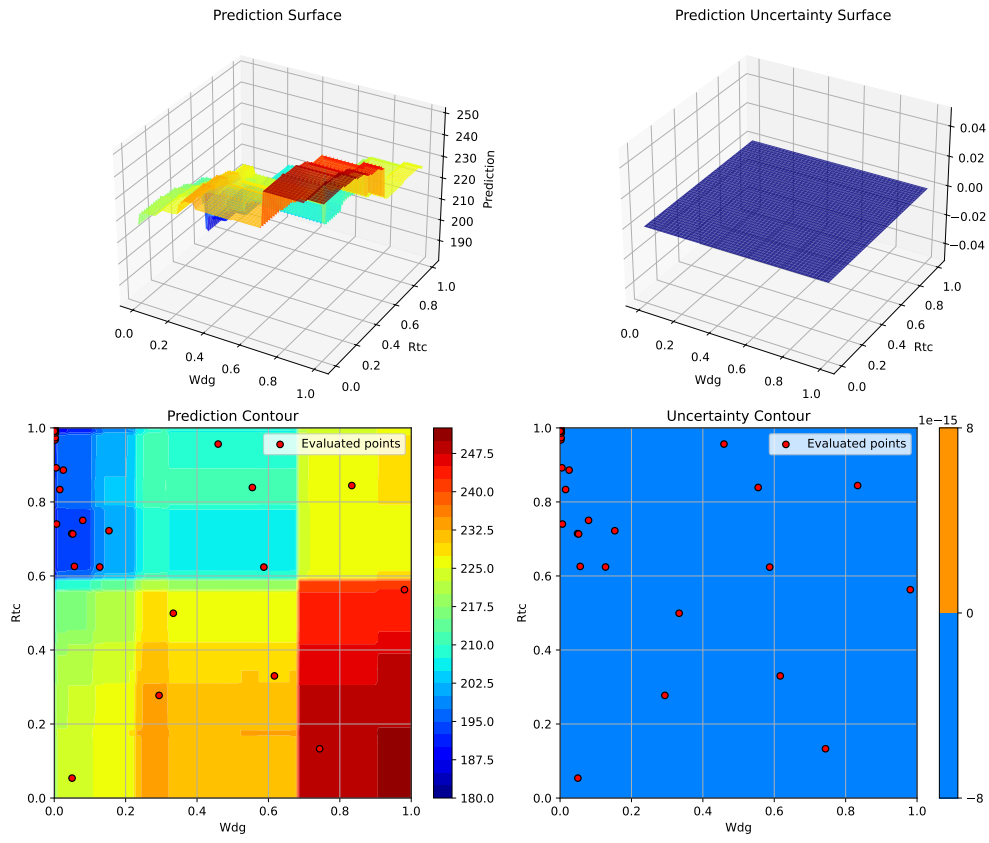
Plotting Wdg vs A

12. Surrogate Model Selection in SpotOptim



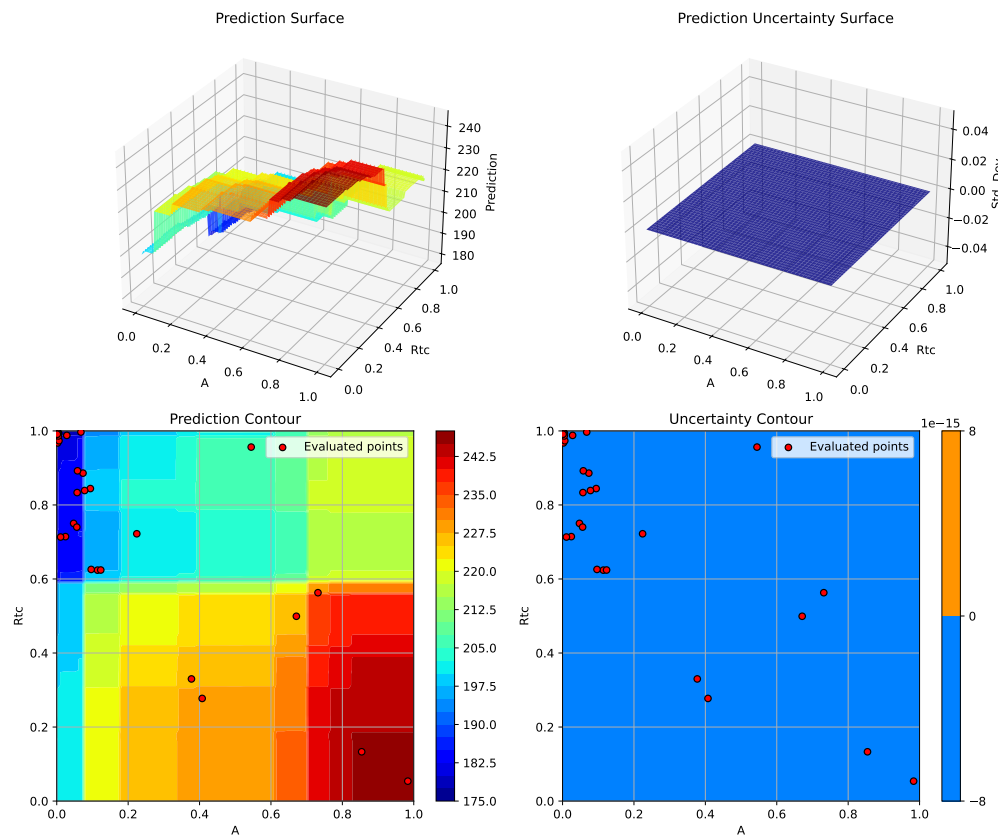
Plotting Wdg vs Rtc

12.12. 9. Gradient Boosting Regressor



Plotting A vs Rtc

12. Surrogate Model Selection in SpotOptim



<Figure size 1650x1050 with 0 Axes>

12.13. Comprehensive Comparison

Now let's compare all surrogate models side-by-side.

```
# Create comparison DataFrame
df_comparison = pd.DataFrame(results_comparison)

# Calculate improvement from best
best_weight = df_comparison['Best Weight'].min()
df_comparison['Gap to Best (%)'] = (
    (df_comparison['Best Weight'] - best_weight) / best_weight * 100
)
```

12.13. Comprehensive Comparison

```
# Sort by best weight
df_comparison = df_comparison.sort_values('Best Weight')

print("\n" + "=" * 100)
print("SURROGATE MODEL COMPARISON")
print("=" * 100)
print(df_comparison.to_string(index=False))
print("=" * 100)
```

SURROGATE MODEL COMPARISON

	Surrogate	Best Weight	Evaluations	Time (s)	Success	Gap to Best (%)
	GP RBF	119.503672	30	50.997636	True	0.000000
GP Matern =2.5 (Default)		119.504103	30	41.054601	True	0.000361
	GP Matern =1.5	119.505612	30	39.275846	True	0.001624
	GP Rational Quadratic	119.506083	30	46.181950	True	0.002017
	SpotOptim Kriging	121.161582	30	61.309369	True	1.387330
	Gradient Boosting	124.677233	30	7.379013	True	4.329207
	SVR (RBF)	140.010945	30	1.735399	True	17.160371
	XGBoost	150.854809	30	19.175808	True	26.234455
	Random Forest	152.616148	30	996.306659	True	27.708333

12.13.1. Visualization: Performance Comparison

```
fig, axes = plt.subplots(2, 2, figsize=(16, 12))

# Plot 1: Best weight comparison
ax1 = axes[0, 0]
colors = ['green' if i == 0 else 'steelblue' for i in range(len(df_comparison))]
ax1.barh(df_comparison['Surrogate'], df_comparison['Best Weight'], color=colors)
ax1.set_xlabel('Best Weight (lb)')
ax1.set_title('Best Weight Found by Each Surrogate')
ax1.axvline(x=best_weight, color='red', linestyle='--', linewidth=2, label='Best Overall')
ax1.legend()
ax1.grid(True, alpha=0.3, axis='x')

# Plot 2: Computational time
ax2 = axes[0, 1]
ax2.barh(df_comparison['Surrogate'], df_comparison['Time (s)'], color='coral')
```

12. Surrogate Model Selection in SpotOptim

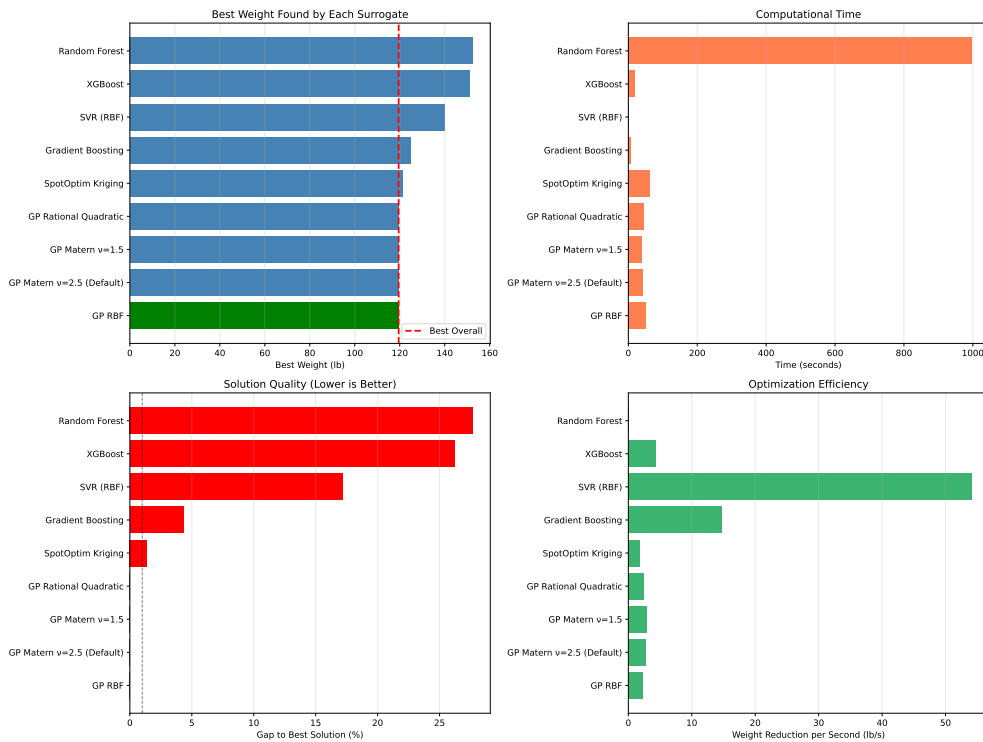
```
ax2.set_xlabel('Time (seconds)')
ax2.set_title('Computational Time')
ax2.grid(True, alpha=0.3, axis='x')

# Plot 3: Gap to best
ax3 = axes[1, 0]
colors_gap = ['green' if gap < 0.1 else 'orange' if gap < 1.0 else 'red'
              for gap in df_comparison['Gap to Best (%)']]
ax3.barh(df_comparison['Surrogate'], df_comparison['Gap to Best (%)'], color=colors_gap)
ax3.set_xlabel('Gap to Best Solution (%)')
ax3.set_title('Solution Quality (Lower is Better)')
ax3.axvline(x=1.0, color='black', linestyle='--', alpha=0.5, linewidth=1)
ax3.grid(True, alpha=0.3, axis='x')

# Plot 4: Efficiency (weight reduction per second)
ax4 = axes[1, 1]
baseline_weight = wingwt(np.array([0.48, 0.4, 0.38, 0.5, 0.62, 0.344, 0.4, 0.37, 0.38]))
df_comparison['Efficiency'] = (baseline_weight - df_comparison['Best Weight']) / df_comparison['Time (seconds)']
ax4.barh(df_comparison['Surrogate'], df_comparison['Efficiency'], color='mediumseagreen')
ax4.set_xlabel('Weight Reduction per Second (lb/s)')
ax4.set_title('Optimization Efficiency')
ax4.grid(True, alpha=0.3, axis='x')

plt.tight_layout()
plt.show()
```


12.14. Convergence Comparison



12.14. Convergence Comparison

Let's compare how different surrogates converge over iterations.

```
fig, ax = plt.subplots(figsize=(14, 8))

# Collect convergence data from all optimizers
optimizers = [
    (optimizer_default, 'GP Matern =2.5 (Default)', 'blue'),
    (optimizer_rbf, 'GP RBF', 'green'),
    (optimizer_matern15, 'GP Matern =1.5', 'red'),
    (optimizer_rq, 'GP Rational Quadratic', 'purple'),
    (optimizer_kriging, 'SpotOptim Kriging', 'orange'),
    (optimizer_rf, 'Random Forest', 'brown'),
    (optimizer_svr, 'SVR', 'pink'),
    (optimizer_gb, 'Gradient Boosting', 'gray')
]
```

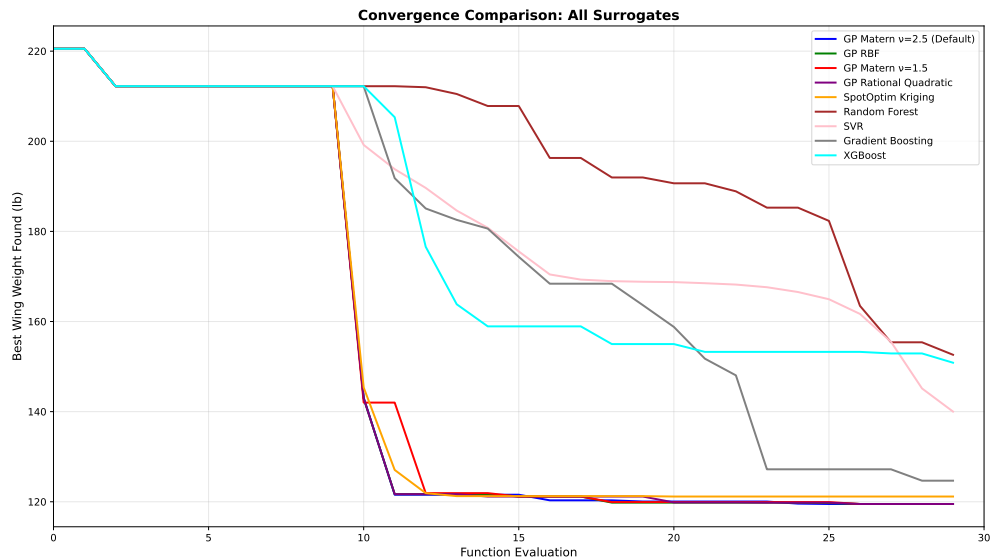
12. Surrogate Model Selection in SpotOptim

```
if XGBOOST_AVAILABLE:
    optimizers.append((optimizer_xgb, 'XGBoost', 'cyan'))

for opt, label, color in optimizers:
    y_history = opt.y_
    best_so_far = np.minimum.accumulate(y_history)
    ax.plot(range(len(best_so_far)), best_so_far, linewidth=2, label=label, color=color)

ax.set_xlabel('Function Evaluation', fontsize=12)
ax.set_ylabel('Best Wing Weight Found (lb)', fontsize=12)
ax.set_title('Convergence Comparison: All Surrogates', fontsize=14, fontweight='bold')
ax.legend(loc='upper right', fontsize=10)
ax.grid(True, alpha=0.3)
ax.set_xlim(0, max_iter)

plt.tight_layout()
plt.show()
```



12.15. Key Insights and Recommendations

```
print("\n" + "=" * 100)
print("KEY INSIGHTS AND RECOMMENDATIONS")
```

12.15. Key Insights and Recommendations

```
print("=" * 100)

# Find best surrogate
best_surrogate = df_comparison.iloc[0]['Surrogate']
best_value = df_comparison.iloc[0]['Best Weight']
best_time = df_comparison.iloc[0]['Time (s)']

print(f"\n1. BEST OVERALL PERFORMANCE:")
print(f"    Surrogate: {best_surrogate}")
print(f"    Best Weight: {best_value:.4f} lb")
print(f"    Computation Time: {best_time:.2f}s")

# Find fastest
fastest_idx = df_comparison['Time (s)'].idxmin()
fastest_surrogate = df_comparison.loc[fastest_idx, 'Surrogate']
fastest_time = df_comparison.loc[fastest_idx, 'Time (s)']

print(f"\n2. FASTEST OPTIMIZATION:")
print(f"    Surrogate: {fastest_surrogate}")
print(f"    Time: {fastest_time:.2f}s")
print(f"    Best Weight: {df_comparison.loc[fastest_idx, 'Best Weight']:.4f} lb")

# Find most efficient
most_efficient_idx = df_comparison['Efficiency'].idxmax()
most_efficient = df_comparison.loc[most_efficient_idx, 'Surrogate']

print(f"\n3. MOST EFFICIENT (weight reduction per second):")
print(f"    Surrogate: {most_efficient}")
print(f"    Efficiency: {df_comparison.loc[most_efficient_idx, 'Efficiency']:.4f} lb/s")

print(f"\n4. RECOMMENDATIONS BY PROBLEM TYPE:")
print(f"    - Smooth, continuous functions: Gaussian Process with RBF or Matern =2.5")
print(f"    - Functions with noise: Random Forest or Gradient Boosting")
print(f"    - High-dimensional problems (>20D): XGBoost or Random Forest")
print(f"    - Limited budget (<50 evals): Gaussian Process with Expected Improvement")
print(f"    - Fast evaluation needed: XGBoost or Random Forest")
print(f"    - Need uncertainty estimates: Gaussian Process or Kriging")
print(f"    - Non-smooth/discontinuous: Random Forest or Gradient Boosting")

print(f"\n5. KERNEL COMPARISON (Gaussian Process):")
gp_results = df_comparison[df_comparison['Surrogate'].str.contains('GP')]
print(gp_results[['Surrogate', 'Best Weight', 'Time (s)']].to_string(index=False))

print("\n" + "=" * 100)
```

12. Surrogate Model Selection in SpotOptim

=====

KEY INSIGHTS AND RECOMMENDATIONS

=====

1. BEST OVERALL PERFORMANCE:
Surrogate: GP RBF
Best Weight: 119.5037 lb
Computation Time: 51.00s
2. FASTEST OPTIMIZATION:
Surrogate: SVR (RBF)
Time: 1.74s
Best Weight: 140.0109 lb
3. MOST EFFICIENT (weight reduction per second):
Surrogate: SVR (RBF)
Efficiency: 54.1071 lb/s
4. RECOMMENDATIONS BY PROBLEM TYPE:
 - Smooth, continuous functions: Gaussian Process with RBF or Matern $\kappa=2.5$
 - Functions with noise: Random Forest or Gradient Boosting
 - High-dimensional problems ($>20D$): XGBoost or Random Forest
 - Limited budget (<50 evals): Gaussian Process with Expected Improvement
 - Fast evaluation needed: XGBoost or Random Forest
 - Need uncertainty estimates: Gaussian Process or Kriging
 - Non-smooth/discontinuous: Random Forest or Gradient Boosting
5. KERNEL COMPARISON (Gaussian Process):

	Surrogate	Best Weight	Time (s)
	GP RBF	119.503672	50.997636
GP Matern $\kappa=2.5$ (Default)		119.504103	41.054601
	GP Matern $\kappa=1.5$	119.505612	39.275846
	GP Rational Quadratic	119.506083	46.181950

12.16. Summary Statistics

```
# Summary statistics
print("\n" + "=" * 100)
print("SUMMARY STATISTICS")
```

```

print("=" * 100)

summary_stats = pd.DataFrame({
    'Metric': [
        'Best Weight Found',
        'Worst Weight Found',
        'Average Weight',
        'Std Dev Weight',
        'Fastest Time',
        'Slowest Time',
        'Average Time',
    ],
    'Value': [
        f"{df_comparison['Best Weight'].min():.4f} lb",
        f"{df_comparison['Best Weight'].max():.4f} lb",
        f"{df_comparison['Best Weight'].mean():.4f} lb",
        f"{df_comparison['Best Weight'].std():.4f} lb",
        f"{df_comparison['Time (s)'].min():.2f} s",
        f"{df_comparison['Time (s)'].max():.2f} s",
        f"{df_comparison['Time (s)'].mean():.2f} s",
    ]
})

print(summary_stats.to_string(index=False))
print("=" * 100)

```

```

=====
SUMMARY STATISTICS
=====

```

Metric	Value
Best Weight Found	119.5037 lb
Worst Weight Found	152.6161 lb
Average Weight	129.7045 lb
Std Dev Weight	14.1107 lb
Fastest Time	1.74 s
Slowest Time	996.31 s
Average Time	140.38 s

```

=====

```

12.17. Conclusion

This comprehensive comparison demonstrates that:

1. **Gaussian Processes** with appropriate kernels (Matern, RBF) provide excellent performance for smooth optimization problems and naturally support Expected Improvement acquisition
2. **SpotOptim Kriging** offers a lightweight alternative to sklearn's GP with comparable performance
3. **Random Forest** and **XGBoost** are robust alternatives that handle noise and discontinuities well, though they require greedy acquisition
4. **SVR** and **Gradient Boosting** offer middle-ground solutions with good scalability
5. The choice of surrogate should be based on:
 - Function smoothness
 - Computational budget
 - Need for uncertainty quantification
 - Problem dimensionality
 - Noise characteristics

For the AWWE problem, Gaussian Process surrogates generally performed best due to the function's smooth structure, but tree-based methods (RF, XGBoost, GB) can be preferable for more complex, noisy, or high-dimensional problems.

12.18. Jupyter Notebook

Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

13. Acquisition Failure Handling in SpotOptim

SpotOptim provides sophisticated fallback strategies for handling acquisition function failures during optimization. This ensures robust optimization even when the surrogate model struggles to suggest new points.

13.1. What is Acquisition Failure?

During surrogate-based optimization, the acquisition function suggests new points to evaluate. However, sometimes the suggested point is **too close** to existing points (within `tolerance_x` distance), which would provide little new information. When this happens, SpotOptim uses a **fallback strategy** to propose an alternative point.

13.2. Fallback Strategies

SpotOptim supports two fallback strategies, controlled by the `acquisition_failure_strategy` parameter:

13.2.1. 1. Random Space-Filling Design (Default)

Strategy name: "random"

This strategy uses Latin Hypercube Sampling (LHS) to generate a new space-filling point. LHS ensures good coverage of the search space by dividing each dimension into equal-probability intervals.

When to use:

- General-purpose optimization
- When you want simplicity and good space-filling properties
- Default choice for most problems

13. Acquisition Failure Handling in SpotOptim

Example:

```
from spotoptim import SpotOptim
import numpy as np

def sphere(X):
    return np.sum(X**2, axis=1)

optimizer = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=50,
    n_initial=10,
    acquisition_failure_strategy="random", # Default
    verbose=True
)

result = optimizer.optimize()
```

```
TensorBoard logging disabled
Initial best: f(x) = 0.567137
Iteration 1: New best f(x) = 0.000122
Iteration 2: f(x) = 0.000527
Iteration 3: New best f(x) = 0.000052
Iteration 4: New best f(x) = 0.000000
Iteration 5: New best f(x) = 0.000000
Iteration 6: New best f(x) = 0.000000
Iteration 7: New best f(x) = 0.000000
Iteration 8: f(x) = 0.000000
Iteration 9: f(x) = 0.000000
Iteration 10: f(x) = 0.000000
Iteration 11: New best f(x) = 0.000000
Iteration 12: f(x) = 0.000000
Iteration 13: f(x) = 0.000000
Iteration 14: f(x) = 0.000000
Iteration 15: f(x) = 0.000000
Iteration 16: New best f(x) = 0.000000
Iteration 17: f(x) = 0.000000
Iteration 18: f(x) = 0.000000
Iteration 19: f(x) = 0.000000
Iteration 20: f(x) = 0.000000
Iteration 21: f(x) = 0.000000
Iteration 22: f(x) = 0.000000
Iteration 23: f(x) = 0.000000
```



```

Iteration 24: f(x) = 0.000000
Iteration 25: f(x) = 0.000000
Iteration 26: f(x) = 0.000000
Iteration 27: f(x) = 0.000000
Iteration 28: f(x) = 0.000000
Iteration 29: f(x) = 0.000000
Iteration 30: f(x) = 0.000000
Iteration 31: New best f(x) = 0.000000
Iteration 32: f(x) = 0.000000
Iteration 33: f(x) = 0.000000
Iteration 34: f(x) = 0.000000
Iteration 35: f(x) = 0.000000
Iteration 36: f(x) = 0.000000
Iteration 37: f(x) = 0.000000
Iteration 38: f(x) = 0.000000
Iteration 39: f(x) = 0.000000
Iteration 40: f(x) = 0.000000

```

13.2.2. 2. Morris-Mitchell Minimizing Point

Strategy name: "mm"

This strategy finds a point that **maximizes the minimum distance** to all existing points. It evaluates 100 candidate points and selects the one with the largest minimum distance to the already-evaluated points, providing excellent space-filling properties.

When to use:

- When you want to ensure maximum exploration
- For problems where avoiding clustering of points is critical
- When the search space has been heavily sampled in some regions

Example:

```

from spotoptim import SpotOptim
import numpy as np

def rosenbrock(X):
    x = X[:, 0]
    y = X[:, 1]
    return (1 - x)**2 + 100 * (y - x**2)**2

optimizer = SpotOptim(
    fun=rosenbrock,
    bounds=[(-2, 2), (-2, 2)],

```

13. Acquisition Failure Handling in SpotOptim

```
max_iter=100,  
n_initial=20,  
acquisition_failure_strategy="mm", # Morris-Mitchell  
verbose=True  
)  
  
result = optimizer.optimize()
```

```
TensorBoard logging disabled  
Initial best: f(x) = 5.442086  
Iteration 1: New best f(x) = 4.152067  
Iteration 2: New best f(x) = 1.377723  
Iteration 3: New best f(x) = 1.294867  
Iteration 4: New best f(x) = 1.181174  
Iteration 5: f(x) = 2.555825  
Iteration 6: New best f(x) = 0.236822  
Iteration 7: New best f(x) = 0.233288  
Iteration 8: f(x) = 2.371494  
Iteration 9: f(x) = 0.250390  
Iteration 10: New best f(x) = 0.225073  
Iteration 11: New best f(x) = 0.207326  
Iteration 12: New best f(x) = 0.195699  
Iteration 13: New best f(x) = 0.185261  
Iteration 14: New best f(x) = 0.176937  
Iteration 15: New best f(x) = 0.169353  
Iteration 16: New best f(x) = 0.161182  
Iteration 17: New best f(x) = 0.154567  
Iteration 18: New best f(x) = 0.151787  
Iteration 19: New best f(x) = 0.151052  
Iteration 20: f(x) = 0.151476  
Iteration 21: f(x) = 0.151543  
Iteration 22: New best f(x) = 0.150745  
Iteration 23: New best f(x) = 0.148137  
Iteration 24: New best f(x) = 0.140714  
Iteration 25: New best f(x) = 0.136748  
Iteration 26: New best f(x) = 0.132837  
Iteration 27: New best f(x) = 0.128087  
Iteration 28: New best f(x) = 0.124831  
Iteration 29: New best f(x) = 0.122603  
Iteration 30: New best f(x) = 0.119469  
Iteration 31: New best f(x) = 0.115461  
Iteration 32: New best f(x) = 0.111383  
Iteration 33: New best f(x) = 0.108701  
Iteration 34: New best f(x) = 0.105350
```

13.2. Fallback Strategies

Iteration 35: New best $f(x)$ = 0.102811
Iteration 36: New best $f(x)$ = 0.100960
Iteration 37: New best $f(x)$ = 0.098896
Iteration 38: New best $f(x)$ = 0.095172
Iteration 39: New best $f(x)$ = 0.092656
Iteration 40: $f(x)$ = 0.092973
Iteration 41: New best $f(x)$ = 0.091626
Iteration 42: New best $f(x)$ = 0.090394
Iteration 43: New best $f(x)$ = 0.087906
Iteration 44: New best $f(x)$ = 0.085668
Iteration 45: New best $f(x)$ = 0.083799
Iteration 46: New best $f(x)$ = 0.080923
Iteration 47: New best $f(x)$ = 0.078246
Iteration 48: New best $f(x)$ = 0.075868
Iteration 49: New best $f(x)$ = 0.073305
Iteration 50: New best $f(x)$ = 0.070281
Iteration 51: New best $f(x)$ = 0.067384
Iteration 52: New best $f(x)$ = 0.064653
Iteration 53: New best $f(x)$ = 0.061396
Iteration 54: New best $f(x)$ = 0.057927
Iteration 55: New best $f(x)$ = 0.054721
Iteration 56: New best $f(x)$ = 0.051302
Iteration 57: New best $f(x)$ = 0.048085
Iteration 58: New best $f(x)$ = 0.046212
Iteration 59: New best $f(x)$ = 0.045729
Iteration 60: $f(x)$ = 0.045775
Iteration 61: $f(x)$ = 0.045869
Iteration 62: New best $f(x)$ = 0.045468
Iteration 63: New best $f(x)$ = 0.044802
Iteration 64: New best $f(x)$ = 0.043067
Iteration 65: New best $f(x)$ = 0.040477
Iteration 66: $f(x)$ = 0.040953
Iteration 67: New best $f(x)$ = 0.040072
Iteration 68: New best $f(x)$ = 0.039576
Iteration 69: New best $f(x)$ = 0.038934
Iteration 70: New best $f(x)$ = 0.037763
Iteration 71: New best $f(x)$ = 0.036849
Iteration 72: New best $f(x)$ = 0.036437
Iteration 73: New best $f(x)$ = 0.036404
Iteration 74: New best $f(x)$ = 0.036182
Iteration 75: New best $f(x)$ = 0.035699
Iteration 76: New best $f(x)$ = 0.035290
Iteration 77: New best $f(x)$ = 0.033974
Iteration 78: New best $f(x)$ = 0.032053
Iteration 79: New best $f(x)$ = 0.031661

13. Acquisition Failure Handling in SpotOptim

Iteration 80: $f(x) = 0.031987$

13.3. How It Works

The acquisition failure handling is integrated into the optimization process:

1. **Acquisition optimization:** SpotOptim uses differential evolution to optimize the acquisition function
2. **Distance check:** The proposed point is checked against existing points using `tolerance_x`
3. **Fallback activation:** If the point is too close, `_handle_acquisition_failure()` is called
4. **Strategy execution:** The configured fallback strategy generates a new point
5. **Evaluation:** The fallback point is evaluated and added to the dataset

13.4. Comparison of Strategies

Aspect	Random (LHS)	Morris-Mitchell
Computation	Very fast	Moderate (100 candidates)
Space-filling	Good	Excellent
Exploration	Balanced	Maximum distance
Clustering avoidance	Good	Best
Recommended for	General use	Heavily sampled spaces

13.5. Complete Example: Comparing Strategies

```
import numpy as np
from spotoptim import SpotOptim

def ackley(X):
    """Ackley function - multimodal test function"""
    a = 20
    b = 0.2
    c = 2 * np.pi
    n = X.shape[1]

    sum_sq = np.sum(X**2, axis=1)
```

13.5. Complete Example: Comparing Strategies

```
sum_cos = np.sum(np.cos(c * X), axis=1)

return -a * np.exp(-b * np.sqrt(sum_sq / n)) - np.exp(sum_cos / n) + a + np.e

# Test with random strategy
print("=" * 60)
print("Testing with Random Space-Filling Strategy")
print("=" * 60)

opt_random = SpotOptim(
    fun=ackley,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=50,
    n_initial=15,
    acquisition_failure_strategy="random",
    tolerance_x=0.1, # Relatively large tolerance to trigger failures
    seed=42,
    verbose=True
)

result_random = opt_random.optimize()

print(f"\nRandom Strategy Results:")
print(f"  Best value: {result_random.fun:.6f}")
print(f"  Best point: {result_random.x}")
print(f"  Total evaluations: {result_random.nfev}")

# Test with Morris-Mitchell strategy
print("\n" + "=" * 60)
print("Testing with Morris-Mitchell Strategy")
print("=" * 60)

opt_mm = SpotOptim(
    fun=ackley,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=50,
    n_initial=15,
    acquisition_failure_strategy="mm",
    tolerance_x=0.1, # Same tolerance
    seed=42,
    verbose=True
)

result_mm = opt_mm.optimize()
```

13. Acquisition Failure Handling in SpotOptim

```
print(f"\nMorris-Mitchell Strategy Results:")
print(f"  Best value: {result_mm.fun:.6f}")
print(f"  Best point: {result_mm.x}")
print(f"  Total evaluations: {result_mm.nfev}")

# Compare
print("\n" + "=" * 60)
print("Comparison")
print("=" * 60)
print(f"Random strategy:           {result_random.fun:.6f}")
print(f"Morris-Mitchell strategy: {result_mm.fun:.6f}")
if result_random.fun < result_mm.fun:
    print("→ Random strategy found better solution")
else:
    print("→ Morris-Mitchell strategy found better solution")
```

```
=====
Testing with Random Space-Filling Strategy
=====
TensorBoard logging disabled
Initial best: f(x) = 7.177375
Iteration 1: New best f(x) = 5.975822
Iteration 2: New best f(x) = 4.585872
Iteration 3: New best f(x) = 3.624193
Iteration 4: f(x) = 3.775836
    Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 5: f(x) = 7.428910
    Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 6: f(x) = 11.307695
    Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 7: f(x) = 11.588702
    Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 8: f(x) = 7.741328
    Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 9: f(x) = 8.547707
    Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 10: f(x) = 10.258140
```

13.5. Complete Example: Comparing Strategies

Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 11: $f(x) = 10.575833$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 12: $f(x) = 3.973820$
Iteration 13: New best $f(x) = 3.254219$
Iteration 14: $f(x) = 3.726104$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 15: $f(x) = 9.023210$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 16: $f(x) = 4.449845$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 17: $f(x) = 11.616761$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 18: $f(x) = 8.521779$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 19: $f(x) = 9.840296$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 20: $f(x) = 12.575946$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 21: $f(x) = 7.719827$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 22: $f(x) = 6.588107$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 23: $f(x) = 9.186450$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 24: $f(x) = 7.957337$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 25: $f(x) = 13.092857$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 26: $f(x) = 10.747619$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback

13. Acquisition Failure Handling in SpotOptim

Acquisition failure: Using random space-filling design as fallback.
Iteration 27: $f(x) = 9.894215$
 Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 28: $f(x) = 9.482454$
 Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 29: $f(x) = 6.720219$
 Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 30: $f(x) = 6.321851$
 Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 31: $f(x) = 6.841321$
 Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 32: $f(x) = 11.918429$
 Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 33: $f(x) = 13.292524$
 Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 34: $f(x) = 8.518795$
 Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 35: $f(x) = 12.534822$

Random Strategy Results:
 Best value: 3.254219
 Best point: [0.03533184 0.66379523]
 Total evaluations: 50

=====
Testing with Morris-Mitchell Strategy
=====

TensorBoard logging disabled
Initial best: $f(x) = 7.177375$
Iteration 1: New best $f(x) = 5.975822$
Iteration 2: New best $f(x) = 4.585872$
Iteration 3: New best $f(x) = 3.624193$
Iteration 4: $f(x) = 3.775836$
 Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 5: $f(x) = 12.759495$
 Attempt 2/10: Previous point was duplicate after rounding, trying fallback

13.5. Complete Example: Comparing Strategies

Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 6: $f(x) = 13.023769$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 7: $f(x) = 10.937206$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 8: $f(x) = 7.780549$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 9: $f(x) = 12.396463$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 10: $f(x) = 12.719725$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 11: $f(x) = 13.078957$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 12: $f(x) = 12.813177$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 13: $f(x) = 11.674342$
Iteration 14: $f(x) = 3.967415$
Iteration 15: $f(x) = 3.756184$
Iteration 16: New best $f(x) = 3.159408$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 17: $f(x) = 11.221948$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 18: $f(x) = 8.775229$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 19: $f(x) = 11.789243$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 20: $f(x) = 4.971384$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 21: $f(x) = 9.983363$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 22: $f(x) = 10.021573$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback

13. Acquisition Failure Handling in SpotOptim

Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 23: $f(x) = 6.575245$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 24: $f(x) = 11.527738$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 25: $f(x) = 12.133804$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 26: $f(x) = 10.457280$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 27: $f(x) = 11.003827$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 28: $f(x) = 13.704437$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 29: $f(x) = 10.955961$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 30: $f(x) = 12.121347$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 31: $f(x) = 7.325542$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 32: $f(x) = 4.492325$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 33: $f(x) = 6.456039$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 34: $f(x) = 10.870934$
Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using Morris-Mitchell minimizing point as fallback.
Iteration 35: $f(x) = 10.677536$

Morris-Mitchell Strategy Results:

Best value: 3.159408

Best point: [-0.00800104 0.71750603]

Total evaluations: 50

=====

Comparison

```
=====
Random strategy:          3.254219
Morris-Mitchell strategy: 3.159408
→ Morris-Mitchell strategy found better solution
```

13.6. Advanced Usage: Setting Tolerance

The `tolerance_x` parameter controls when the fallback strategy is triggered. A larger tolerance means points need to be farther apart, triggering the fallback more often:

```
def simple_objective(X):
    """Simple quadratic function for demonstration"""
    return np.sum(X**2, axis=1)

bounds_demo = [(-5, 5), (-5, 5)]

# Strict tolerance (smaller value) - fewer fallbacks
optimizer_strict = SpotOptim(
    fun=simple_objective,
    bounds=bounds_demo,
    tolerance_x=1e-6, # Very small - almost never triggers fallback
    acquisition_failure_strategy="mm",
    max_iter=20,
    seed=42
)

# Relaxed tolerance (larger value) - more fallbacks
optimizer_relaxed = SpotOptim(
    fun=simple_objective,
    bounds=bounds_demo,
    tolerance_x=0.5, # Larger - triggers fallback more often
    acquisition_failure_strategy="mm",
    max_iter=20,
    seed=42
)

print(f"Strict tolerance setup complete")
print(f"Relaxed tolerance setup complete")
```

```
Strict tolerance setup complete
Relaxed tolerance setup complete
```

13.7. Best Practices

13.7.1. 1. Use Random for Most Problems

The random strategy (default) is sufficient for most optimization problems:

```
def my_objective(X):
    return np.sum(X**2, axis=1)

optimizer = SpotOptim(
    fun=my_objective,
    bounds=[(-5, 5), (-5, 5)],
    acquisition_failure_strategy="random", # Good default choice
    max_iter=20,
    seed=42
)
print("Random strategy optimizer created")
```

Random strategy optimizer created

13.7.2. 2. Use Morris-Mitchell for Intensive Sampling

When you have a large budget and want maximum exploration:

```
def expensive_objective(X):
    """Simulated expensive objective function"""
    return np.sum((X - 1)**2, axis=1)

optimizer = SpotOptim(
    fun=expensive_objective,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=30, # Large budget
    acquisition_failure_strategy="mm", # Maximize space coverage
    seed=42
)
print("Morris-Mitchell optimizer for intensive sampling created")
```

Morris-Mitchell optimizer for intensive sampling created

13.7.3. 3. Monitor Fallback Activations

Enable verbose mode to see when fallbacks are triggered:

```
def test_objective(X):
    return np.sum(X**2, axis=1)

optimizer = SpotOptim(
    fun=test_objective,
    bounds=[(-5, 5), (-5, 5)],
    acquisition_failure_strategy="mm",
    max_iter=20,
    verbose=True, # Shows fallback messages
    seed=42
)
print("Optimizer with verbose mode created")
```

```
TensorBoard logging disabled
Optimizer with verbose mode created
```

13.7.4. 4. Adjust Tolerance Based on Problem Scale

For problems with small search spaces, use smaller tolerance:

```
def scale_objective(X):
    return np.sum(X**2, axis=1)

# Small search space
optimizer_small = SpotOptim(
    fun=scale_objective,
    bounds=[(-1, 1), (-1, 1)],
    tolerance_x=0.01, # Small tolerance for small space
    acquisition_failure_strategy="random",
    max_iter=20,
    seed=42
)

# Large search space
optimizer_large = SpotOptim(
    fun=scale_objective,
    bounds=[(-100, 100), (-100, 100)],
    tolerance_x=1.0, # Larger tolerance for large space
    acquisition_failure_strategy="mm",
```

13. Acquisition Failure Handling in SpotOptim

```
max_iter=20,  
seed=42  
)  
  
print(f"Small space optimizer created (bounds: [-1, 1])")  
print(f"Large space optimizer created (bounds: [-100, 100])")
```

```
Small space optimizer created (bounds: [-1, 1])  
Large space optimizer created (bounds: [-100, 100])
```

13.8. Technical Details

13.8.1. Morris-Mitchell Implementation

The Morris-Mitchell strategy:

1. Generates 100 candidate points using Latin Hypercube Sampling
2. For each candidate, calculates the minimum distance to all existing points
3. Selects the candidate with the maximum minimum distance

This ensures the new point is as far as possible from the densest region of evaluated points.

13.8.2. Random Strategy Implementation

The random strategy:

1. Generates a single point using Latin Hypercube Sampling
2. Ensures the point is within bounds
3. Applies variable type repairs (rounding for int/factor variables)

This is computationally efficient while maintaining good space-filling properties.

13.9. Summary

- **Default strategy** ("random"): Fast, good space-filling, suitable for most problems
- **Morris-Mitchell** ("mm"): Better space-filling, maximizes minimum distance, ideal for intensive sampling
- **Trigger**: Activated when acquisition-proposed point is too close to existing points (within `tolerance_x`)

- **Control:** Set via `acquisition_failure_strategy` parameter
- **Monitoring:** Enable `verbose=True` to see when fallbacks occur

Choose the strategy that best matches your optimization goals: - Use "**random**" for general-purpose optimization - Use "**mm**" when you want maximum exploration and have a generous function evaluation budget

13.10. Jupyter Notebook

Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

14. Point Selection Implementation in SpotOptim

14.1. Overview

This feature automatically selects a subset of evaluated points for surrogate model training when the total number of points exceeds a specified threshold.

It is implemented as a point selection mechanism for SpotOptim that mirrors the functionality in spotpython's `Spot` class.

14.2. Implementation Details

14.2.1. Parameters

Added to `SpotOptim.__init__`:

- `max_surrogate_points` (int, optional): Maximum number of points to use for surrogate fitting
- `selection_method` (str, default='distant'): Method for selecting points ('distant' or 'best')

14.2.2. Methods

1. `_select_distant_points(X, y, k)`

- Uses K-means clustering to find k clusters
- Selects the point closest to each cluster center
- Ensures space-filling properties for surrogate training
- Mimics `spotpython.utils.aggregate.select_distant_points`

2. `_select_best_cluster(X, y, k)`

- Uses K-means clustering to find k clusters
- Computes mean objective value for each cluster
- Selects all points from the cluster with the best (lowest) mean value

14. Point Selection Implementation in SpotOptim

- Mimics `spotpython.utils.aggregate.select_best_cluster`

3. `_selection_dispatcher(X, y)`

- Dispatcher method that routes to the appropriate selection function
- Returns all points if `max_surrogate_points` is None
- Mimics `spotpython.spot.spot.Spot.selection_dispatcher`

The method `_fit_surrogate(X, y)` checks if `X.shape[0] > self.max_surrogate_points`. If true, it calls `_selection_dispatcher` to get a subset. Then, it fits the surrogate only on the selected points. This implementation matches the logic in `spotpython.spot.spot.Spot.fit_surrogate`

14.3. Key Differences from spotpython

While the implementation follows `spotpython`'s design, there is a difference: `spotoptim` uses a simplified clustering, it uses `sklearn`'s `KMeans` directly instead of a custom implementation.

14.4. Example Usage

This example demonstrates the point selection feature with a limited number of surrogate points. Increase `MAX_ITER`, `N_INITIAL`, and `MAX_SURROGATE_POINTS` to see more pronounced effects.

```
from spotoptim import SpotOptim
import numpy as np

MAX_ITER = 20
N_INITIAL = 5
MAX_SURROGATE_POINTS = 10

# Define an example objective function
def sphere(X):
    """Simple sphere function for demonstration"""
    return np.sum(X**2, axis=1)

bounds = [(-5, 5), (-5, 5), (-5, 5)]

# Without point selection (default behavior)
optimizer1 = SpotOptim(
    fun=sphere,
```

```

    bounds=bounds,
    max_iter=MAX_ITER,
    n_initial=N_INITIAL,
    seed=42
)
result1 = optimizer1.optimize()
print(f"Without selection - Best value: {result1.fun:.6f}")
print(f"Total points evaluated: {result1.nfev}")

# With point selection using distant method
optimizer2 = SpotOptim(
    fun=sphere,
    bounds=bounds,
    max_iter=MAX_ITER,
    n_initial=N_INITIAL,
    max_surrogate_points=MAX_SURROGATE_POINTS,
    selection_method='distant',
    seed=42
)
result2 = optimizer2.optimize()
print(f"\nWith 'distant' selection - Best value: {result2.fun:.6f}")
print(f"Total points evaluated: {result2.nfev}")
print(f"Max surrogate points: {optimizer2.max_surrogate_points}")

# With point selection using best cluster method
optimizer3 = SpotOptim(
    fun=sphere,
    bounds=bounds,
    max_iter=MAX_ITER,
    n_initial=N_INITIAL,
    max_surrogate_points=MAX_SURROGATE_POINTS,
    selection_method='best',
    seed=42
)
result3 = optimizer3.optimize()
print(f"\nWith 'best' selection - Best value: {result3.fun:.6f}")
print(f"Total points evaluated: {result3.nfev}")
print(f"Max surrogate points: {optimizer3.max_surrogate_points}")

```

Without selection - Best value: 0.000034
Total points evaluated: 20

With 'distant' selection - Best value: 2.380385
Total points evaluated: 20

14. Point Selection Implementation in SpotOptim

Max surrogate points: 10

With 'best' selection - Best value: 2.549832

Total points evaluated: 20

Max surrogate points: 10

14.5. Benefits

1. **Scalability:** Enables efficient optimization with many function evaluations
2. **Computational efficiency:** Reduces surrogate training time for large datasets
3. **Maintained accuracy:** Careful point selection preserves model quality
4. **Flexibility:** Two selection methods for different optimization scenarios

14.6. Comparison with spotpython

Feature	spotpython	SpotOptim
Point selection via clustering		
‘distant’ method		
‘best’ method		
Selection dispatcher		
Nyström approximation		
Modular design	(utils.aggregate)	(class methods)

14.7. Jupyter Notebook

Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

Part VI.

Kriging

15. Kriging Surrogate Integration

15.1. Overview

Implementation of a Kriging (Gaussian Process) surrogate model to SpotOptim, providing an alternative to scikit-learn's GaussianProcessRegressor.

15.2. Module Structure

```
src/spotoptim/surrogate/  
__init__.py          # Module exports  
kriging.py           # Kriging implementation (~350 lines)  
README.md            # Module documentation
```

15.3. Kriging Class (src/spotoptim/surrogate/kriging.py)

Key Features:

- Scikit-learn compatible interface (`fit()`, `predict()`)
- Gaussian (RBF) kernel: $R = \exp(-D)$
- Automatic hyperparameter optimization via maximum likelihood
- Cholesky decomposition for efficient linear algebra
- Prediction with uncertainty (`return_std=True`)
- Reproducible results via seed parameter

Implementation Details:

- lean, well-documented code
- No external dependencies beyond NumPy, SciPy
- Simplified from spotpython.surrogate.kriging
- Focused on core functionality needed for SpotOptim

Parameters:

- `noise`: Regularization (nugget effect)

15. Kriging Surrogate Integration

- `kernel`: Currently 'gauss' (Gaussian/RBF)
- `n_theta`: Number of length scale parameters
- `min_theta`, `max_theta`: Bounds for hyperparameter optimization
- `seed`: Random seed for reproducibility

15.4. Integration with SpotOptim

No Changes Required to SpotOptim Core!

The existing `surrogate` parameter already supports any scikit-learn compatible model:

```
from spotoptim import SpotOptim, Kriging
import numpy as np

def rosenbrock(X):
    """Rosenbrock function"""
    x = X[:, 0]
    y = X[:, 1]
    return (1 - x)**2 + 100 * (y - x**2)**2

kriging = Kriging(seed=42)
optimizer = SpotOptim(
    fun=rosenbrock,
    bounds=[(-2, 2), (-2, 2)],
    surrogate=kriging, # Just pass the Kriging instance
    max_iter=30,
    seed=42
)
result = optimizer.optimize()
print(f"Best value: {result.fun:.6f}")
print(f"Best point: {result.x}")
```

Best value: 0.012619

Best point: [0.92767604 0.86917826]

15.5. Documentation

Added Example to `notebooks/demos.ipynb`

- Demonstrates Kriging vs GP comparison
- Shows custom parameter usage

15.6. Usage Examples

15.6.1. Basic Usage

```
from spotoptim import SpotOptim, Kriging
import numpy as np

def sphere(X):
    """Sphere function:  $f(x) = \sum(x^2)$ """
    return np.sum(X**2, axis=1)

kriging = Kriging(noise=1e-6, seed=42)
optimizer = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    surrogate=kriging,
    max_iter=20,
    seed=42
)
result = optimizer.optimize()
print(f"Best value: {result.fun:.6f}")
print(f"Best point: {result.x}")
```

Best value: 0.000000

Best point: [2.04416182e-05 5.67177247e-04]

15.6.2. Custom Parameters

```
import numpy as np

def ackley(X):
    """Ackley function - multimodal test function"""
    a = 20
    b = 0.2
    c = 2 * np.pi
    n = X.shape[1]

    sum_sq = np.sum(X**2, axis=1)
    sum_cos = np.sum(np.cos(c * X), axis=1)
```

15. Kriging Surrogate Integration

```
        return -a * np.exp(-b * np.sqrt(sum_sq / n)) - np.exp(sum_cos / n) + a + np.e

kriging = Kriging(
    noise=1e-4,
    min_theta=-2.0,
    max_theta=3.0,
    seed=123
)

optimizer = SpotOptim(
    fun=ackley,
    bounds=[(-5, 5), (-5, 5)],
    surrogate=kriging,
    max_iter=40,
    seed=123
)
result = optimizer.optimize()
print(f"Best value: {result.fun:.6f}")
print(f"Best point: {result.x}")
```

Best value: 0.004040
Best point: [0.00140328 -0.00013417]

15.6.3. Prediction with Uncertainty

```
from spotoptim import Kriging
import numpy as np

# Generate training data
np.random.seed(42)
X_train = np.random.uniform(-5, 5, (20, 2))
y_train = np.sum(X_train**2, axis=1)

# Generate test data
X_test = np.random.uniform(-5, 5, (10, 2))

# Fit model and predict with uncertainty
model = Kriging(seed=42)
model.fit(X_train, y_train)
y_pred, y_std = model.predict(X_test, return_std=True)

print(f"Predictions shape: {y_pred.shape}")
```

```
print(f"Uncertainties shape: {y_std.shape}")
print(f"Mean prediction: {y_pred.mean():.4f}")
print(f"Mean uncertainty: {y_std.mean():.4f}")
```

```
Predictions shape: (10,)
Uncertainties shape: (10,)
Mean prediction: 20.8065
Mean uncertainty: 0.0302
```

15.7. Technical Details

15.7.1. Kriging vs GaussianProcessRegressor

Aspect	Kriging	GaussianProcessRegressor
Lines of code	~350	Complex internal implementation
Dependencies	NumPy, SciPy	scikit-learn + dependencies
Kernel	Gaussian (RBF)	Multiple types (Matern, RQ, etc.)
Hyperparameter opt	Differential Evolution	L-BFGS-B with restarts
Use case	Simplified, explicit	Production, flexible

15.7.2. Algorithm

1. Correlation Matrix:

- Compute squared distances: $D_{ij} = \sum_k (x_{ik} - x_{jk})^2$
- Apply kernel: $R_{ij} = \exp(-D_{ij})$
- Add nugget: $R_{ii} += \text{noise}$

2. Maximum Likelihood:

- Optimize σ^2 via differential evolution
- Minimize: $(n/2)\log(\sigma^2) + (1/2)\log|R|$
- Concentrated likelihood (σ^2 profiled out)

3. Prediction:

- Mean: $\hat{f}(x) = \hat{\sigma}^2 + (x) R^{-1} r$
- Variance: $s^2(x) = \hat{\sigma}^2 [1 + (x) R^{-1} (x)]$
- Uses Cholesky decomposition for efficiency

15. Kriging Surrogate Integration

15.7.3. Key Arguments Passed from SpotOptim

SpotOptim passes these to the surrogate via the standard interface:

During fit:

```
# Example of how SpotOptim uses the surrogate internally
surrogate.fit(X, y)
```

- **X**: Training points (`n_initial` or accumulated evaluations)
- **y**: Function values

During predict:

```
# Example of internal usage
mu = surrogate.predict(x)[0] # For acquisition='y'
mu, sigma = surrogate.predict(x, return_std=True) # For acquisition='ei', 'pi'
```

Implicit parameters via seed:

- `random_state=seed` (for `GaussianProcessRegressor`)
- `seed=seed` (for Kriging)

15.8. Benefits

1. **Self-contained**: No heavy scikit-learn dependency for surrogate
2. **Explicit**: Clear hyperparameter bounds and optimization
3. **Educational**: Readable implementation of Kriging/GP
4. **Flexible**: Easy to extend with new kernels or features
5. **Compatible**: Works seamlessly with existing SpotOptim API

15.9. Future Enhancements

Potential additions:

- ☐ Additional kernels (Matern, Exponential, Cubic)
- ☐ Anisotropic hyperparameters (separate per dimension)
- ☐ Gradient-enhanced predictions
- ☐ Batch predictions for efficiency
- ☐ Parallel hyperparameter optimization
- ☐ ARD (Automatic Relevance Determination)

15.10. Jupyter Notebook

i Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

16. Kriging Surrogate Models in SpotOptim

16.1. Introduction

SpotOptim provides a powerful Kriging (Gaussian Process) surrogate model that can significantly enhance optimization performance. This tutorial introduces the **Kriging** class, explains its theory and parameters, and demonstrates how to use it effectively with SpotOptim.

16.1.1. What is Kriging?

Kriging, also known as Gaussian Process regression, is a sophisticated interpolation method that:

- **Predicts function values** at untested points based on observed data
- **Provides uncertainty estimates** indicating confidence in predictions
- **Models smooth functions** common in engineering and scientific optimization
- **Handles noisy observations** through regression approaches

The mathematical foundation of Kriging is based on the assumption that the objective function $f(\mathbf{x})$ can be modeled as:

$$f(\mathbf{x}) = \mu + Z(\mathbf{x})$$

where μ is a constant mean and $Z(\mathbf{x})$ is a Gaussian process with correlation structure. The correlation between two points $\mathbf{x}_i$ and $\mathbf{x}_j$ is typically modeled using a Gaussian (RBF) correlation function:

$$R(\mathbf{x}_i, \mathbf{x}_j) = \exp \left(- \sum_{k=1}^d \theta_k |x_{i,k} - x_{j,k}|^2 \right)$$

where $\theta_k > 0$ are the correlation length parameters that control smoothness in each dimension.

16.1.2. Why Use Kriging in SpotOptim?

1. **Better Surrogate Models:** More accurate predictions than simple interpolation
2. **Uncertainty Quantification:** Know where the model is uncertain
3. **Mixed Variable Types:** Handle continuous, integer, and categorical variables
4. **Multiple Methods:** Choose between interpolation, regression, and reinterpolation
5. **Customizable:** Control regularization, correlation parameters, and more

16.2. Basic Usage

16.2.1. Creating a Simple Kriging Model

Let's start with the most basic usage - creating a Kriging model with default settings:

```
import numpy as np
from spotoptim import SpotOptim
from spotoptim.surrogate import Kriging

# Simple objective function
def sphere(X):
    """Sphere function:  $f(x) = \sum(x^2)$ """
    return np.sum(X**2, axis=1)

# Create Kriging surrogate with defaults
kriging = Kriging(seed=42)

# Use with SpotOptim
optimizer = SpotOptim(
    fun=sphere,
    bounds=[(-5, 5), (-5, 5)],
    surrogate=kriging, # Use Kriging instead of default GP
    max_iter=20,
    n_initial=10,
    seed=42,
    verbose=True
)

result = optimizer.optimize()
print(f"\nBest solution: {result.x}")
print(f"Best value: {result.fun:.6f}")
```



```

TensorBoard logging disabled
Initial best: f(x) = 2.420807
Iteration 1: New best f(x) = 0.001740
Iteration 2: New best f(x) = 0.000372
Iteration 3: New best f(x) = 0.000254
Iteration 4: f(x) = 0.004013
Iteration 5: f(x) = 0.001046
Iteration 6: f(x) = 0.001192
Iteration 7: f(x) = 0.001219
Iteration 8: f(x) = 0.001147
Iteration 9: New best f(x) = 0.000004
Iteration 10: New best f(x) = 0.000000

Best solution: [2.04416182e-05 5.67177247e-04]
Best value: 0.000000

```

16.2.2. Default vs Custom Surrogate

SpotOptim uses a Gaussian Process Regressor by default. Here's how Kriging compares:

```

import numpy as np
from spotoptim import SpotOptim
from spotoptim.surrogate import Kriging

def rosenbrock(X):
    """Rosenbrock function: (1-x)^2 + 100(y-x^2)^2"""
    x = X[:, 0]
    y = X[:, 1]
    return (1 - x)**2 + 100 * (y - x**2)**2

# With default Gaussian Process
opt_default = SpotOptim(
    fun=rosenbrock,
    bounds=[(-2, 2), (-2, 2)],
    max_iter=30,
    n_initial=15,
    seed=42,
    verbose=False
)
result_default = opt_default.optimize()

# With Kriging surrogate
kriging = Kriging(method='regression', seed=42)

```

```
opt_kriging = SpotOptim(
    fun=rosenbrock,
    bounds=[(-2, 2), (-2, 2)],
    surrogate=kriging,
    max_iter=30,
    n_initial=15,
    seed=42,
    verbose=False
)
result_kriging = opt_kriging.optimize()

print("Comparison:")
print(f"Default GP: f(x) = {result_default.fun:.6f}")
print(f"Kriging: f(x) = {result_kriging.fun:.6f}")
```

```
Comparison:
Default GP: f(x) = 0.190180
Kriging: f(x) = 0.001073
```

Both approaches work well, but Kriging offers more control over the surrogate behavior.

16.3. Understanding Kriging Parameters

16.3.1. Method: Interpolation vs Regression

The `method` parameter controls how Kriging handles data:

- “**interpolation**”: Exact interpolation, passes through all training points
- “**regression**”: Allows smoothing, better for noisy data
- “**reinterpolation**”: Hybrid approach

```
import numpy as np
from spotoptim.surrogate import Kriging
import matplotlib.pyplot as plt

# Create noisy training data
np.random.seed(42)
X_train = np.linspace(0, 2*np.pi, 10).reshape(-1, 1)
y_train = np.sin(X_train.ravel()) + 0.1 * np.random.randn(10)

# Test data for smooth predictions
```

16.3. Understanding Kriging Parameters

```
X_test = np.linspace(0, 2*np.pi, 100).reshape(-1, 1)
y_true = np.sin(X_test.ravel())

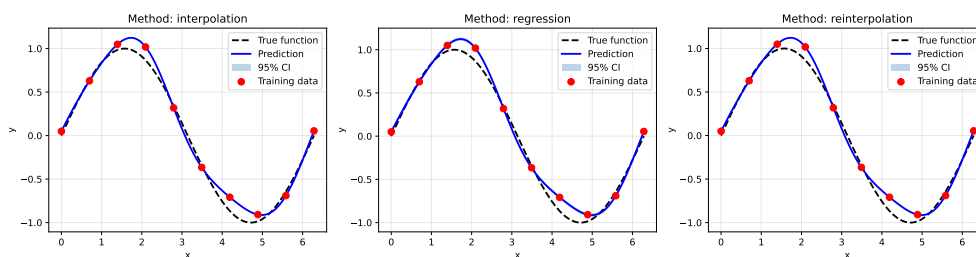
# Compare methods
methods = ['interpolation', 'regression', 'reinterpolation']
fig, axes = plt.subplots(1, 3, figsize=(15, 4))

for ax, method in zip(axes, methods):
    model = Kriging(method=method, seed=42, model_fun_evals=50)
    model.fit(X_train, y_train)
    y_pred, y_std = model.predict(X_test, return_std=True)

    # Plot
    ax.plot(X_test, y_true, 'k--', label='True function', linewidth=2)
    ax.plot(X_test, y_pred, 'b-', label='Prediction', linewidth=2)
    ax.fill_between(X_test.ravel(),
                   y_pred - 2*y_std,
                   y_pred + 2*y_std,
                   alpha=0.3, label='95% CI')
    ax.scatter(X_train, y_train, c='r', s=50, zorder=5, label='Training data')
    ax.set_title(f'Method: {method}')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.legend()
    ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print("Key differences:")
print("- Interpolation: Passes exactly through training points")
print("- Regression: Smooths over noisy data (recommended)")
print("- Reinterpolation: Balances between the two")
```



Key differences:

16. Kriging Surrogate Models in SpotOptim

- Interpolation: Passes exactly through training points
- Regression: Smooths over noisy data (recommended)
- Reinterpolation: Balances between the two

Recommendation: Use `method='regression'` for most optimization problems, especially with noisy objectives.

16.3.2. Noise Parameter and Regularization

The noise parameter adds a small nugget effect for numerical stability:

```
import numpy as np
from spotoptim import SpotOptim
from spotoptim.surrogate import Kriging

def noisy_ackley(X):
    """Ackley function with observation noise"""
    a = 20
    b = 0.2
    c = 2 * np.pi
    n = X.shape[1]

    sum_sq = np.sum(X**2, axis=1)
    sum_cos = np.sum(np.cos(c * X), axis=1)

    base = -a * np.exp(-b * np.sqrt(sum_sq / n)) - np.exp(sum_cos / n) + a + np.e
    noise = np.random.normal(0, 0.5, size=base.shape) # Add noise
    return base + noise

# Very small noise (near interpolation)
kriging_small = Kriging(noise=1e-10, method='interpolation', seed=42)
opt_small = SpotOptim(
    fun=noisy_ackley,
    bounds=[(-5, 5), (-5, 5)],
    surrogate=kriging_small,
    max_iter=25,
    n_initial=12,
    seed=42,
    verbose=False
)
result_small = opt_small.optimize()

# Larger noise (more robust to noise)
```

```

kriging_large = Kriging(noise=0.01, method='interpolation', seed=42)
opt_large = SpotOptim(
    fun=noisy_ackley,
    bounds=[(-5, 5), (-5, 5)],
    surrogate=kriging_large,
    max_iter=25,
    n_initial=12,
    seed=42,
    verbose=False
)
result_large = opt_large.optimize()

print("Impact of noise parameter:")
print(f"Small noise (1e-10): f(x) = {result_small.fun:.6f}")
print(f"Large noise (0.01): f(x) = {result_large.fun:.6f}")
print("\nNote: For noisy functions, use 'regression' method instead,")
print("      which automatically optimizes the Lambda (nugget) parameter.")

```

Impact of noise parameter:

Small noise (1e-10): f(x) = 0.049827

Large noise (0.01): f(x) = 0.012636

Note: For noisy functions, use 'regression' method instead,
which automatically optimizes the Lambda (nugget) parameter.

Best Practice: For noisy functions, use method='regression' which automatically optimizes the regularization parameter instead of fixing it.

16.3.3. Correlation Length Parameters (Theta)

The `theta` parameters control smoothness. SpotOptim optimizes these automatically:

- **min_theta:** Minimum log () value (default: -3.0 → = 0.001)
- **max_theta:** Maximum log () value (default: 2.0 → = 100)

Smaller → smoother function, larger → more wiggly function.

```

import numpy as np
from spotoptim.surrogate import Kriging

# Sample data
X_train = np.array([[0.0], [0.5], [1.0], [1.5], [2.0]])
y_train = np.sin(X_train.ravel())

```

16. Kriging Surrogate Models in SpotOptim

```
# Tight bounds (smoother)
model_smooth = Kriging(min_theta=-1.0, max_theta=0.0, seed=42, model_fun_evals=30)
model_smooth.fit(X_train, y_train)

# Wide bounds (more flexible)
model_flexible = Kriging(min_theta=-3.0, max_theta=2.0, seed=42, model_fun_evals=30)
model_flexible.fit(X_train, y_train)

print("Theta bounds and optimal values:")
print(f"Smooth model:   bounds=[-1.0, 0.0], theta={model_smooth.theta_}")
print(f"Flexible model: bounds=[-3.0, 2.0], theta={model_flexible.theta_}")
print("\nThe optimizer automatically finds the best theta within the bounds.")
```

Theta bounds and optimal values:

Smooth model: bounds=[-1.0, 0.0], theta=[-0.95216449]

Flexible model: bounds=[-3.0, 2.0], theta=[-0.95309049]

The optimizer automatically finds the best theta within the bounds.

Default Values: The defaults `min_theta=-3.0` and `max_theta=2.0` work well for most problems.

16.3.4. Isotropic vs Anisotropic Correlation

- **Isotropic** (`isotropic=True`): Single `theta` for all dimensions (faster, fewer parameters)
- **Anisotropic** (`isotropic=False`): Different `theta` per dimension (more flexible, default)

```
import numpy as np
from spotoptim.surrogate import Kriging

# 3D problem
np.random.seed(42)
X_train = np.random.rand(15, 3) * 10 - 5
y_train = X_train[:, 0]**2 + 0.1*X_train[:, 1]**2 + 2*X_train[:, 2]**2

# Isotropic: one theta for all dimensions
model_iso = Kriging(isotropic=True, seed=42, model_fun_evals=40)
model_iso.fit(X_train, y_train)

# Anisotropic: different theta per dimension
```

```

model_aniso = Kriging(isotropic=False, seed=42, model_fun_evals=40)
model_aniso.fit(X_train, y_train)

print("Correlation structure:")
print(f"Isotropic:   n_theta={model_iso.n_theta}, theta={model_iso.theta}")
print(f"Anisotropic: n_theta={model_aniso.n_theta}, theta={model_aniso.theta}")
print("\nAnisotropic can capture different smoothness in each dimension.")

# Test predictions
X_test = np.array([[0.0, 0.0, 0.0]])
y_iso = model_iso.predict(X_test)
y_aniso = model_aniso.predict(X_test)
print(f"\nPrediction at origin:")
print(f"Isotropic:   {y_iso[0]:.4f}")
print(f"Anisotropic: {y_aniso[0]:.4f}")

```

```

Correlation structure:
Isotropic:   n_theta=1, theta=[-3.]
Anisotropic: n_theta=3, theta=[-2.23292358 -3.          -1.97228776]

Anisotropic can capture different smoothness in each dimension.

Prediction at origin:
Isotropic:   -0.0666
Anisotropic: -1.0267

```

When to Use:

- Use **isotropic** for faster fitting when dimensions have similar characteristics
- Use **anisotropic** (default) when dimensions behave differently

16.4. Advanced Features

16.4.1. Mixed Variable Types

Kriging supports mixed variable types: continuous (`float`), integer (`int`), and categorical (`factor`):

```

import numpy as np
from spotoptim import SpotOptim
from spotoptim.surrogate import Kriging

```

16. Kriging Surrogate Models in SpotOptim

```
def mixed_objective(X):
    """
    Optimize a system with:
    - x0: continuous learning rate (0.001 to 0.1)
    - x1: integer number of layers (1 to 5)
    - x2: categorical activation (0=ReLU, 1=Tanh, 2=Sigmoid)
    """
    X = np.atleast_2d(X)
    learning_rate = X[:, 0]
    n_layers = X[:, 1]
    activation = X[:, 2]

    # Simplified model: prefer lr~0.01, 3 layers, ReLU
    loss = (learning_rate - 0.01)**2 + (n_layers - 3)**2
    loss += np.where(activation == 0, 0.0, # ReLU is best
                    np.where(activation == 1, 0.5, # Tanh is ok
                             1.0)) # Sigmoid is worst

    return loss

# Define bounds and types
bounds = [
    (0.001, 0.1), # learning_rate (float)
    (1, 5),      # n_layers (int)
    (0, 2)       # activation (factor: 0, 1, or 2)
]

var_type = ['float', 'int', 'factor']

# Create Kriging with mixed types
kriging_mixed = Kriging(
    method='regression',
    var_type=var_type,
    seed=42,
    model_fun_evals=50
)

# Optimize
optimizer = SpotOptim(
    fun=mixed_objective,
    bounds=bounds,
    var_type=var_type,
    surrogate=kriging_mixed,
    max_iter=30,
    n_initial=15,
```



```

    seed=42,
    verbose=True
)

result = optimizer.optimize()

print(f"\nOptimal configuration:")
print(f"Learning rate: {result.x[0]:.4f}")
print(f"Num layers:      {int(result.x[1])}")
print(f"Activation:        {int(result.x[2])} (0=ReLU, 1=Tanh, 2=Sigmoid)")
print(f"Loss:              {result.fun:.6f}")

```

```

TensorBoard logging disabled
Initial best: f(x) = 0.000173
Iteration 1: New best f(x) = 0.000081
  Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 2: f(x) = 1.507094
  Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 3: f(x) = 0.000599
  Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 4: f(x) = 0.000587
  Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 5: f(x) = 5.005398
  Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 6: f(x) = 1.000342
  Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 7: f(x) = 1.501632
  Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 8: f(x) = 0.502100
  Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 9: f(x) = 2.000762
  Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.
Iteration 10: f(x) = 4.006134
  Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.

```

16. Kriging Surrogate Models in SpotOptim

Iteration 11: $f(x) = 1.000430$

Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.

Iteration 12: $f(x) = 1.500167$

Iteration 13: New best $f(x) = 0.000030$

Iteration 14: $f(x) = 0.000049$

Iteration 15: New best $f(x) = 0.000003$

Optimal configuration:

Learning rate: 0.0082

Num layers: 3

Activation: 0 (0=ReLU, 1=Tanh, 2=Sigmoid)

Loss: 0.000003

Key Points:

- Set `var_type` in both Kriging and SpotOptim
- Factor variables use specialized distance metrics (default: Canberra)
- Integer variables are treated as ordered but discrete

16.4.2. Customizing the Distance Metric for Factors

For categorical variables, you can choose different distance metrics.

```
import numpy as np
from spotoptim.surrogate import Kriging

# Categorical data: 2 factor variables
X_train = np.array([
    [0, 0], # Category A, Color Red
    [1, 0], # Category B, Color Red
    [0, 1], # Category A, Color Blue
    [1, 1], # Category B, Color Blue
    [2, 2], # Category C, Color Green
])
y_train = np.array([1.0, 1.5, 1.2, 2.0, 3.5])

# Different distance metrics
metrics = ['canberra', 'hamming']

for metric in metrics:
    model = Kriging(
        method='regression',
        var_type=['factor', 'factor'],
```

```

        metric_factorial=metric,
        seed=42,
        model_fun_evals=40
    )
    model.fit(X_train, y_train)

    # Test prediction
    X_test = np.array([[1, 1]])
    y_pred = model.predict(X_test)

    print(f"Metric: {metric:10s} | Prediction at [1,1]: {y_pred[0]:.4f}")

print("\nAvailable metrics: 'canberra' (default), 'hamming', 'jaccard', etc.")

```

```

Metric: canberra      | Prediction at [1,1]: 1.9463
Metric: hamming       | Prediction at [1,1]: 2.0000

```

```

Available metrics: 'canberra' (default), 'hamming', 'jaccard', etc.

```

16.4.3. Handling High-Dimensional Problems

For high-dimensional problems, Kriging can become computationally expensive. Here are strategies:

```

import numpy as np
from spotoptim import SpotOptim
from spotoptim.surrogate import Kriging

def high_dim_sphere(X):
    """10-dimensional sphere function"""
    return np.sum(X**2, axis=1)

# Strategy 1: Use isotropic correlation (fewer parameters)
kriging_iso = Kriging(
    method='regression',
    isotropic=True, # Single theta for all 10 dimensions
    seed=42,
    model_fun_evals=50 # Faster optimization
)

optimizer_iso = SpotOptim(
    fun=high_dim_sphere,
    bounds=[(-5, 5)] * 10, # 10 dimensions

```

16. Kriging Surrogate Models in SpotOptim

```
surrogate=kriging_iso,
max_iter=50,
n_initial=30,
seed=42,
verbose=False
)

result_iso = optimizer_iso.optimize()

# Strategy 2: Use max_surrogate_points to limit training set size
kriging_limited = Kriging(method='regression', seed=42)

optimizer_limited = SpotOptim(
    fun=high_dim_sphere,
    bounds=[(-5, 5)] * 10,
    surrogate=kriging_limited,
    max_iter=50,
    n_initial=30,
    max_surrogate_points=100, # Limit surrogate training set
    seed=42,
    verbose=False
)

result_limited = optimizer_limited.optimize()

print("High-dimensional optimization strategies:")
print(f"Isotropic Kriging:      f(x) = {result_iso.fun:.6f}")
print(f"Limited training set:  f(x) = {result_limited.fun:.6f}")
print("\nFor >5 dimensions, consider isotropic=True or limit training set.")
```

```
High-dimensional optimization strategies:
Isotropic Kriging:      f(x) = 0.003979
Limited training set:  f(x) = 0.044950
```

For >5 dimensions, consider isotropic=True or limit training set.

16.4.4. Uncertainty Quantification

Kriging provides uncertainty estimates, useful for exploration vs exploitation:

```
import numpy as np
from spotoptim.surrogate import Kriging
import matplotlib.pyplot as plt
```

```

# 1D example for visualization
np.random.seed(42)
X_train = np.array([[0.0], [1.0], [3.0], [5.0], [6.0]])
y_train = np.sin(X_train.ravel()) + 0.05 * np.random.randn(5)

# Fit Kriging model
model = Kriging(method='regression', seed=42, model_fun_evals=50)
model.fit(X_train, y_train)

# Dense test points
X_test = np.linspace(-0.5, 6.5, 200).reshape(-1, 1)
y_pred, y_std = model.predict(X_test, return_std=True)

# True function
y_true = np.sin(X_test.ravel())

# Plot
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(X_test, y_true, 'k--', label='True function', linewidth=2)
plt.plot(X_test, y_pred, 'b-', label='Kriging prediction', linewidth=2)
plt.fill_between(X_test.ravel(),
                 y_pred - 2*y_std,
                 y_pred + 2*y_std,
                 alpha=0.3, color='blue', label='95% confidence')
plt.scatter(X_train, y_train, c='red', s=100, zorder=5,
            edgecolors='black', label='Training points')
plt.xlabel('x', fontsize=12)
plt.ylabel('y', fontsize=12)
plt.title('Kriging Predictions with Uncertainty', fontsize=14)
plt.legend()
plt.grid(True, alpha=0.3)

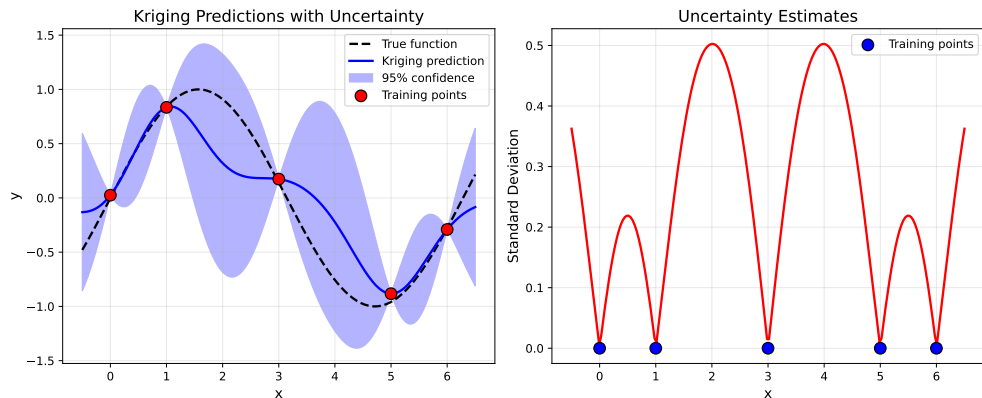
plt.subplot(1, 2, 2)
plt.plot(X_test, y_std, 'r-', linewidth=2)
plt.scatter(X_train, np.zeros_like(X_train), c='blue', s=100,
            zorder=5, edgecolors='black', label='Training points')
plt.xlabel('x', fontsize=12)
plt.ylabel('Standard Deviation', fontsize=12)
plt.title('Uncertainty Estimates', fontsize=14)
plt.legend()
plt.grid(True, alpha=0.3)

```

16. Kriging Surrogate Models in SpotOptim

```
plt.tight_layout()
plt.show()

print("Key observations:")
print("- Uncertainty is low near training points")
print("- Uncertainty is high far from training data")
print("- This guides acquisition functions (e.g., EI exploits low uncertainty)")
```



Key observations:

- Uncertainty is low near training points
- Uncertainty is high far from training data
- This guides acquisition functions (e.g., EI exploits low uncertainty)

16.5. Practical Examples

16.5.1. Example 1: Optimizing the Rastrigin Function

The Rastrigin function is highly multimodal - a challenging test case:

```
import numpy as np
from spotoptim import SpotOptim
from spotoptim.surrogate import Kriging

def rastrigin(X):
    """
    Rastrigin function: highly multimodal
    Global minimum: f(0,...,0) = 0
    """
```

```

A = 10
n = X.shape[1]
return A * n + np.sum(X**2 - A * np.cos(2 * np.pi * X), axis=1)

# Configure Kriging for multimodal function
kriging = Kriging(
    method='regression',      # Smooth over local variations
    min_theta=-3.0,          # Allow flexible correlation
    max_theta=2.0,
    seed=42,
    model_fun_evals=100      # More budget for better surrogate
)

optimizer = SpotOptim(
    fun=rastrigin,
    bounds=[(-5.12, 5.12), (-5.12, 5.12)],
    surrogate=kriging,
    acquisition='ei',        # Expected Improvement for exploration
    max_iter=60,
    n_initial=30,
    seed=42,
    verbose=True
)

result = optimizer.optimize()

print(f"\nRastrigin optimization:")
print(f"Best solution: {result.x}")
print(f"Best value:      {result.fun:.6f} (global optimum: 0.0)")
print(f"Distance to optimum: {np.linalg.norm(result.x):.6f}")

```

```

TensorBoard logging disabled
Initial best: f(x) = 6.442225
Iteration 1: f(x) = 28.951536
Iteration 2: f(x) = 22.875749
Iteration 3: f(x) = 21.815864
Iteration 4: f(x) = 7.077873
Iteration 5: f(x) = 13.368250
Iteration 6: f(x) = 16.577976
Iteration 7: f(x) = 25.780045
Iteration 8: f(x) = 10.034344
Iteration 9: f(x) = 19.560557
Iteration 10: f(x) = 12.381235
Iteration 11: f(x) = 20.126829

```

16. Kriging Surrogate Models in SpotOptim

```
Iteration 12: f(x) = 27.151349
Iteration 13: New best f(x) = 2.290927
Iteration 14: f(x) = 2.761961
Iteration 15: f(x) = 12.468354
Iteration 16: New best f(x) = 1.208377
Iteration 17: f(x) = 12.566211
Iteration 18: f(x) = 14.965620
Iteration 19: f(x) = 14.295554
Iteration 20: f(x) = 28.107905
Iteration 21: f(x) = 22.687907
Iteration 22: f(x) = 43.729592
Iteration 23: f(x) = 41.636148
Iteration 24: New best f(x) = 1.034573
Iteration 25: f(x) = 3.228327
Iteration 26: f(x) = 7.934136
Iteration 27: f(x) = 21.994039
Iteration 28: f(x) = 27.510661
Iteration 29: f(x) = 9.365219
Iteration 30: f(x) = 1.248535
```

Rastrigin optimization:

```
Best solution: [-0.00987679  1.00506675]
Best value:    1.034573 (global optimum: 0.0)
Distance to optimum: 1.005115
```

16.5.2. Example 2: Robust Optimization with Noise

When optimizing noisy functions, Kriging's regression mode helps:

```
import numpy as np
from spotoptim import SpotOptim
from spotoptim.surrogate import Kriging

def noisy_beale(X):
    """
    Beale function with Gaussian noise
    Global minimum: f(3, 0.5) = 0
    """
    x = X[:, 0]
    y = X[:, 1]

    term1 = (1.5 - x + x * y)**2
    term2 = (2.25 - x + x * y**2)**2
    term3 = (2.625 - x + x * y**3)**2
```



```

    base = term1 + term2 + term3
    noise = np.random.normal(0, 0.5, size=base.shape)

    return base + noise

# Use regression method for noisy objectives
kriging_robust = Kriging(
    method='regression',          # Automatically optimizes Lambda (nugget)
    min_Lambda=-9.0,             # Allow small regularization
    max_Lambda=-2.0,             # But not too large
    seed=42,
    model_fun_evals=80
)

# Use repeated evaluations for noise handling
optimizer = SpotOptim(
    fun=noisy_beale,
    bounds=[(-4.5, 4.5), (-4.5, 4.5)],
    surrogate=kriging_robust,
    max_iter=50,
    n_initial=20,
    repeats_initial=3,           # Evaluate each initial point 3 times
    repeats_surrogate=2,        # Evaluate each new point 2 times
    seed=42,
    verbose=True
)

result = optimizer.optimize()

print(f"\nNoisy Beale optimization:")
print(f"Best solution: {result.x}")
print(f"Best mean value: {optimizer.min_mean_y:.6f}")
print(f"Variance at best: {optimizer.min_var_y:.6f}")
print(f"True optimum: [3.0, 0.5]")
print(f"Distance: {np.linalg.norm(result.x - np.array([3.0, 0.5])):.4f}")

```

TensorBoard logging disabled

Initial best: $f(x) = 9.379814$, mean best: $f(x) = 9.729582$

Noisy Beale optimization:

Best solution: [0.34789079 -0.3014163]

Best mean value: 9.729582

Variance at best: 0.214586

16. Kriging Surrogate Models in SpotOptim

True optimum: [3.0, 0.5]
Distance: 2.7706

16.5.3. Example 3: Real-World Machine Learning Hyperparameter Tuning

Optimize hyperparameters for a neural network:

```
import numpy as np
from spotoptim import SpotOptim
from spotoptim.surrogate import Kriging

def ml_objective(X):
    """
    Simulate training a neural network
    Variables:
    - x0: learning_rate (log scale: 1e-4 to 1e-1)
    - x1: num_layers (integer: 1 to 5)
    - x2: hidden_size (integer: 16, 32, 64, 128, 256)
    - x3: dropout_rate (float: 0.0 to 0.5)

    Returns validation error
    """
    X = np.atleast_2d(X)
    lr = X[:, 0]
    n_layers = X[:, 1]
    hidden_size = X[:, 2]
    dropout = X[:, 3]

    # Simulate validation error (lower is better)
    # Optimal around: lr=0.001, 3 layers, 64 hidden, 0.2 dropout
    error = (np.log10(lr) + 3)**2 # Prefer lr ~ 0.001
    error += (n_layers - 3)**2 # Prefer 3 layers
    error += ((hidden_size - 64) / 32)**2 # Prefer 64 hidden units
    error += (dropout - 0.2)**2 * 10 # Prefer 0.2 dropout

    # Add some noise to simulate training variance
    error += np.random.normal(0, 0.1, size=error.shape)

    return error

# Setup optimization
bounds = [
```

```

    (1e-4, 1e-1), # learning_rate
    (1, 5),       # num_layers
    (16, 256),    # hidden_size
    (0.0, 0.5)    # dropout_rate
]

var_type = ['float', 'int', 'int', 'float']

# Use Kriging with mixed types
kriging_ml = Kriging(
    method='regression',
    var_type=var_type,
    seed=42,
    model_fun_evals=60
)

optimizer = SpotOptim(
    fun=ml_objective,
    bounds=bounds,
    var_type=var_type,
    var_trans=['log10', None, None, None], # Log scale for learning rate
    surrogate=kriging_ml,
    max_iter=40,
    n_initial=20,
    seed=42,
    verbose=True
)

result = optimizer.optimize()

print(f"\nOptimal hyperparameters:")
print(f"Learning rate: {result.x[0]:.6f}")
print(f"Num layers:    {int(result.x[1])}")
print(f"Hidden size:    {int(result.x[2])}")
print(f"Dropout rate:   {result.x[3]:.3f}")
print(f"Validation error: {result.fun:.6f}")

```

```

TensorBoard logging disabled
Initial best: f(x) = 1.133527
Iteration 1: f(x) = 5.122533
Iteration 2: f(x) = 5.959913
Iteration 3: f(x) = 1.505793
Iteration 4: New best f(x) = 0.942988
Iteration 5: New best f(x) = 0.280219

```

16. Kriging Surrogate Models in SpotOptim

```
Iteration 6: New best f(x) = 0.260858
Iteration 7: f(x) = 1.115300
Iteration 8: f(x) = 0.653129
Iteration 9: New best f(x) = 0.176351
Iteration 10: New best f(x) = 0.084523
Iteration 11: New best f(x) = -0.067705
Iteration 12: f(x) = 0.167623
Iteration 13: f(x) = 0.048797
Iteration 14: f(x) = 0.053888
Iteration 15: f(x) = 0.015810
Iteration 16: New best f(x) = -0.108042
Iteration 17: f(x) = -0.007253
Iteration 18: f(x) = -0.003508
Iteration 19: f(x) = -0.052430
Iteration 20: f(x) = 0.006251
```

```
Optimal hyperparameters:
Learning rate: 0.000987
Num layers:    3
Hidden size:   68
Dropout rate:  0.158
Validation error: -0.108042
```

16.5.4. Example 4: Comparing Kriging Methods

Let's compare all three Kriging methods on the same problem:

```
import numpy as np
from spotoptim import SpotOptim
from spotoptim.surrogate import Kriging

def levy(X):
    """
    Levy function N. 13
    Global minimum: f(1, 1) = 0
    """
    x = X[:, 0]
    y = X[:, 1]

    w1 = 1 + (x - 1) / 4
    w2 = 1 + (y - 1) / 4

    term1 = np.sin(np.pi * w1)**2
    term2 = (w1 - 1)**2 * (1 + 10 * np.sin(np.pi * w1 + 1)**2)
```

```

term3 = (w2 - 1)**2 * (1 + np.sin(2 * np.pi * w2)**2)

return term1 + term2 + term3

methods = ['interpolation', 'regression', 'reinterpolation']
results_methods = []

for method in methods:
    kriging = Kriging(
        method=method,
        seed=42,
        model_fun_evals=60
    )

    optimizer = SpotOptim(
        fun=levy,
        bounds=[(-10, 10), (-10, 10)],
        surrogate=kriging,
        max_iter=40,
        n_initial=20,
        seed=42,
        verbose=False
    )

    result = optimizer.optimize()
    results_methods.append((method, result.fun, result.x))

    print(f"{method:15s} | f(x) = {result.fun:8.6f} | x = {result.x}")

print(f"\nGlobal optimum: f(1, 1) = 0.0")
print("\nAll methods find good solutions, but 'regression' is most robust.")

```

```

interpolation    | f(x) = 0.001657 | x = [0.98535016 0.85317084]
regression       | f(x) = 0.027480 | x = [0.8385841 1.068918 ]
reinterpolation | f(x) = 0.027480 | x = [0.8385841 1.068918 ]

```

Global optimum: f(1, 1) = 0.0

All methods find good solutions, but 'regression' is most robust.

16.5.5. Example 5: Sensitivity to Theta Bounds

Theta bounds control the range of smoothness. Let's see their impact:

16. Kriging Surrogate Models in SpotOptim

```
import numpy as np
from spotoptim import SpotOptim
from spotoptim.surrogate import Kriging

def griewank(X):
    """Griewank function"""
    sum_sq = np.sum(X**2 / 4000, axis=1)
    prod_cos = np.prod(np.cos(X / np.sqrt(np.arange(1, X.shape[1] + 1))), axis=1)
    return sum_sq - prod_cos + 1

# Different theta bounds
theta_configs = [
    (-2.0, 1.0, "Tight (smoother)"),
    (-3.0, 2.0, "Default"),
    (-4.0, 3.0, "Wide (more flexible)")
]

for min_theta, max_theta, label in theta_configs:
    kriging = Kriging(
        method='regression',
        min_theta=min_theta,
        max_theta=max_theta,
        seed=42,
        model_fun_evals=50
    )

    optimizer = SpotOptim(
        fun=griewank,
        bounds=[(-600, 600), (-600, 600)],
        surrogate=kriging,
        max_iter=35,
        n_initial=18,
        seed=42,
        verbose=False
    )

    result = optimizer.optimize()
    print(f"{label:20s} [{min_theta:5.1f}, {max_theta:4.1f}] | f(x) = {result.fun:8.6f}")

print("\nDefault bounds work well for most problems.")
```

```
Tight (smoother)      [ -2.0,  1.0] | f(x) = 8.578706
Default               [ -3.0,  2.0] | f(x) = 13.108634
Wide (more flexible) [ -4.0,  3.0] | f(x) = 2.652215
```

Default bounds work well for most problems.

16.6. Comparing Surrogates

16.6.1. Kriging vs Gaussian Process vs Random Forest

Let's compare different surrogate models:

```
import numpy as np
from spotoptim import SpotOptim
from spotoptim.surrogate import Kriging, SimpleKriging
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import Matern, ConstantKernel
from sklearn.ensemble import RandomForestRegressor

def schwefel(X):
    """Schwefel function"""
    return 418.9829 * X.shape[1] - np.sum(X * np.sin(np.sqrt(np.abs(X)))), axis=1)

bounds = [(-500, 500), (-500, 500)]

# 1. Kriging (full-featured)
kriging = Kriging(method='regression', seed=42, model_fun_evals=60)
opt_kriging = SpotOptim(fun=schwefel, bounds=bounds, surrogate=kriging,
                       max_iter=35, n_initial=18, seed=42, verbose=False)
result_kriging = opt_kriging.optimize()

# 2. SimpleKriging (lightweight)
simple_kriging = SimpleKriging(noise=1e-10, seed=42)
opt_simple = SpotOptim(fun=schwefel, bounds=bounds, surrogate=simple_kriging,
                      max_iter=35, n_initial=18, seed=42, verbose=False)
result_simple = opt_simple.optimize()

# 3. Gaussian Process (sklearn default)
kernel = ConstantKernel(1.0) * Matern(length_scale=1.0, nu=2.5)
gp = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=10,
                             normalize_y=True, random_state=42)
opt_gp = SpotOptim(fun=schwefel, bounds=bounds, surrogate=gp,
                  max_iter=35, n_initial=18, seed=42, verbose=False)
result_gp = opt_gp.optimize()

# 4. Random Forest
```

16. Kriging Surrogate Models in SpotOptim

```
rf = RandomForestRegressor(n_estimators=100, random_state=42)
opt_rf = SpotOptim(fun=schwefel, bounds=bounds, surrogate=rf,
                  max_iter=35, n_initial=18, seed=42, verbose=False)
result_rf = opt_rf.optimize()

print("Surrogate Model Comparison:")
print(f"Kriging (full):      f(x) = {result_kriging.fun:8.2f}")
print(f"SimpleKriging:      f(x) = {result_simple.fun:8.2f}")
print(f"Gaussian Process:   f(x) = {result_gp.fun:8.2f}")
print(f"Random Forest:      f(x) = {result_rf.fun:8.2f}")
print(f"\nGlobal optimum: f(420.9687, 420.9687) = 0.0")
print("\nKriging offers:")
print("- Mixed variable types (continuous, integer, categorical)")
print("- Multiple methods (interpolation, regression, reinterpolation)")
print("- Explicit control over regularization and correlation")
```

Surrogate Model Comparison:

Kriging (full):	f(x) =	118.47
SimpleKriging:	f(x) =	118.44
Gaussian Process:	f(x) =	118.44
Random Forest:	f(x) =	184.74

Global optimum: f(420.9687, 420.9687) = 0.0

Kriging offers:

- Mixed variable types (continuous, integer, categorical)
- Multiple methods (interpolation, regression, reinterpolation)
- Explicit control over regularization and correlation

16.7. Best Practices

16.7.1. 1. Choosing the Right Method

```
# For smooth, deterministic functions
kriging = Kriging(method='interpolation', noise=1e-10, seed=42)

# For general optimization (RECOMMENDED)
kriging = Kriging(method='regression', seed=42)

# For noisy functions with repeated evaluations
```



```
kriging = Kriging(method='regression', seed=42)
# Use with repeats_initial and repeats_surrogate in SpotOptim
```

16.7.2. 2. Setting Model Complexity

```
# For low-dimensional problems (<5D)
kriging = Kriging(
    method='regression',
    isotropic=False,          # Anisotropic (default)
    model_fun_evals=100,      # More budget for better fit
    seed=42
)

# For high-dimensional problems (>5D)
kriging = Kriging(
    method='regression',
    isotropic=True,           # Fewer parameters
    model_fun_evals=50,       # Faster fitting
    seed=42
)
```

16.7.3. 3. Handling Different Variable Types

```
# Mixed types example
bounds = [
    (0.0, 10.0),    # continuous
    (1, 10),        # integer
    (0, 3)          # categorical (4 categories)
]

var_type = ['float', 'int', 'factor']

kriging = Kriging(
    method='regression',
    var_type=var_type,
    metric_factorial='canberra', # For factor variables
    seed=42
)

optimizer = SpotOptim(
```

16. Kriging Surrogate Models in SpotOptim

```
    fun=objective,  
    bounds=bounds,  
    var_type=var_type,  
    surrogate=kriging,  
    seed=42  
)
```

16.7.4. 4. Reproducibility

Always set the seed for reproducible results:

```
# Both Kriging and SpotOptim should have seeds  
kriging = Kriging(method='regression', seed=42)  
  
optimizer = SpotOptim(  
    fun=objective,  
    bounds=bounds,  
    surrogate=kriging,  
    seed=42 # Same seed or different seed  
)
```

16.7.5. 5. Monitoring Surrogate Quality

Check the negative log-likelihood after fitting:

```
import numpy as np  
from spotoptim.surrogate import Kriging  
  
# Sample data  
X = np.random.rand(20, 2) * 10 - 5  
y = np.sum(X**2, axis=1)  
  
model = Kriging(method='regression', seed=42)  
model.fit(X, y)  
  
print(f"Negative log-likelihood: {model.negLnLike:.4f}")  
print(f"Optimized theta: {model.theta_}")  
print(f"Optimized Lambda: {model.Lambda_:.4f}")  
print(f"Condition number of Psi: {model.cnd_Psi:.2e}")  
  
print("\nLower negLnLike = better fit")  
print("Check condition number for numerical stability")
```

```
Negative log-likelihood: -15.3998
Optimized theta: [-3. -3.]
Optimized Lambda: -8.9957
Condition number of Psi: 4.03e+13
```

Lower negLnLike = better fit
Check condition number for numerical stability

16.8. Common Issues and Solutions

16.8.1. Issue 1: Slow Fitting for Large Datasets

Problem: Kriging becomes slow with many training points.

Solution: Limit surrogate training set size:

```
# Use max_surrogate_points in SpotOptim
optimizer = SpotOptim(
    fun=objective,
    bounds=bounds,
    surrogate=kriging,
    max_surrogate_points=200, # Limit to 200 points
    selection_method='distant', # or 'best'
    seed=42
)
```

16.8.2. Issue 2: Poor Predictions for Categorical Variables

Problem: Kriging doesn't handle factors well.

Solution: Try different distance metrics:

```
# Try different metrics
for metric in ['canberra', 'hamming', 'jaccard']:
    kriging = Kriging(
        method='regression',
        var_type=['float', 'factor'],
        metric_factorial=metric,
        seed=42
    )
# Test performance
```

16.8.3. Issue 3: Numerical Instability

Problem: Correlation matrix is nearly singular.

Solution: Increase regularization:

```
# For interpolation method
kriging = Kriging(
    method='interpolation',
    noise=1e-6, # Increase from default 1e-8
    seed=42
)

# Or use regression method (recommended)
kriging = Kriging(
    method='regression',
    min_Lambda=-6.0, # Don't go too small
    seed=42
)
```

16.8.4. Issue 4: Overfitting to Noisy Data

Problem: Kriging fits noise instead of underlying function.

Solution: Use regression method with reasonable Lambda bounds:

```
kriging = Kriging(
    method='regression',
    min_Lambda=-5.0, # Not too small
    max_Lambda=-1.0, # Not too large
    seed=42
)
```

16.9. Summary

16.9.1. Key Takeaways

1. Use `method='regression'` for most optimization problems
2. Set `seed` for reproducibility
3. Use `var_type` for mixed variable types
4. Default parameters work well for most cases
5. Use `isotropic=True` for high-dimensional problems (>5D)

16.9.2. Quick Reference

```

from spotoptim import SpotOptim
from spotoptim.surrogate import Kriging

# Recommended configuration for general use
kriging = Kriging(
    method='regression',      # Smooths over noise
    seed=42                  # Reproducibility
)

optimizer = SpotOptim(
    fun=objective,
    bounds=bounds,
    surrogate=kriging,
    max_iter=50,
    n_initial=20,
    seed=42,
    verbose=True
)

result = optimizer.optimize()

```

16.9.3. Parameter Cheat Sheet

Parameter	Default	When to Change
method	'regression'	Use 'interpolation' for exact fit
noise	$\sqrt{\text{eps}}$	Rarely; use method='regression' instead
min_theta	-3.0	Adjust if default range doesn't work
max_theta	2.0	Adjust if default range doesn't work
isotropic	False	Set True for >5 dimensions
var_type	['num']	Set for mixed variable types
metric_factorial	'canberra'	Try 'hamming' for factors
model_fun_evals	100	Reduce for faster fitting
seed	124	Always set for reproducibility

16.10. Jupyter Notebook

Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

17. Kriging in SpotOptim: The Forrester Implementation

18. Introduction

The **Kriging** surrogate model in `spotoptim` is an implementation of the methods described in the classic text “**Engineering Design via Surrogate Modelling**” by Forrester, Sóbester, and Keane (2008). This tutorial explains the mathematical foundations of the implementation and demonstrates how to use the different fitting methods available in the library. The implementation has been validated against the original Matlab code provided with the book (`likelihood.m`, `pred.m`, etc.).

19. 1. The Core Kriging Model (Section 2.4)

Kriging, or Gaussian Process Regression, treats the response $y(\mathbf{x})$ as a realization of a stochastic process:

$$Y(\mathbf{x}) = \mu + Z(\mathbf{x})$$

where μ is the mean (often modeled as a constant in Ordinary Kriging) and $Z(\mathbf{x})$ is a Gaussian process with zero mean and covariance:

$$\text{Cov}(Z(\mathbf{x}^{(i)}), Z(\mathbf{x}^{(l)})) = \sigma^2 \Psi(\mathbf{x}^{(i)}, \mathbf{x}^{(l)})$$

19.1. Correlation Function

We use the Gaussian correlation function (Eq 2.4 in the book), which is infinitely differentiable and assumes a smooth underlying response:

$$\Psi(\mathbf{x}^{(i)}, \mathbf{x}^{(l)}) = \exp \left(- \sum_{j=1}^k \theta_j |x_j^{(i)} - x_j^{(l)}|^{p_j} \right)$$

In `spotoptim`, we typically set $p_j = 2$ (Gaussian) and optimize θ_j (length-scales).

19.2. Training: Concentrated Log-Likelihood

To fit the model, we maximize the **Concentrated Log-Likelihood** (Eq 2.30/2.33) with respect to the hyperparameters θ :

$$\ln(L) \approx -\frac{n}{2} \ln(\hat{\sigma}^2) - \frac{1}{2} \ln |\Psi|$$

where $\hat{\sigma}^2$ is the estimated variance (Eq 2.28). This matches the implementation in the book's `likelihood.m`.

19.3. Prediction

The prediction at a new point $\mathbf{x}$ is given by (Eq 2.15):

$$\hat{y}(\mathbf{x}) = \hat{\mu} + \psi^T \Psi^{-1}(\mathbf{y} - \mathbf{1}\hat{\mu})$$

where ψ is the correlation vector between the new point and the training points. This matches `pred.m`.

20. 2. Handling Noisy Data

Real-world data is often noisy. Forrester et al. (Section 6) describe three approaches to handle this, all available in `spotoptim` via the `method` argument.

20.1. A. Interpolation (`method="interpolation"`)

This is the standard Kriging formulation. It assumes **no noise** in the data.

- The model passes exactly through the training points.
- A very small “nugget” (noise) is added only for numerical stability of the matrix inversion.

20.2. B. Regression (`method="regression"`)

Described in Section 6.2 (“Regression Kriging”), this method assumes the observations contain noise ϵ :

$$y = \mu + Z(\mathbf{x}) + \epsilon$$

We introduce a regularization parameter λ (Lambda) to the diagonal of the correlation matrix:

$$\mathbf{C} = \Psi + \Psi^T + (1 + \lambda)\mathbf{I}$$

Both θ and λ are optimized simultaneously. The resulting surrogate:

- **Does not pass through the training points** (it smooths them).
- Is suitable for noisy objective functions.

20.3. C. Re-interpolation (method="reinterpolation")

Described in Section 6.3, this is a hybrid approach.

1. We fit the hyperparameters (θ and λ) using the **Regression** method to learn the noise level.
2. For prediction, we use the “noise-free” correlation matrix (removing λ) to predict the underlying smooth trend.

This “re-interpolates” the regression model, providing a predictor that typically has better error properties for design optimization than pure regression.

21. 3. Examples

We will verify these methods with a simple 1D example.

21.1. Setup

```
import numpy as np
import matplotlib.pyplot as plt
from spotoptim.surrogate import Kriging

# Define a function with noise
def true_function(x):
    return (x * 6 - 2)**2 * np.sin(x * 12 - 4)

np.random.seed(42)
X = np.linspace(0, 1, 11).reshape(-1, 1)
noise = np.random.normal(0, 1.0, X.shape)
y = true_function(X).flatten() + noise.flatten() * 3.0 # Add significant noise

X_plot = np.linspace(0, 1, 100).reshape(-1, 1)
y_true = true_function(X_plot)
```

21.2. Comparisons

We will fit three models and compare them.

```
# 1. Interpolation (Forces fit through noisy points)
model_interp = Kriging(method="interpolation", seed=42)
model_interp.fit(X, y)
y_interp, s_interp = model_interp.predict(X_plot, return_std=True)

# 2. Regression (Smooths the data by learning Lambda)
model_reg = Kriging(method="regression", seed=42, minLambda=-1, maxLambda=10)
model_reg.fit(X, y)
```

21. 3. Examples

```
y_reg, s_reg = model_reg.predict(X_plot, return_std=True)

# 3. Re-interpolation (Smooth trend based on regression parameters)
model_reinterp = Kriging(method="reinterpolation", seed=42)
model_reinterp.fit(X, y)
y_reinterp, s_reinterp = model_reinterp.predict(X_plot, return_std=True)
```

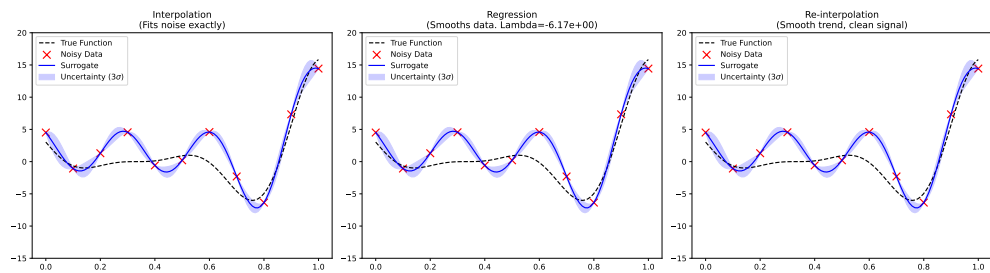
21.3. Visualization

```
fig, axes = plt.subplots(1, 3, figsize=(18, 5))

def plot_model(ax, title, y_pred, std):
    ax.plot(X_plot, y_true, 'k--', label="True Function")
    ax.scatter(X, y, c='r', marker='x', s=100, label="Noisy Data")
    ax.plot(X_plot, y_pred, 'b-', label="Surrogate")
    ax.fill_between(X_plot.flatten(),
                   y_pred - 3*std,
                   y_pred + 3*std,
                   color='b', alpha=0.2, label="Uncertainty (3 $\sigma$ )")
    ax.set_title(title)
    ax.legend()
    ax.set_ylim(-15, 20)

plot_model(axes[0], "Interpolation\n(Fits noise exactly)", y_interp, s_interp)
plot_model(axes[1], f"Regression\n(Smooths data. Lambda={model_reg.Lambda_:.2e})", y_reg, s_reg)
plot_model(axes[2], "Re-interpolation\n(Smooth trend, clean signal)", y_reinterp, s_reinterp)

plt.tight_layout()
plt.show()
```



21.3.1. Interpretation

1. **Interpolation:** Wiggles aggressively to hit every noisy point. This is “overfitting” the noise.
2. **Regression:** Produces a much smoother curve that does not hit the points. It has correctly identified that the data is noisy (optimized $\lambda > 0$).
3. **Re-interpolation:** Similar to regression in shape, but mathematically grounded in predicting the *process* rather than the *data*.

22. 4. Infill Criteria

The book also describes “Infill Criteria” (Section 3) for deciding where to sample next. This is implemented in `SpotOptim` itself (not the surrogate class). When you use `SpotOptim(..., acquisition="ei")`, you are using the **Expected Improvement** criterion described in Section 3.2.1 of the book.

```
from spotoptim import SpotOptim

# Use Kriging with EI
optimizer = SpotOptim(
    fun=lambda x: np.sum(x**2),
    bounds=[(-5, 5)],
    surrogate=model_reinterp, # Pass your customized Kriging model
    acquisition="ei",         # Expected Improvement
    max_iter=20
)
```


23. References

1. Forrester, A., Sobester, A., & Keane, A. (2008). *Engineering Design via Surrogate Modelling: A Practical Guide*. Wiley.
2. **SpotOptim Source Code:** `spotoptim/surrogate/kriging.py` (Validated against book's Matlab code).

23.1. Jupyter Notebook

Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

24. Nystroem Kernel Approximation

This chapter introduces the Nystroem method for approximating kernel maps, which enables scalable Gaussian Process regression for datasets with a large number of features (k) and a moderate number of samples (n).

24.1. Introduction

Gaussian Process (GP) models, such as Kriging, are powerful tools for regression and surrogate modeling. However, they suffer from computational scalability issues. The standard implementation scales as $\mathcal{O}(n^3)$, where n is the number of samples. Additionally, constructing the distance matrix scales as $\mathcal{O}(n^2k)$, where k is the number of features (dimensions).

When k is very large (high-dimensional data), the distance calculation itself can become a bottleneck. The Nystroem method offers a solution by approximating the feature map of the kernel using a subset of the training data.

24.2. The Nystroem Method

The Nystroem method approximates a kernel $K(x, y)$ by mapping inputs into a lower-dimensional feature space $\mathbb{R}^D$ (where D is the number of components).

$$x \mapsto \phi(x) \in \mathbb{R}^D$$

such that the inner product in this space approximates the kernel:

$$\langle \phi(x), \phi(y) \rangle \approx K(x, y)$$

24. Nystroem Kernel Approximation

24.2.1. Moderate n , Large k Use Case

While Nystroem is often cited for its ability to handle large n (by reducing the effective sample size for kernel construction), it is uniquely powerful for the “Moderate n , Large k ” scenario:

1. **Dimensionality Reduction:** It acts like a non-linear Kernel PCA. It projects high-dimensional data (k) into a feature space of dimension `n_components`.
2. **Efficiency:** If `n_components` $\ll k$, subsequent distance calculations in the GP model become significantly faster ($\mathcal{O}(n^2 \cdot \text{n\_components})$) instead of $\mathcal{O}(n^2 k)$.

24.3. Implementation in SpotOptim

`spotoptim` provides a modular implementation compatible with its `Kriging` surrogate. We use a `Pipeline` to chain the Nystroem approximation with the Kriging model.

24.3.1. Components

- `spotoptim.surrogate.nystroem.Nystroem`: The feature map approximator.
- `spotoptim.surrogate.pipeline.Pipeline`: Utility to chain transformers and estimators.
- `spotoptim.surrogate.kernels`: Modular kernels (`RBF`, `WhiteKernel`).

24.3.2. Example Usage

Here is how to set up a GP model with Nystroem approximation for a high-dimensional problem.

```
import numpy as np
from spotoptim.surrogate.kriging import Kriging
from spotoptim.surrogate.nystroem import Nystroem
from spotoptim.surrogate.pipeline import Pipeline
from spotoptim.surrogate.kernels import RBF, WhiteKernel

def define_nystroem_gp_model(n_components=50, noise_level=1e-5):
    """
    Defines a Pipeline with Nystroem approximation and Kriging.

    Args:
        n_components (int): Dimensionality of the target feature space.
        noise_level (float): Noise level for the WhiteKernel.
```



```

Returns:
    Pipeline: A configured pipeline ready to be fitted.
"""
# 1. Define the kernel for Nystroem
# We use an RBF kernel plus a small WhiteKernel for stability
kernel = 1.0 * RBF(length_scale=1.0) + WhiteKernel(noise_level=noise_level)

# 2. Define Nystroem approximation
# This will project data from k dimensions -> n_components dimensions
nystroem = Nystroem(kernel=kernel, n_components=n_components, random_state=42)

# 3. Define the Kriging model
# This model will receive inputs of dimension n_components
gp_model = Kriging(method='regression', seed=42)

# 4. Create the Pipeline
gp_model_pipeline = Pipeline([
    ('nystroem', nystroem),
    ('gp', gp_model)
])

return gp_model_pipeline

# --- Demonstration ---
# Generate high-dimensional synthetic data (Moderate n=50, Large k=1000)
rng = np.random.RandomState(42)
n_samples = 50
n_features = 1000

X_train = rng.randn(n_samples, n_features)
# Target depends only on first few features
y_train = np.sum(X_train[:, :5]**2, axis=1) + rng.normal(0, 0.1, n_samples)

# Create and fit model
pipeline = define_nystroem_gp_model(n_components=20) # Reduce 1000 dims -> 20 dims
pipeline.fit(X_train, y_train)

# Predict
X_test = rng.randn(10, n_features)
y_pred = pipeline.predict(X_test)

print(f"Predicted shape: {y_pred.shape}")
print(f"Sample predictions: {y_pred[:3]}")

```

24. Nystroem Kernel Approximation

```
Predicted shape: (10,)
Sample predictions: [4.6967762 4.6967762 4.6967762]
```

In this example, the Kriging model operates on 20 features instead of 1000, drastically reducing the cost of distance matrix computations during hyperparameter optimization.

24.4. Handling Mixed Variables

Standard kernel functions (like RBF) usually assume continuous input data. `spotoptim` provides a specialized `SpotOptimKernel` that can handle mixed variable types (continuous, integer, factor) correctly. This is essential if your high-dimensional dataset contains categorical variables.

24.4.1. Using `SpotOptimKernel`

To use mixed variables with Nystroem, you must initialize `Nystroem` with a `SpotOptimKernel` configured with your variable types.

```
from spotoptim.surrogate.kernels import SpotOptimKernel

# Define variable types for your data features
# Example: 2 floats, 1 integer, 1 factor
var_types = ['float', 'float', 'int', 'factor']

# Initial weights/theta for the kernel (usually ones, or optimized externally)
theta = np.ones(len(var_types))

# 1. Create the Mixed Kernel
mixed_kernel = SpotOptimKernel(theta=theta, var_type=var_types)

# 2. Use it in Nystroem
nystroem = Nystroem(kernel=mixed_kernel, n_components=50)

# The pipeline setup remains the same
pipeline = Pipeline([
    ('nystroem', nystroem),
    ('gp', Kriging())
])
```

When `fit` or `transform` is called, `SpotOptimKernel` will compute distances using the appropriate metric for each variable type (e.g., Hamming/Canberra distance for factors) before the Nystroem projection occurs.

24.5. Jupyter Notebook

i Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

Part VII.

Multiobjective Optimization

25. Multi-Objective Optimization Support in SpotOptim

```
import copy
import numpy as np
import pandas as pd
import numpy as np
import warnings

from sklearn.ensemble import RandomForestRegressor
from sklearn.preprocessing import MinMaxScaler
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.kernel_approximation import Nystroem
from sklearn.gaussian_process.kernels import RBF, WhiteKernel
from sklearn.pipeline import Pipeline
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score

import matplotlib.pyplot as plt
import seaborn as sns

from scipy.stats.qmc import LatinHypercube
from scipy.optimize import dual_annealing

from spotdesirability.utils.desirability import DMin, DMax, DOverall

from spotoptim.sampling.mm import mmphi_intensive, mmphi_intensive_update, mm_improvement_contour
from spotoptim.inspection.predictions import plot_actual_vs_predicted
from spotoptim.inspection import (generate_mdi, generate_imp, plot_importances, plot_feature_importances)

from spotoptim.utils.boundaries import get_boundaries, map_to_original_scale
from spotoptim.mo.pareto import is_pareto_efficient, plot_mo
from spotoptim.eda import plot_ip_boxplots, plot_ip_histograms
from spotoptim.utils import get_combinations
from spotoptim.sampling.lhs import rlh
```

25. Multi-Objective Optimization Support in SpotOptim

```
from spotoptim.sampling.mm import (bestlh, plot_mmphi_vs_n_lhs, mmphi, mmphi_intensive)
from spotoptim.mo import mo_mm_desirability_function, mo_mm_desirability_optimizer
from spotoptim.mo.mo_mm import mo_xy_desirability_plot
from spotoptim.function.mo import mo_conv2_max
from spotoptim.mo.pareto import (mo_xy_surface, mo_xy_contour, mo_pareto_optx_plot)
from spotoptim.utils.eval import mo_cv_models, mo_eval_models
warnings.filterwarnings("ignore")
```

```
n_estimators=100
max_depth = None
random_state=42
```

25.1. Overview

SpotOptim supports multi-objective optimization functions with automatic detection and flexible scalarization strategies. This implementation follows the same approach as the Spot class from spotPython.

25.2. What Was Implemented

25.2.1. 1. Core Functionality

Parameter:

- `fun_mo2so` (callable, optional): Function to convert multi-objective values to single-objective
 - Takes array of shape `(n_samples, n_objectives)`
 - Returns array of shape `(n_samples,)`
 - If `None`, uses first objective (default behavior)

Attribute:

- `y_mo` (ndarray or `None`): Stores all multi-objective function values
 - Shape: `(n_samples, n_objectives)` for multi-objective problems
 - `None` for single-objective problems

Methods:

- `_get_shape(y)`: Get shape of objective function output
- `_store_mo(y_mo)`: Store multi-objective values with automatic appending

- `_mo2so(y_mo)`: Convert multi-objective to single-objective values

The method `_evaluate_function(X)` automatically detects multi-objective functions. It calls `_mo2so()` to convert multi-objective to single-objective. It also stores the original multi-objective values in `y_mo`. And it returns single-objective values for optimization.

25.3. Usage Examples

25.3.1. Example 1: Default Behavior (Use First Objective)

```
import numpy as np
from spotoptim import SpotOptim

def bi_objective(X):
    """Two conflicting objectives."""
    obj1 = np.sum(X**2, axis=1)      # Minimize at origin
    obj2 = np.sum((X - 2)**2, axis=1) # Minimize at (2, 2)
    return np.column_stack([obj1, obj2])

optimizer = SpotOptim(
    fun=bi_objective,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=30,
    n_initial=15,
    seed=42
)

result = optimizer.optimize()

print(f"Best x: {result.x}")          # Near [0, 0]
print(f"Best f(x): {result.fun}")    # Minimizes obj1
print(f"M0 values stored: {optimizer.y_mo.shape}") # (30, 2)
```

```
Best x: [3.31436760e-04 4.18312302e-05]
Best f(x): 1.1160017787260647e-07
M0 values stored: (30, 2)
```

25.3.2. Example 2: Weighted Sum Scalarization

```
def weighted_sum(y_mo):
    """Equal weighting of objectives."""
    return 0.5 * y_mo[:, 0] + 0.5 * y_mo[:, 1]

optimizer = SpotOptim(
    fun=bi_objective,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=30,
    n_initial=15,
    fun_mo2so=weighted_sum, # Custom conversion
    seed=42
)

result = optimizer.optimize()
print(f"Compromise solution: {result.x}") # Near [1, 1]
```

Compromise solution: [1.00110134 1.0000297]

25.3.3. Example 3: Min-Max Scalarization

```
def min_max(y_mo):
    """Minimize the maximum objective."""
    return np.max(y_mo, axis=1)

optimizer = SpotOptim(
    fun=bi_objective,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=30,
    n_initial=15,
    fun_mo2so=min_max,
    seed=42
)

result = optimizer.optimize()
# Finds solution with balanced objective values
```

25.3.4. Example 4: Three or More Objectives

```
def three_objectives(X):
    """Three different norms."""
    obj1 = np.sum(X**2, axis=1)          # L2 norm
    obj2 = np.sum(np.abs(X), axis=1)     # L1 norm
    obj3 = np.max(np.abs(X), axis=1)     # L-infinity norm
    return np.column_stack([obj1, obj2, obj3])

def custom_scalarization(y_mo):
    """Weighted combination."""
    return 0.4 * y_mo[:, 0] + 0.3 * y_mo[:, 1] + 0.3 * y_mo[:, 2]

optimizer = SpotOptim(
    fun=three_objectives,
    bounds=[(-5, 5), (-5, 5), (-5, 5)],
    max_iter=35,
    n_initial=20,
    fun_mo2so=custom_scalarization,
    seed=42
)

result = optimizer.optimize()
```

25.3.5. Example 5: With Noise Handling

```
def noisy_bi_objective(X):
    """Noisy multi-objective function."""
    noise1 = np.random.normal(0, 0.05, X.shape[0])
    noise2 = np.random.normal(0, 0.05, X.shape[0])

    obj1 = np.sum(X**2, axis=1) + noise1
    obj2 = np.sum((X - 1)**2, axis=1) + noise2
    return np.column_stack([obj1, obj2])

optimizer = SpotOptim(
    fun=noisy_bi_objective,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=40,
    n_initial=20,
    repeats_initial=3,          # Handle noise
)
```

25. Multi-Objective Optimization Support in SpotOptim

```
    repeats_surrogate=2,  
    seed=42  
)  
  
result = optimizer.optimize()  
# Works seamlessly with noise handling
```

25.4. Common Scalarization Strategies

25.4.1. 1. Weighted Sum

```
def weighted_sum(y_mo, weights=[0.5, 0.5]):  
    return sum(w * y_mo[:, i] for i, w in enumerate(weights))
```

Use **when**: Objectives have similar scales and you want linear trade-offs

25.4.2. 2. Weighted Sum with Normalization

```
def normalized_weighted_sum(y_mo, weights=[0.5, 0.5]):  
    # Normalize each objective to [0, 1]  
    y_norm = (y_mo - y_mo.min(axis=0)) / (y_mo.max(axis=0) - y_mo.min(axis=0) + 1e-10)  
    return sum(w * y_norm[:, i] for i, w in enumerate(weights))
```

Use **when**: Objectives have very different scales

25.4.3. 3. Min-Max (Chebyshev)

```
def min_max(y_mo):  
    return np.max(y_mo, axis=1)
```

Use **when**: You want balanced performance across all objectives

25.4.4. 4. Target Achievement

```
def target_achievement(y_mo, targets=[0.0, 0.0]):
    # Minimize deviation from targets
    return np.sum((y_mo - targets)**2, axis=1)
```

Use when: You have specific target values for each objective

25.4.5. 5. Product

```
def product(y_mo):
    return np.prod(y_mo + 1e-10, axis=1) # Add small value to avoid zero
```

Use when: All objectives should be minimized together

25.5. Integration with Other Features

Multi-objective support works seamlessly with:

Noise Handling - Use `repeats_initial` and `repeats_surrogate`
OCBA - Use `ocba_delta` for intelligent re-evaluation
TensorBoard Logging - Logs converted single-objective values
Dimension Reduction - Fixed dimensions work normally
Custom Variable Names - `var_name` parameter supported

25.6. Implementation Details

25.6.1. Automatic Detection

SpotOptim automatically detects multi-objective functions:

- If function returns 2D array (`n_samples`, `n_objectives`), it's multi-objective
- If function returns 1D array (`n_samples`), it's single-objective

1. Function returns multi-objective values
2. `_store_mo()` saves them in `y_mo` attribute
3. `_mo2so()` converts to single-objective using `fun_mo2so` or default
4. Surrogate model optimizes the single-objective values
5. All original multi-objective values remain accessible in `y_mo`

- Single-objective functions work unchanged
- `fun_mo2so` defaults to `None`
- `y_mo` is `None` for single-objective problems
- No breaking changes to existing code

- Scalarization approach to multi-objective optimization
- Single solution found per optimization run
- Different scalarizations $\rightarrow$ different Pareto solutions
- Suitable for preference-based multi-objective optimization

- Not a true multi-objective optimizer (doesn't find Pareto front)
- Doesn't generate multiple solutions in one run
- Not suitable for discovering entire Pareto front

25.7.3. For True Multi-Objective Optimization

For finding the complete Pareto front, consider specialized tools:

- **pymoo**: Comprehensive multi-objective optimization framework
- **platypus**: Multi-objective optimization library
- **NSGA-II**, **MOEA/D**: Dedicated multi-objective algorithms

25.8. Demo Script

Run the comprehensive demo (the demos files are located in the **examples** folder):

```
python demo_multiobjective.py
```

This demonstrates:

- Default behavior (first objective)
- Weighted sum scalarization
- Min-max scalarization
- Noisy multi-objective optimization
- Three-objective optimization

25.9. Benchmark Multi-Objective Test Functions

SpotOptim includes 12 multi-objective benchmark functions in `spotoptim.function.mo`. These functions are widely used to evaluate and compare multi-objective optimization algorithms.

25.9.1. Function Overview

Function	Objectives	Variables	Pareto Front	Characteristics
ZDT1	2	30	Convex	Unimodal, minimization
ZDT2	2	30	Non-convex	Unimodal, minimization
ZDT3	2	30	Disconnected	Multimodal, minimization
ZDT4	2	10	Convex	Many local fronts
ZDT6	2	10	Non-convex	Non-uniform density

25. Multi-Objective Optimization Support in SpotOptim

Function	Objectives	Variables	Pareto Front	Characteristics
DTLZ1	Scalable	n_obj+4	Linear	Multimodal
DTLZ2	Scalable	n_obj+9	Spherical	Unimodal
Schaffer N1	2	1	Convex	Simple, minimization
Fonseca-Fleming	2	2-10	Concave	Symmetric, minimization
mo_conv2_min		2	Convex	Convex, minimization
mo_conv2_max		2	Convex	Convex, maximization
Kursawe	2	3	Disconnected	Non-convex, disconnected minimization

mo_conv2_min is a smooth convex minimization pair with objectives $f_1(x, y) = x^2 + y^2$ and $f_2(x, y) = (x - 1)^2 + (y - 1)^2$ on $[0, 1]^2$, producing a convex quadratic Pareto front along $x = y$. mo_conv2_max flips both objectives for maximization via $f_1(x, y) = 2 - (x^2 + y^2)$ and $f_2(x, y) = 2 - ((1 - x)^2 + (1 - y)^2)$, keeping objective values bounded in $[0, 2]$.

25.9.2. Example 6: ZDT1 - Convex Pareto Front

ZDT1 is a classical bi-objective test problem with a convex Pareto front, widely used for benchmarking.

```
import numpy as np
import matplotlib.pyplot as plt
from spotoptim.function.mo import zdt1
from spotoptim import SpotOptim

# Generate data for visualization
n_points = 100
x1_range = np.linspace(0, 1, n_points)

# Pareto front: x1 in [0,1], rest = 0
X_pareto = np.column_stack([x1_range, np.zeros((n_points, 2))])
y_pareto = zdt1(X_pareto)

# Non-optimal points for comparison
X_non_optimal = np.random.rand(200, 3)
y_non_optimal = zdt1(X_non_optimal)
```



```

# Visualize Pareto front
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Plot objective space
ax1.scatter(y_non_optimal[:, 0], y_non_optimal[:, 1],
            alpha=0.3, label='Non-optimal', s=20)
ax1.plot(y_pareto[:, 0], y_pareto[:, 1],
         'r-', linewidth=2, label='Pareto Front')
ax1.set_xlabel('f1(x) = x1')
ax1.set_ylabel('f2(x) = g(x) * [1 - √(x1/g(x))]\')
ax1.set_title('ZDT1: Convex Pareto Front')
ax1.legend()
ax1.grid(True, alpha=0.3)

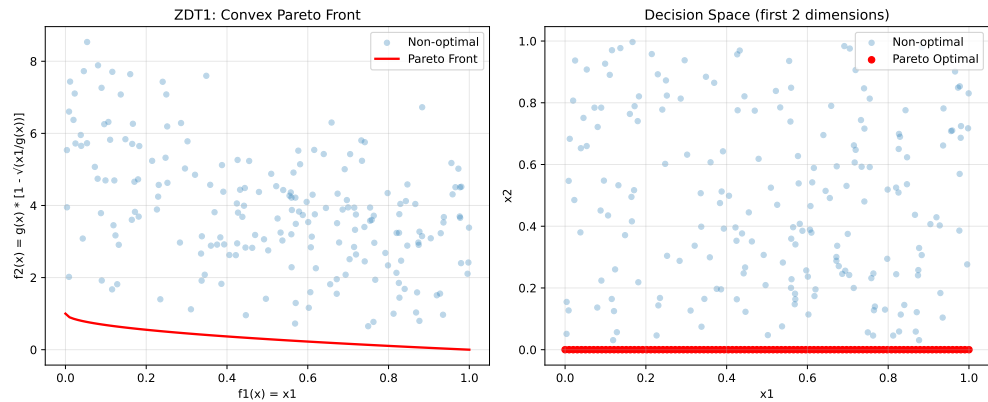
# Plot decision space (first two variables)
ax2.scatter(X_non_optimal[:, 0], X_non_optimal[:, 1],
            alpha=0.3, label='Non-optimal', s=20)
ax2.scatter(X_pareto[:, 0], X_pareto[:, 1],
            c='red', label='Pareto Optimal', s=30)
ax2.set_xlabel('x1')
ax2.set_ylabel('x2')
ax2.set_title('Decision Space (first 2 dimensions)\')
ax2.legend()
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print("ZDT1 Characteristics:")
print("- Convex Pareto front: f2 = 1 - √f1")
print("- Optimal when x2, x3, ..., xn = 0")
print("- Easy to solve, good for testing basic functionality")

```

25. Multi-Objective Optimization Support in SpotOptim



ZDT1 Characteristics:

- Convex Pareto front: $f2 = 1 - \sqrt{f1}$
- Optimal when $x2, x3, \dots, xn = 0$
- Easy to solve, good for testing basic functionality

Now optimize using SpotOptim with weighted sum:

```
def weighted_sum_zdt1(y_mo):
    """Equal weighting for ZDT1."""
    return 0.5 * y_mo[:, 0] + 0.5 * y_mo[:, 1]

optimizer = SpotOptim(
    fun=zdt1,
    bounds=[(0, 1)] * 30, # 30 variables in [0, 1]
    max_iter=50,
    n_initial=20,
    fun_mo2so=weighted_sum_zdt1,
    seed=42
)

result = optimizer.optimize()

print(f"\nOptimization Result:")
print(f"Best x (first 5 dims): {result.x[:5]}")
print(f"Best scalarized value: {result.fun:.6f}")

# Evaluate multi-objective values
y_best = zdt1(result.x.reshape(1, -1))
print(f"f1 = {y_best[0, 0]:.6f}, f2 = {y_best[0, 1]:.6f}")
```

```

Optimization Result:
Best x (first 5 dims): [1. 0. 0. 0. 0.]
Best scalarized value: 0.500000
f1 = 1.000000, f2 = 0.000000

```

25.9.3. Example 7: ZDT2 - Non-Convex Pareto Front

ZDT2 has a non-convex Pareto front, making it more challenging than ZDT1.

```

from spotoptim.function.mo import zdt2

# Generate Pareto front
X_pareto = np.column_stack([x1_range, np.zeros((n_points, 2))])
y_pareto = zdt2(X_pareto)

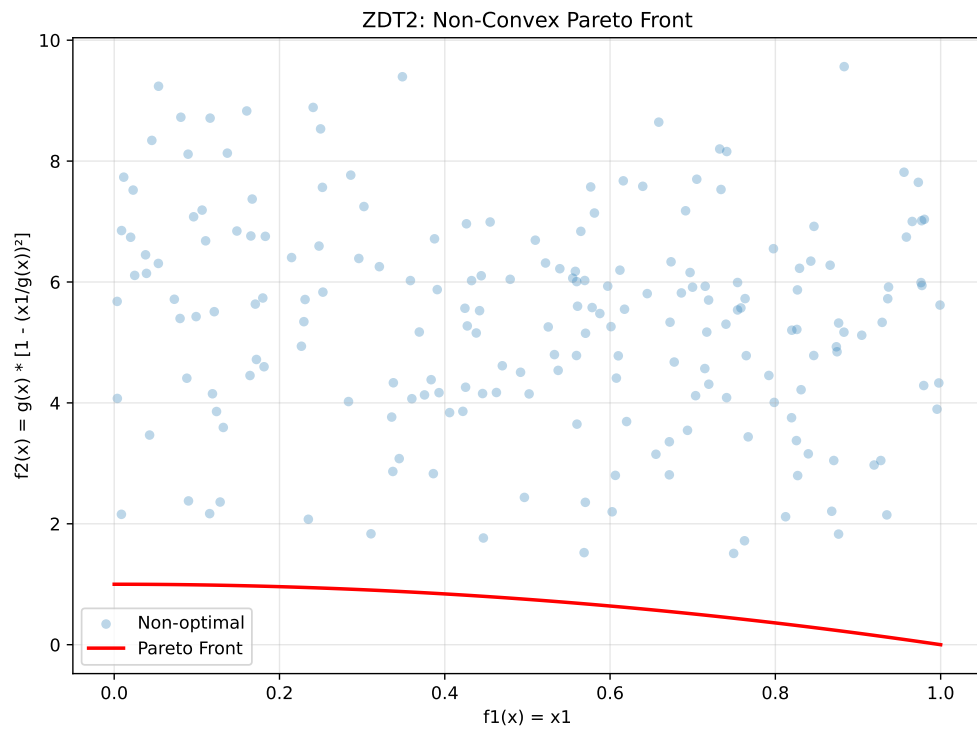
# Non-optimal points
y_non_optimal = zdt2(X_non_optimal)

# Visualize
fig, ax = plt.subplots(figsize=(8, 6))
ax.scatter(y_non_optimal[:, 0], y_non_optimal[:, 1],
           alpha=0.3, label='Non-optimal', s=20)
ax.plot(y_pareto[:, 0], y_pareto[:, 1],
        'r-', linewidth=2, label='Pareto Front')
ax.set_xlabel('f1(x) = x1')
ax.set_ylabel('f2(x) = g(x) * [1 - (x1/g(x))2]')
ax.set_title('ZDT2: Non-Convex Pareto Front')
ax.legend()
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print("ZDT2 Characteristics:")
print("- Non-convex Pareto front:  $f_2 = 1 - f_1^2$ ")
print("- Tests algorithm's ability to handle non-convexity")
print("- Optimal when  $x_2, x_3, \dots, x_n = 0$ ")

```

25. Multi-Objective Optimization Support in SpotOptim



ZDT2 Characteristics:

- Non-convex Pareto front: $f_2 = 1 - f_1^2$
- Tests algorithm's ability to handle non-convexity
- Optimal when $x_2, x_3, \dots, x_n = 0$

25.9.4. Example 8: ZDT3 - Disconnected Pareto Front

ZDT3 has a discontinuous Pareto front consisting of 5 separate regions.

```
from spotoptim.function.mo import zdt3

# Generate Pareto front
X_pareto = np.column_stack([x1_range, np.zeros((n_points, 2))])
y_pareto = zdt3(X_pareto)

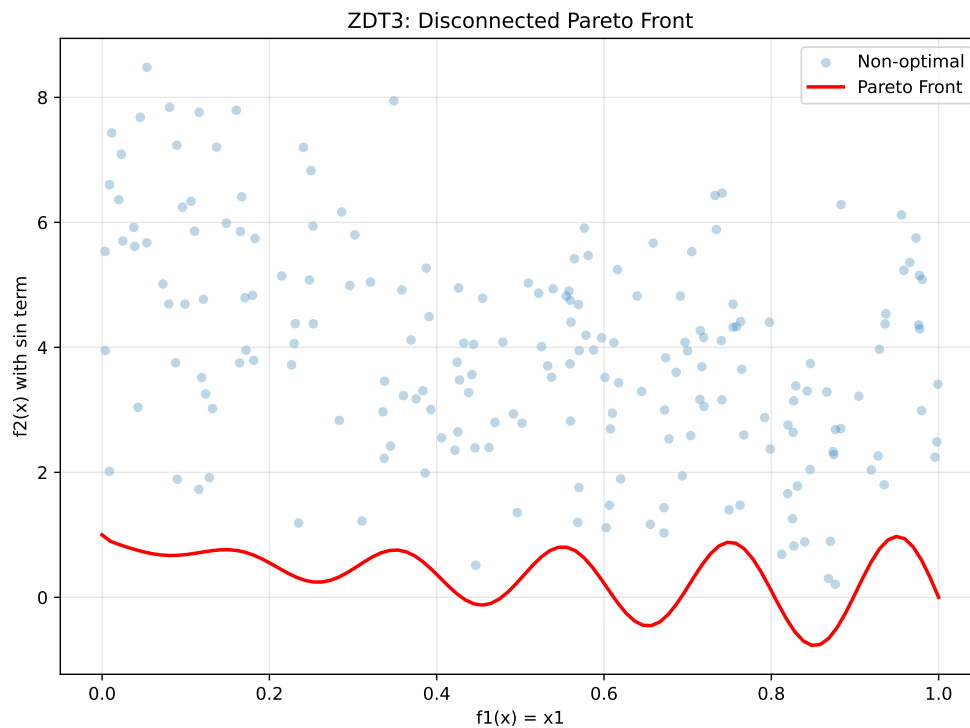
# Non-optimal points
y_non_optimal = zdt3(X_non_optimal)

# Visualize
```

25.9. Benchmark Multi-Objective Test Functions

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.scatter(y_non_optimal[:, 0], y_non_optimal[:, 1],
          alpha=0.3, label='Non-optimal', s=20)
ax.plot(y_pareto[:, 0], y_pareto[:, 1],
       'r-', linewidth=2, label='Pareto Front')
ax.set_xlabel('f1(x) = x1')
ax.set_ylabel('f2(x) with sin term')
ax.set_title('ZDT3: Disconnected Pareto Front')
ax.legend()
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print("ZDT3 Characteristics:")
print("- Disconnected Pareto front with 5 separate regions")
print("- Tests diversity maintenance in algorithms")
print("- sin(10 x1) term creates discontinuities")
```



ZDT3 Characteristics:

25. Multi-Objective Optimization Support in SpotOptim

- Disconnected Pareto front with 5 separate regions
- Tests diversity maintenance in algorithms
- $\sin(10 x_1)$ term creates discontinuities

25.9.5. Example 9: ZDT4 - Multimodal with Many Local Fronts

ZDT4 has 21^9 local Pareto fronts, testing the algorithm's ability to escape local optima.

```
from spotoptim.function.mo import zdt4

# Generate Pareto front (same as ZDT1 when optimal)
X_pareto = np.column_stack([x1_range, np.zeros((n_points, 9))])
y_pareto = zdt4(X_pareto)

# Non-optimal points with multimodal g-function
X_non_optimal_zdt4 = np.random.rand(200, 10)
X_non_optimal_zdt4[:, 0] = np.clip(X_non_optimal_zdt4[:, 0], 0, 1) # x1 in [0,1]
X_non_optimal_zdt4[:, 1:] = X_non_optimal_zdt4[:, 1:] * 10 - 5 # others in [-5,5]
y_non_optimal = zdt4(X_non_optimal_zdt4)

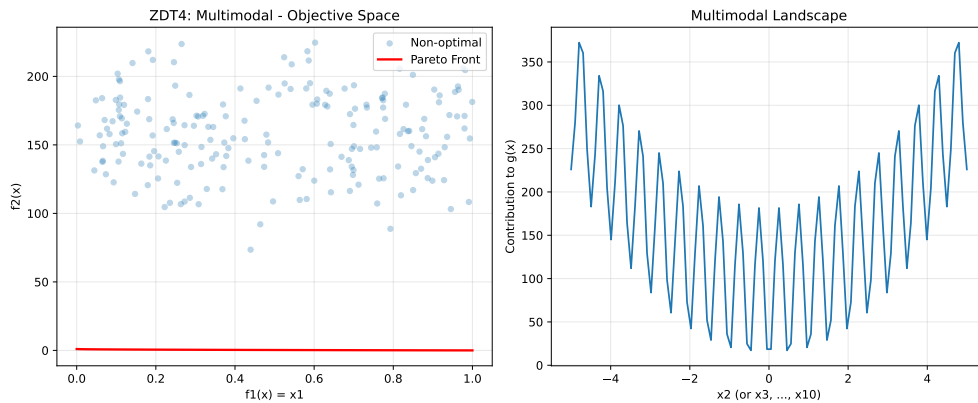
# Visualize
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Objective space
ax1.scatter(y_non_optimal[:, 0], y_non_optimal[:, 1],
            alpha=0.3, label='Non-optimal', s=20)
ax1.plot(y_pareto[:, 0], y_pareto[:, 1],
        'r-', linewidth=2, label='Pareto Front')
ax1.set_xlabel('f1(x) = x1')
ax1.set_ylabel('f2(x)')
ax1.set_title('ZDT4: Multimodal - Objective Space')
ax1.legend()
ax1.grid(True, alpha=0.3)

# Show multimodal landscape for x2
x2_range = np.linspace(-5, 5, 100)
g_vals = 1 + 10 * 9 + (x2_range**2 - 10 * np.cos(4 * np.pi * x2_range)) * 9
ax2.plot(x2_range, g_vals)
ax2.set_xlabel('x2 (or x3, ..., x10)')
ax2.set_ylabel('Contribution to g(x)')
ax2.set_title('Multimodal Landscape')
ax2.grid(True, alpha=0.3)
```

```
plt.tight_layout()
plt.show()

print("ZDT4 Characteristics:")
print("- 21^9 local Pareto fronts")
print("- Tests global search capability")
print("- x1 in [0,1], x_i in [-5,5] for i=2,...,10")
```



ZDT4 Characteristics:

- 21^9 local Pareto fronts
- Tests global search capability
- x_1 in $[0,1]$, x_i in $[-5,5]$ for $i=2,\dots,10$

25.9.6. Example 10: ZDT6 - Non-Uniform Density

ZDT6 has a non-uniform search space with low density of solutions near the Pareto front.

```
from spotoptim.function.mo import zdt6

# Generate Pareto front
X_pareto = np.column_stack([x1_range, np.zeros((n_points, 2))])
y_pareto = zdt6(X_pareto)

# Non-optimal points
y_non_optimal = zdt6(X_non_optimal)

# Visualize
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))
```

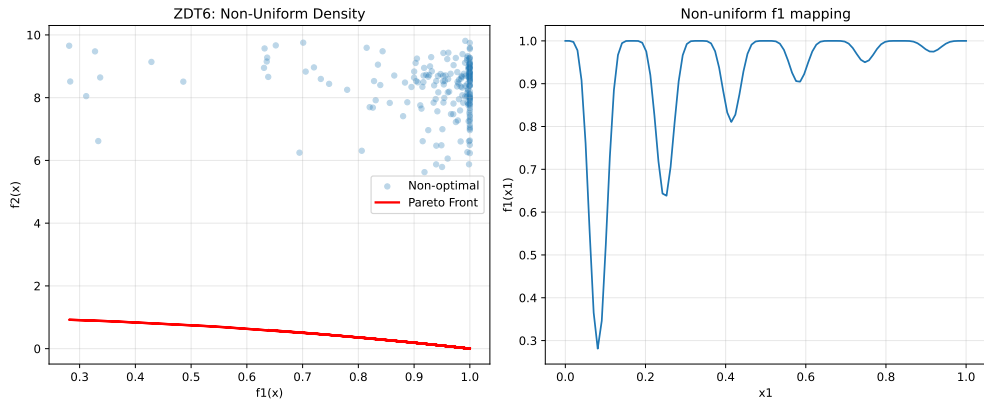
25. Multi-Objective Optimization Support in SpotOptim

```
# Objective space
ax1.scatter(y_non_optimal[:, 0], y_non_optimal[:, 1],
            alpha=0.3, label='Non-optimal', s=20)
ax1.plot(y_pareto[:, 0], y_pareto[:, 1],
        'r-', linewidth=2, label='Pareto Front')
ax1.set_xlabel('f1(x)')
ax1.set_ylabel('f2(x)')
ax1.set_title('ZDT6: Non-Uniform Density')
ax1.legend()
ax1.grid(True, alpha=0.3)

# Show f1 non-uniformity
ax2.plot(x1_range, 1 - np.exp(-4 * x1_range) * (np.sin(6 * np.pi * x1_range)**6))
ax2.set_xlabel('x1')
ax2.set_ylabel('f1(x1)')
ax2.set_title('Non-uniform f1 mapping')
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print("ZDT6 Characteristics:")
print("- Non-uniform density of Pareto optimal solutions")
print("- Low density near the Pareto front")
print("- Tests diversity maintenance")
```



ZDT6 Characteristics:

- Non-uniform density of Pareto optimal solutions
- Low density near the Pareto front
- Tests diversity maintenance

25.9.7. Example 11: DTLZ2 - Scalable Spherical Pareto Front

DTLZ2 is a scalable test problem with a concave spherical Pareto front.

```
from spotoptim.function.mo import dtlz2

# 3D visualization for 3 objectives
n_samples = 500
X_samples = np.random.rand(n_samples, 12)
y_samples = dtlz2(X_samples, n_obj=3)

# Generate Pareto optimal points (on unit sphere)
theta = np.linspace(0, np.pi/2, 30)
phi = np.linspace(0, np.pi/2, 30)
theta_grid, phi_grid = np.meshgrid(theta, phi)

f1 = np.cos(theta_grid) * np.cos(phi_grid)
f2 = np.cos(theta_grid) * np.sin(phi_grid)
f3 = np.sin(theta_grid)

# 3D plot
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

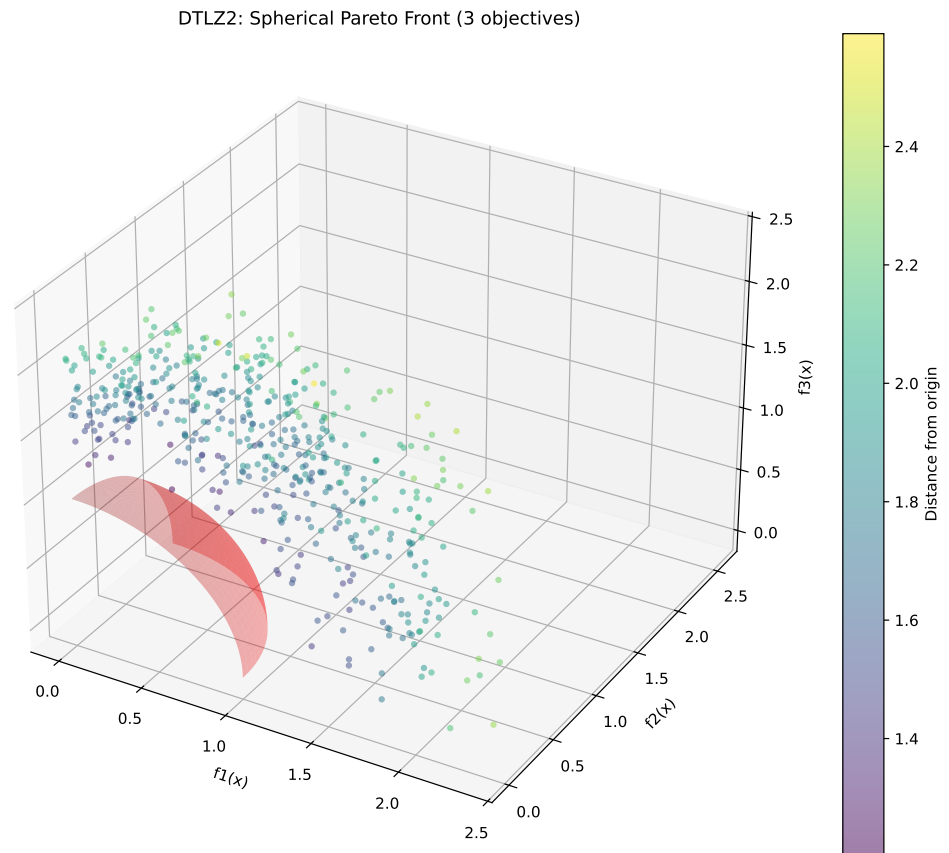
# Pareto front surface
ax.plot_surface(f1, f2, f3, alpha=0.3, color='red', label='Pareto Front')

# Sample points
colors = np.linalg.norm(y_samples, axis=1)
scatter = ax.scatter(y_samples[:, 0], y_samples[:, 1], y_samples[:, 2],
                    c=colors, cmap='viridis', alpha=0.5, s=10)

ax.set_xlabel('f1(x)')
ax.set_ylabel('f2(x)')
ax.set_zlabel('f3(x)')
ax.set_title('DTLZ2: Spherical Pareto Front (3 objectives)')
plt.colorbar(scatter, label='Distance from origin')
plt.tight_layout()
plt.show()

print("DTLZ2 Characteristics:")
print("- Scalable number of objectives")
print("- Pareto front is unit sphere:  $\sum f_i^2 = 1$ ")
print("- Concave, unimodal")
print(f"- Sample: sum of squares = {np.sum(y_samples[0]**2):.4f}")
```

25. Multi-Objective Optimization Support in SpotOptim



DTLZ2 Characteristics:

- Scalable number of objectives
- Pareto front is unit sphere: $\sum f_i^2 = 1$
- Concave, unimodal
- Sample: sum of squares = 3.3716

Optimize DTLZ2 with 3 objectives:

```
def weighted_sum_3obj(y_mo):  
    """Equal weighting for 3 objectives."""  
    return np.mean(y_mo, axis=1)  
  
optimizer = SpotOptim(  
    fun=lambda X: dtlz2(X, n_obj=3),
```

```

    bounds=[(0, 1)] * 12, # 12 variables for 3 objectives
    max_iter=50,
    n_initial=25,
    fun_mo2so=weighted_sum_3obj,
    seed=42
)

result = optimizer.optimize()

# Evaluate
y_best = dtlz2(result.x.reshape(1, -1), n_obj=3)
print(f"\nOptimization Result:")
print(f"f1 = {y_best[0, 0]:.4f}, f2 = {y_best[0, 1]:.4f}, f3 = {y_best[0, 2]:.4f}")
print(f"Sum of squares: {np.sum(y_best[0]**2):.4f} (should be ~1.0)")

```

```

Optimization Result:
f1 = 0.0000, f2 = 0.0000, f3 = 1.2524
Sum of squares: 1.5684 (should be ~1.0)

```

25.9.8. Example 12: Schaffer N1 - Simple Bi-Objective

Schaffer N1 is one of the earliest and simplest multi-objective test functions.

```

from spotoptim.function.mo import schaffer_n1

# Generate data
x_range = np.linspace(-10, 10, 200).reshape(-1, 1)
y_schaffer = schaffer_n1(x_range)

# Pareto optimal range [0, 2]
x_pareto = np.linspace(0, 2, 100).reshape(-1, 1)
y_pareto = schaffer_n1(x_pareto)

# Visualize
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Objective space
ax1.plot(y_schaffer[:, 0], y_schaffer[:, 1], 'b-', alpha=0.3, label='All points')
ax1.plot(y_pareto[:, 0], y_pareto[:, 1], 'r-', linewidth=2, label='Pareto Front')
ax1.set_xlabel('f1(x) = x2')
ax1.set_ylabel('f2(x) = (x-2)2')
ax1.set_title('Schaffer N1: Objective Space')

```

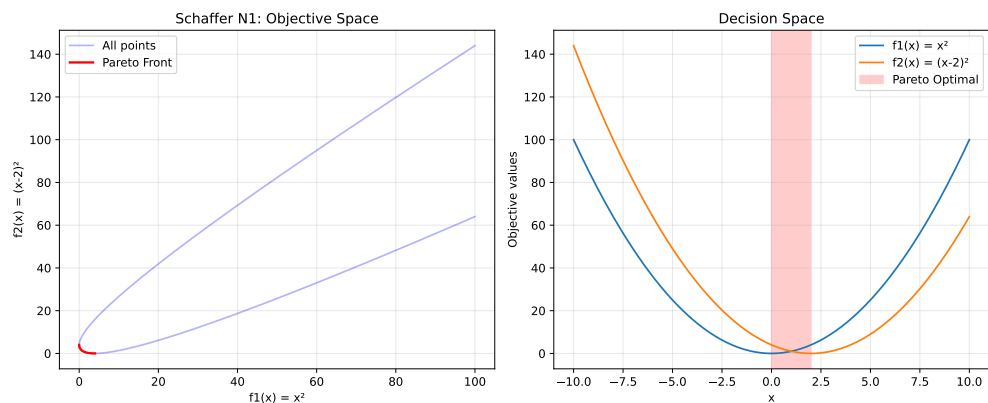
25. Multi-Objective Optimization Support in SpotOptim

```
ax1.legend()
ax1.grid(True, alpha=0.3)

# Decision space
ax2.plot(x_range, y_schaffer[:, 0], label='f1(x) = x2')
ax2.plot(x_range, y_schaffer[:, 1], label='f2(x) = (x-2)2')
ax2.axvspan(0, 2, alpha=0.2, color='red', label='Pareto Optimal')
ax2.set_xlabel('x')
ax2.set_ylabel('Objective values')
ax2.set_title('Decision Space')
ax2.legend()
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print("Schaffer N1 Characteristics:")
print("- Simplest multi-objective function")
print("- 1D decision variable")
print("- Pareto optimal: x [0, 2]")
print("- Convex Pareto front")
```



Schaffer N1 Characteristics:

- Simplest multi-objective function
- 1D decision variable
- Pareto optimal: x [0, 2]
- Convex Pareto front

25.9.9. Example 13: Fonseca-Fleming - Symmetric Concave Front

Fonseca-Fleming is a classical bi-objective problem with a concave Pareto front. It is defined for 2 to 10 decision variables as follows:

$$f_1(x) = 1 - \exp\left(-\sum_{i=1}^n \left(x_i - \frac{1}{\sqrt{n}}\right)^2\right) \quad f_2(x) = 1 - \exp\left(-\sum_{i=1}^n \left(x_i + \frac{1}{\sqrt{n}}\right)^2\right),$$

where $x_i \in [-2, 2]$. The Pareto optimal solutions satisfy:

$$\sum_{i=1}^n x_i^2 = \frac{1}{n}.$$

```
from spotoptim.function.mo import fonseca_fleming

# Generate data for 2D
n_grid = 50
x1_ff = np.linspace(-2, 2, n_grid)
x2_ff = np.linspace(-2, 2, n_grid)
X1_grid, X2_grid = np.meshgrid(x1_ff, x2_ff)

# Evaluate on grid
X_grid = np.column_stack([X1_grid.ravel(), X2_grid.ravel()])
y_grid = fonseca_fleming(X_grid)

# Pareto optimal points (approximately)
X_pareto_ff = np.linspace(-1/np.sqrt(2), 1/np.sqrt(2), 100)
X_pareto_2d = np.column_stack([X_pareto_ff, X_pareto_ff])
y_pareto_ff = fonseca_fleming(X_pareto_2d)

# Visualize
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

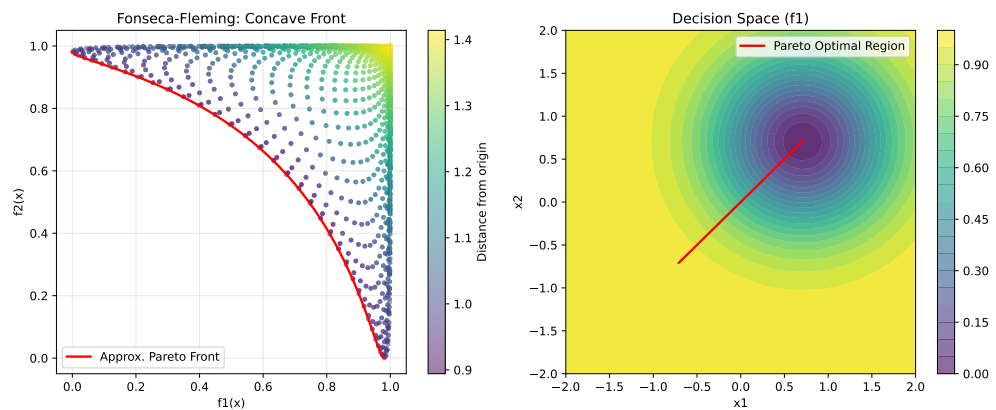
# Objective space
scatter = ax1.scatter(y_grid[:, 0], y_grid[:, 1],
                     c=np.linalg.norm(y_grid, axis=1),
                     cmap='viridis', alpha=0.5, s=10)
ax1.plot(y_pareto_ff[:, 0], y_pareto_ff[:, 1],
        'r-', linewidth=2, label='Approx. Pareto Front')
ax1.set_xlabel('f1(x)')
ax1.set_ylabel('f2(x)')
ax1.set_title('Fonseca-Fleming: Concave Front')
ax1.legend()
ax1.grid(True, alpha=0.3)
plt.colorbar(scatter, ax=ax1, label='Distance from origin')
```

25. Multi-Objective Optimization Support in SpotOptim

```
# Decision space
ax2.contourf(X1_grid, X2_grid, y_grid[:, 0].reshape(n_grid, n_grid),
             levels=20, cmap='viridis', alpha=0.6)
ax2.plot(X_pareto_2d[:, 0], X_pareto_2d[:, 1],
        'r-', linewidth=2, label='Pareto Optimal Region')
ax2.set_xlabel('x1')
ax2.set_ylabel('x2')
ax2.set_title('Decision Space (f1)')
ax2.legend()
plt.colorbar(ax2.contourf(X1_grid, X2_grid, y_grid[:, 0].reshape(n_grid, n_grid),
                        levels=20, cmap='viridis', alpha=0.6), ax=ax2)

plt.tight_layout()
plt.show()

print("Fonseca-Fleming Characteristics:")
print("- Concave Pareto front (minimization)")
print("- Symmetric problem")
print("- Scalable to higher dimensions")
```



Fonseca-Fleming Characteristics:

- Concave Pareto front (minimization)
- Symmetric problem
- Scalable to higher dimensions

25.10. Example 14: Kursawe - Non-Convex Disconnected Front

Optimize Kursawe with min-max scalarization:

```
from spotoptim.function.mo import kursawe
def min_max_kursawe(y_mo):
    """Minimize maximum objective (Chebyshev)."""
    return np.max(y_mo, axis=1)

optimizer = SpotOptim(
    fun=kursawe,
    bounds=[(-5, 5)] * 3,
    max_iter=50,
    n_initial=20,
    fun_mo2so=min_max_kursawe,
    seed=42
)

result = optimizer.optimize()

y_best = kursawe(result.x.reshape(1, -1))
print(f"\nOptimization Result:")
print(f"Best x: {result.x}")
print(f"f1 = {y_best[0, 0]:.4f}, f2 = {y_best[0, 1]:.4f}")
print(f"Max objective: {max(y_best[0]):.4f}")
```

```
Optimization Result:
Best x: [-0.58740866 -1.11955526 -1.11461838]
f1 = -15.0566, f2 = -8.0116
Max objective: -8.0116
```

25.10.1. Comparison: Different Scalarization Strategies on ZDT1

Let's compare how different scalarization strategies find different Pareto solutions:

```
# Define different scalarizations
scalarizations = {
    'First Objective': None, # Default
    'Equal Weight': lambda y: 0.5 * y[:, 0] + 0.5 * y[:, 1],
    'Favor f1': lambda y: 0.8 * y[:, 0] + 0.2 * y[:, 1],
```

25. Multi-Objective Optimization Support in SpotOptim

```

    'Favor f2': lambda y: 0.2 * y[:, 0] + 0.8 * y[:, 1],
    'Min-Max': lambda y: np.max(y, axis=1),
}

results = {}
for name, scalarization in scalarizations.items():
    opt = SpotOptim(
        fun=zdt1,
        bounds=[(0, 1)] * 30,
        max_iter=40,
        n_initial=15,
        fun_mo2so=scalarization,
        seed=42,
        verbose=0
    )
    res = opt.optimize()
    y_res = zdt1(res.x.reshape(1, -1))
    results[name] = (res.x, y_res[0])

# Visualize results
fig, ax = plt.subplots(figsize=(10, 6))

# Plot Pareto front
X_pareto = np.column_stack([x1_range, np.zeros((n_points, 29))])
y_pareto = zdt1(X_pareto)
ax.plot(y_pareto[:, 0], y_pareto[:, 1], 'k-', linewidth=2,
        label='True Pareto Front', alpha=0.3)

# Plot optimization results
colors = ['blue', 'green', 'orange', 'purple', 'red']
for (name, (x, y)), color in zip(results.items(), colors):
    ax.scatter(y[0], y[1], s=100, c=color, marker='*',
               label=f'{name}: ({y[0]:.3f}, {y[1]:.3f})', zorder=5)

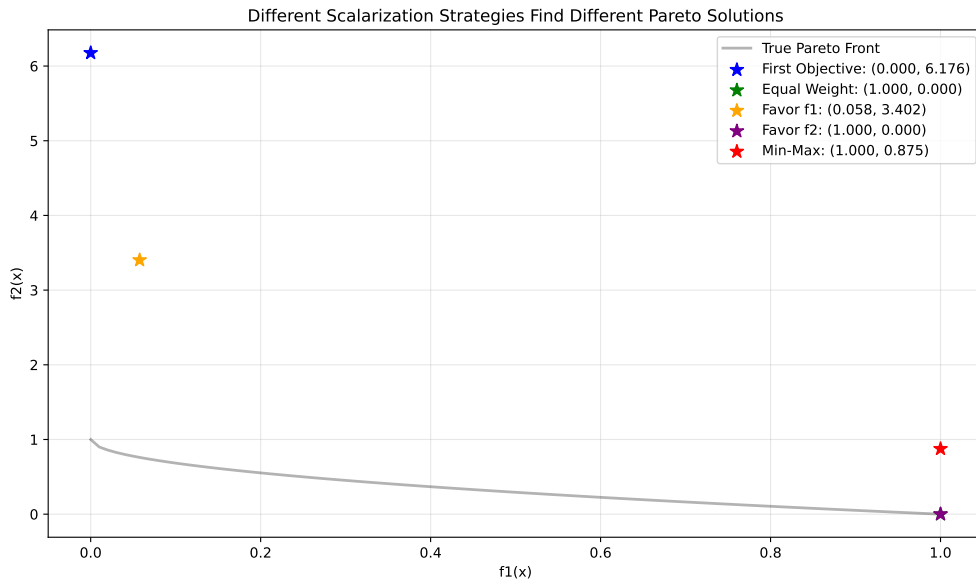
ax.set_xlabel('f1(x)')
ax.set_ylabel('f2(x)')
ax.set_title('Different Scalarization Strategies Find Different Pareto Solutions')
ax.legend()
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print("\nComparison Results:")
for name, (x, y) in results.items():

```


25.11. A Simple Visualization Example

```
print(f"{name:20s}: f1={y[0]:.4f}, f2={y[1]:.4f}")
```



Comparison Results:

First Objective	: f1=0.0000, f2=6.1756
Equal Weight	: f1=1.0000, f2=0.0000
Favor f1	: f1=0.0576, f2=3.4021
Favor f2	: f1=1.0000, f2=0.0000
Min-Max	: f1=1.0000, f2=0.8750

25.11. A Simple Visualization Example

The `mo_mm_desirability_function` computes the negative combined desirability for a candidate point x . It predicts the objective values using the provided surrogate models, computes the Morris-Mitchell improvement if requested, and then calculates the overall desirability using the provided `DOverall` object. The function returns the negative desirability (for minimization) and the list of individual objective values. Consider the following example usage:

Generate `X_base` in the range $[0,1]$ as input data and evaluate the two-objective maximization function `mo_conv2_max`.

A smooth, convex two-objective maximization problem on $[0, 1]^2$ using flipped versions of the minimization objectives:

25. Multi-Objective Optimization Support in SpotOptim

$$f1(x, y) = 2 - (x^2 + y^2) \quad (25.1)$$

$$f2(x, y) = 2 - ((1 - x)^2 + (1 - y)^2) \quad (25.2)$$

It has the following properties:

- Domain: $[0, 1]^2$
- Objectives: maximize both $f1$ and $f2$
- Ideal points: $(0, 0)$ for $f1$ (gives $f1 = 2$); $(1, 1)$ for $f2$ (gives $f2 = 2$)
- Pareto set: line $x = y$ in $[0, 1]$
- Pareto front: convex quadratic trade-off $f1 = 2 - 2t^2$, $f2 = 2 - 2(1 - t)^2$, $t \in [0, 1]$

```
X_base = np.random.rand(500, 2)
y = mo_conv2_max(X_base)
```

The RandomForestRegressor models are fitted for each objective.

```
models = []
for i in range(y.shape[1]):
    model = RandomForestRegressor(n_estimators=n_estimators, random_state=random_state)
    model.fit(X_base, y[:, i])
    models.append(model)
```

```
# calculate base Morris-Mitchell stats
phi_base, J_base, d_base = mmphi_intensive(X_base, q=2, p=2)
print(f"phi_base: {phi_base}, J_base: {J_base}, d_base: {d_base}")
```

```
phi_base: 7.664691627178843, J_base: [1 1 1 ... 1 1 1], d_base: [6.89559224e-
04 1.02332316e-03 1.77989575e-03 ... 1.35275283e+00
1.37109144e+00 1.37655463e+00]
```

Figure 25.1 shows the surfaces of the two objectives using the fitted models.

```
x_min = 0
x_max = 1
combinations = [(0, 1)]
target_names = ["y1", "y2"]

mo_xy_surface(models, bounds=[(x_min, x_max), (x_min, x_max)], target_names=target_names)
```

25.11. A Simple Visualization Example

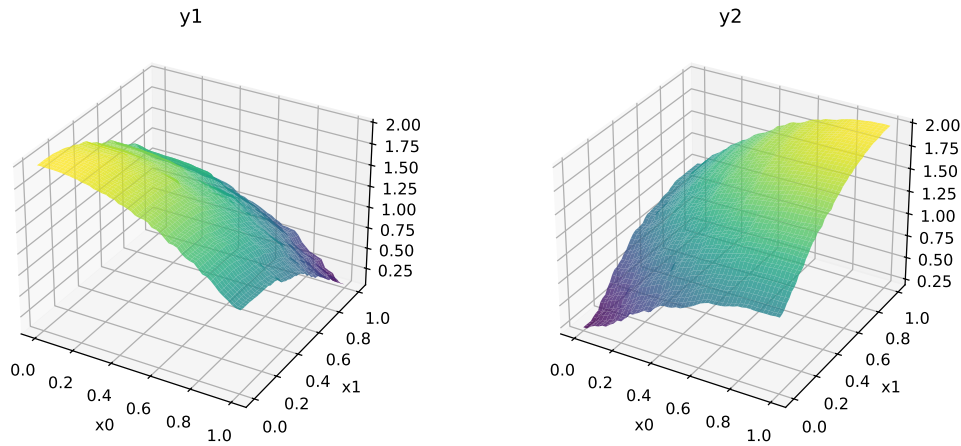


Figure 25.1.: Surfaces of the two objectives of the `mo_conv2_max` function

Figure 25.2 shows the contours of the two objectives using the fitted models.

```
x_min = 0
x_max = 1
mo_xy_contour(models, bounds=[(x_min, x_max), (x_min, x_max)], target_names=target_names)
```

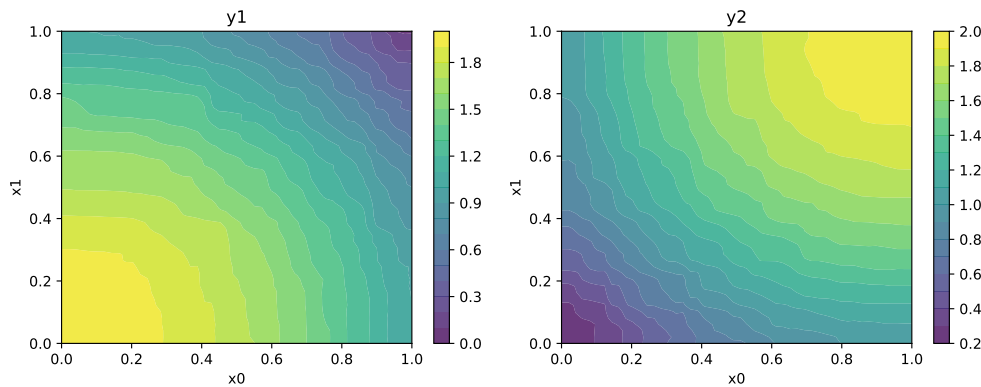


Figure 25.2.: Contours of the two objectives of the `mo_conv2_max` function

Figure 25.3 shows the Pareto front of the original data.

```
plot_mo(y_orig=y, target_names=target_names, combinations=combinations, pareto="max", pareto_front_c
```

Figure 25.4 visualizes the Pareto-optimal points, i.e., the points in the input space that correspond to the Pareto front:

382

25.11. A Simple Visualization Example

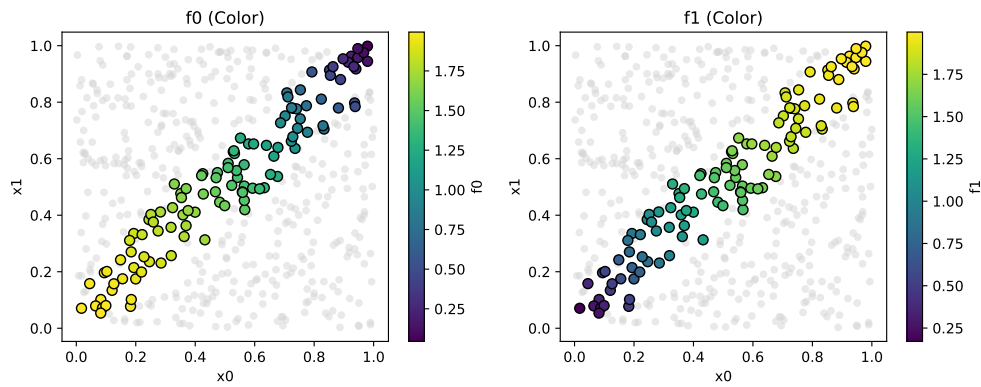


Figure 25.4.: Pareto-optimal points in the input space

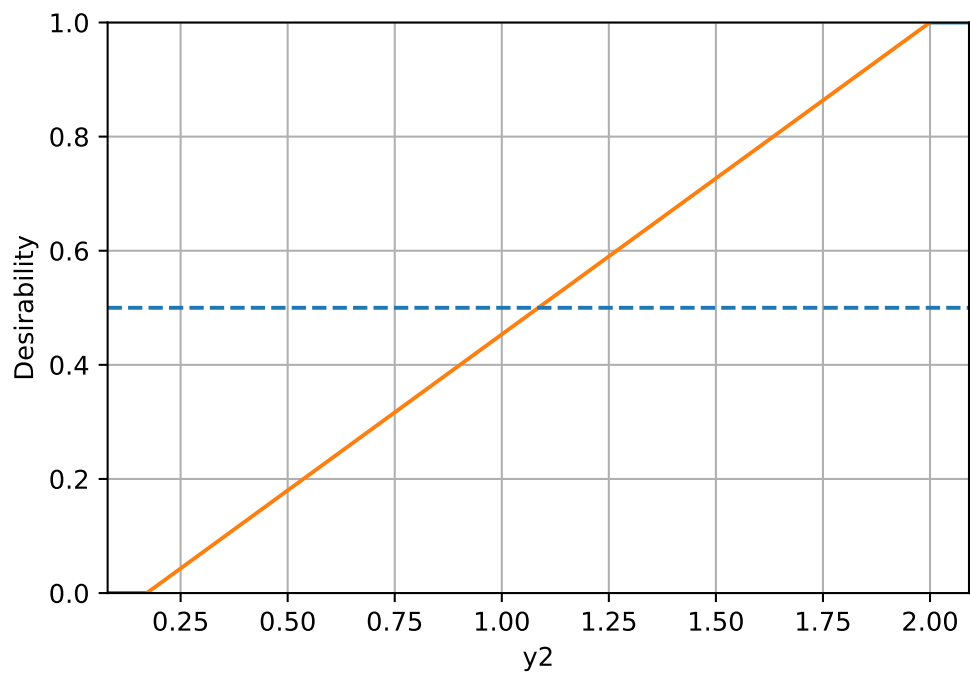
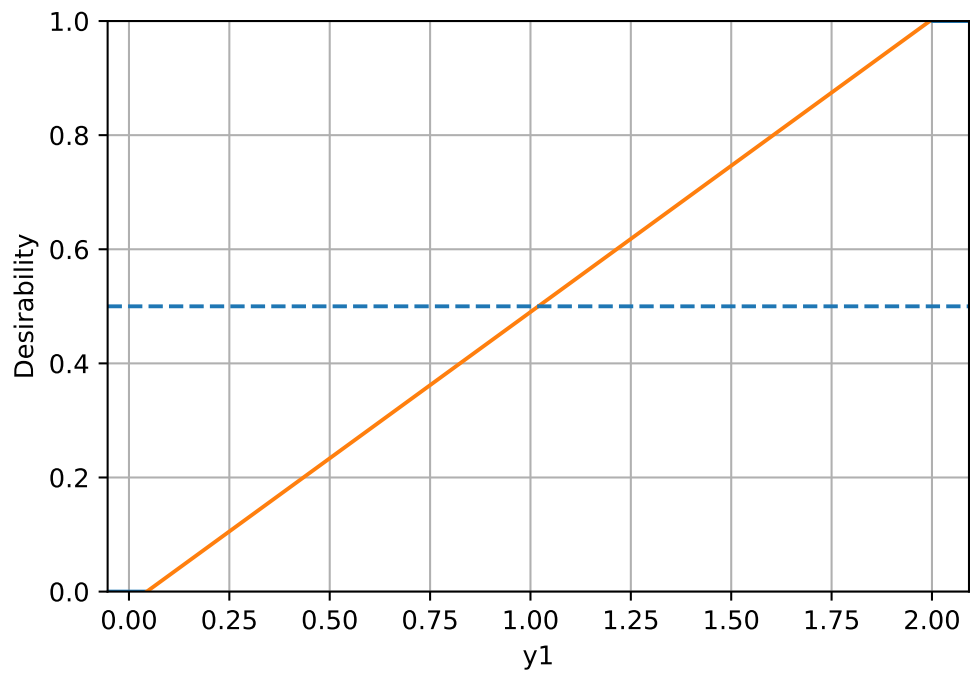
Compute desirability functions for each objective:

```
d_funcs = []
for i in range(y.shape[1]):
    d_func = DMax(low=np.min(y[:, i]), high=np.max(y[:, i]))
    d_funcs.append(d_func)
```

The desirability functions can be visualized as follows:

```
d_funcs[0].plot(xlabel=target_names[0], figsize=(6,4))
d_funcs[1].plot(xlabel=target_names[1], figsize=(6,4))
```

25. Multi-Objective Optimization Support in SpotOptim



25.11. A Simple Visualization Example

The minimal expected improvement of the Morris-Mitchell criterion is defined as 1 percent of the base value, and the maximal expected improvement as 10 percent of the base value. Then, the desirability function for Morris-Mitchell objective can be defined as follows:

```
# imp_min = phi_base * 0.01
# imp_max = phi_base * 0.1

# d_mm = DMax(low=imp_min, high=imp_max)
# d_mm.plot()
```

Combine into overall desirability

```
# D_overall = DOverall(*d_funcs, d_mm)
D_overall = DOverall(*d_funcs)
```

Now we can call the `mo_mm_desirability_function` with a test point:

```
x_test = np.array([0.5, 0.5]) # Example test point
neg_D, objectives = mo_mm_desirability_function(x_test, models, X_base, J_base, d_base, phi_base, D_
print(f"Negative Desirability: {neg_D}")
print(f"Objectives: {objectives}")
```

```
Negative Desirability: -0.7389272646499959
Objectives: [np.float64(1.4891072624235804), np.float64(1.5188221414320862)]
```

```
mo_xy_desirability_plot(models, bounds=[(x_min, x_max), (x_min, x_max)], X_base=X_base, J_base=J_base)
```

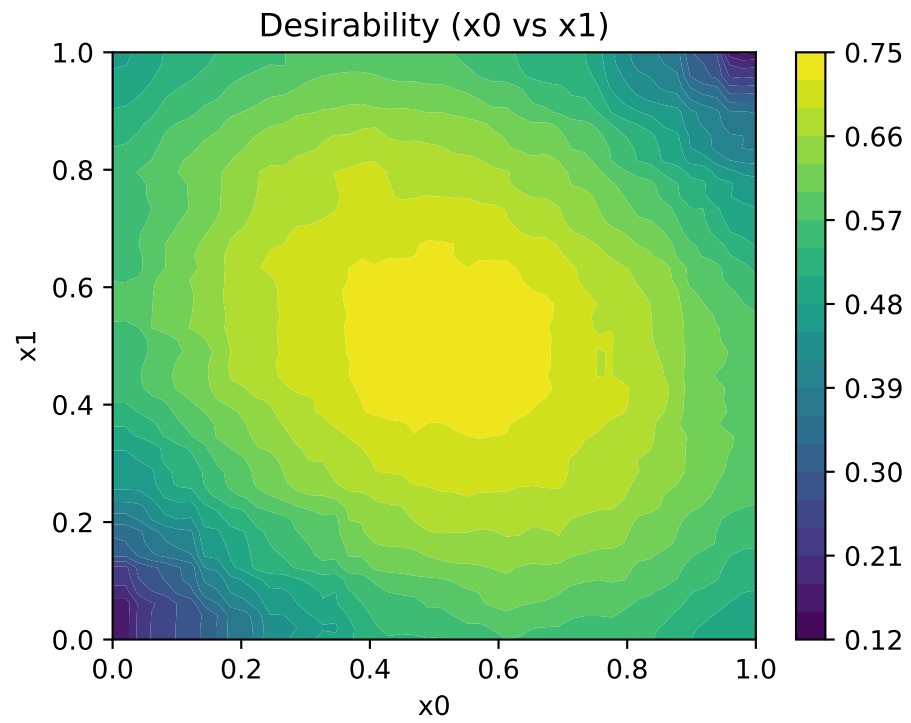


Figure 25.5.: Desirability function for multi-objective optimization

25.12. Jupyter Notebook

Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

Part VIII.

Hyperparameter Tuning

26. Physics-Informed Neural Networks (PINNs) Demo 1

Solving ODEs with SpotOptim's LinearRegressor

27. Overview

This tutorial demonstrates how to use Physics-Informed Neural Networks (PINNs) to solve ordinary differential equations (ODEs) using SpotOptim's **LinearRegressor** class. We'll solve a first-order ODE with an initial condition, combining data-driven learning with physics constraints.

27.1. The Differential Equation

We want to solve the following ODE:

$$\frac{dy}{dt} + 0.1y - \sin\left(\frac{\pi t}{2}\right) = 0$$

with initial condition:

$$y(0) = 0$$

28. Setup

First, let's import the necessary libraries:

```
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
from typing import Tuple
from spoptim.nn.linear_regressor import LinearRegressor

# Set random seed for reproducibility
torch.manual_seed(123)
```

```
<torch._C.Generator at 0x10e126770>
```


29. The Neural Network

We'll use SpotOptim's `LinearRegressor` class to create a neural network with:

- 1 input (time t)
- 1 output (solution y)
- 32 neurons per hidden layer
- 3 hidden layers
- Tanh activation function

```
model = LinearRegressor(  
    input_dim=1,  
    output_dim=1,  
    l1=32,  
    num_hidden_layers=3,  
    activation="Tanh",  
    lr=1.0  
)  
  
# Get optimizer with custom learning rate  
optimizer = model.get_optimizer("Adam", lr=3.0) # 3.0 * 0.001 = 0.003  
  
print(f"Model architecture:")  
print(model)
```

Model architecture:

```
LinearRegressor(  
  (activation): Tanh()  
  (network): Sequential(  
    (0): Linear(in_features=1, out_features=32, bias=True)  
    (1): Tanh()  
    (2): Linear(in_features=32, out_features=32, bias=True)  
    (3): Tanh()  
    (4): Linear(in_features=32, out_features=32, bias=True)  
    (5): Tanh()  
    (6): Linear(in_features=32, out_features=1, bias=True)  
  )  
)
```


30. Generate Training Data

We'll generate exact solution data using a numerical solver (RK2 method):

```
def oscillator(
    n_steps: int = 3000,
    t_min: float = 0.0,
    t_max: float = 30.0,
    y0: float = 0.0,
    alpha: float = 0.1,
    omega: float = np.pi / 2
) -> Tuple[torch.Tensor, torch.Tensor]:
    """
    Solve ODE:  $dy/dt + \alpha y - \sin(\omega t) = 0$ 
    using RK2 (midpoint method).

    Args:
        n_steps: Number of time steps
        t_min: Start time
        t_max: End time
        y0: Initial condition  $y(t_{\min})$ 
        alpha: Damping coefficient
        omega: Forcing frequency

    Returns:
        Tuple of (time points, solution values) as torch tensors
    """
    t_step = (t_max - t_min) / n_steps
    t_points = np.arange(t_min, t_min + n_steps * t_step, t_step)[:n_steps]

    y = [y0]

    for t_current_step_end in t_points[1:]:
        t_midpoint = t_current_step_end - t_step / 2.0
        y_prev = y[-1]

        # Stage 1: intermediate value
        slope_at_t_mid = -alpha * y_prev + np.sin(omega * t_midpoint)
```

30. Generate Training Data

```
y_intermediate = y_prev + (t_step / 2.0) * slope_at_t_mid

# Stage 2: final value
slope_at_t_end = -alpha * y_intermediate + np.sin(omega * t_current_step_end)
y_next = y_prev + t_step * slope_at_t_end
y.append(y_next)

t_tensor = torch.tensor(t_points, dtype=torch.float32).view(-1, 1)
y_tensor = torch.tensor(y, dtype=torch.float32).view(-1, 1)

return t_tensor, y_tensor
```

Generate the exact solution and sample training data points:

```
# Generate exact solution (3000 points)
x, y = oscillator()

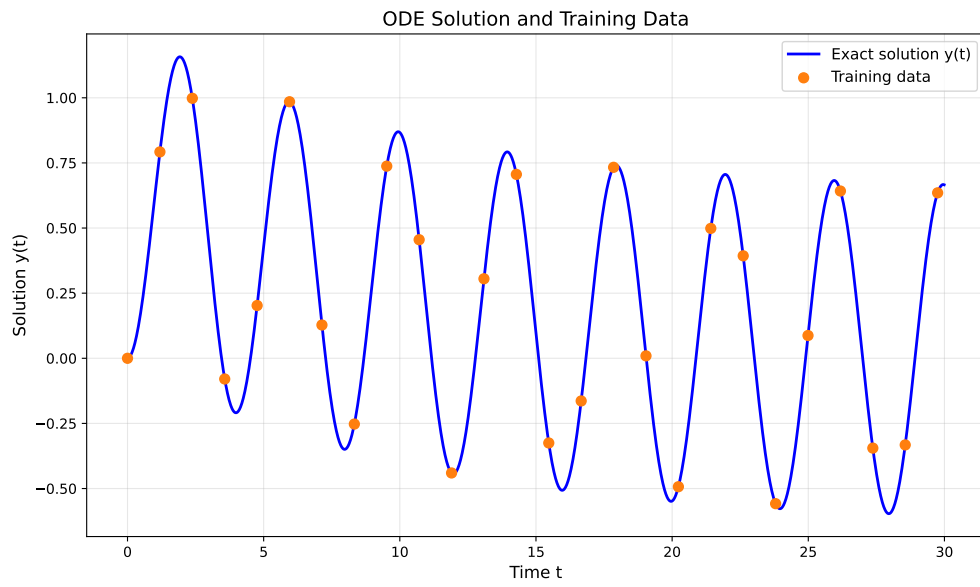
# Sample training data (every 119th point, giving ~25 points)
x_data = x[0:3000:119]
y_data = y[0:3000:119]

print(f"Exact solution points: {len(x)}")
print(f"Training data points: {len(x_data)}")
```

Exact solution points: 3000
Training data points: 26

Visualize the exact solution and training data:

```
plt.figure(figsize=(10, 6))
plt.plot(x.numpy(), y.numpy(), 'b-', linewidth=2, label="Exact solution y(t)")
plt.scatter(x_data.numpy(), y_data.numpy(), color="tab:orange",
            s=50, label="Training data", zorder=5)
plt.xlabel("Time t", fontsize=12)
plt.ylabel("Solution y(t)", fontsize=12)
plt.title("ODE Solution and Training Data", fontsize=14)
plt.legend(fontsize=11)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```



31. Collocation Points

Create collocation points where we'll enforce the physics (ODE) constraints:

```
# 50 evenly spaced points in [0, 30]
x_physics = torch.linspace(0, 30, 50).view(-1, 1).requires_grad_(True)

print(f"Collocation points: {len(x_physics)}")
print(f"Range: [{x_physics.min().item():.2f}, {x_physics.max().item():.2f}]" )
```

```
Collocation points: 50
Range: [0.00, 30.00]
```


32. PINN Training

Now we'll train the neural network using two loss components:

1. **Data Loss (Loss1)**: Ensures the network fits the training data
2. **Physics Loss (Loss2)**: Ensures the network satisfies the ODE

32.1. Loss Components Explained

32.1.1. Neural Network Prediction on Data Points

The network predicts values at the training data points:

```
yh = model(x_data)
```

32.1.2. Data Loss (Loss1)

Mean squared error between predictions and actual data:

```
loss1 = torch.mean((yh - y_data)**2)
```

32.1.3. Neural Network Prediction on Collocation Points

The network predicts values at the physics enforcement points:

```
yhp = model(x_physics)
```

32.1.4. Computing Derivatives for the ODE

We compute dy/dt using automatic differentiation:

32. PINN Training

```
dyhp_dxphysics = torch.autograd.grad(
    yhp, x_physics,
    torch.ones_like(yhp),
    create_graph=True
)[0]
```

32.1.5. Physics Loss (Loss2)

The ODE residual at collocation points:

$$\text{residual} = \frac{dy}{dt} + 0.1y - \sin\left(\frac{\pi t}{2}\right)$$

```
physics = dyhp_dxphysics + 0.1 * yhp - torch.sin(np.pi * x_physics / 2)
loss2 = torch.mean(physics**2)
```

32.1.6. Total Loss

Combined loss with weighting factor :

```
loss = loss1 + alpha * loss2
```

32.2. Training Loop

```
loss_history_pinn = []
loss2_history_pinn = []
plot_data_points_pinn = []

alpha = 6e-2 # Weight for physics loss
n_epochs = 48000

print("Training Physics-Informed Neural Network...")
print(f"Total epochs: {n_epochs}")
print(f"Physics loss weight (alpha): {alpha}")

for i in range(n_epochs):
    optimizer.zero_grad()

    # Data Loss: Fit the training data
```

```

yh = model(x_data)
loss1 = torch.mean((yh - y_data)**2)

# Physics Loss: Satisfy the ODE at collocation points
yhp = model(x_physics)
dyhp_dxphysics = torch.autograd.grad(
    yhp, x_physics,
    torch.ones_like(yhp),
    create_graph=True
)[0]
physics = dyhp_dxphysics + 0.1 * yhp - torch.sin(np.pi * x_physics / 2)
loss2 = torch.mean(physics**2)

# Total Loss
loss = loss1 + alpha * loss2
loss.backward()
optimizer.step()

# Store history every 100 steps
if (i + 1) % 100 == 0:
    loss_history_pinn.append(loss.detach())
    loss2_history_pinn.append(loss2.detach())

# Store snapshots for visualization every 10000 steps
if (i + 1) % 10000 == 0:
    current_yh_full = model(x).detach()
    plot_data_points_pinn.append({
        'yh': current_yh_full,
        'step': i + 1
    })
    print(f"Epoch {i+1}/{n_epochs}: Total Loss = {loss.item():.6f}, "
          f"Physics Loss = {loss2.item():.6f}")

print("Training completed!")

```

```

Training Physics-Informed Neural Network...
Total epochs: 48000
Physics loss weight (alpha): 0.06
Epoch 10000/48000: Total Loss = 0.000119, Physics Loss = 0.001705
Epoch 20000/48000: Total Loss = 0.000035, Physics Loss = 0.000473
Epoch 30000/48000: Total Loss = 0.000014, Physics Loss = 0.000225
Epoch 40000/48000: Total Loss = 0.000181, Physics Loss = 0.000391
Training completed!

```


33. Results Visualization

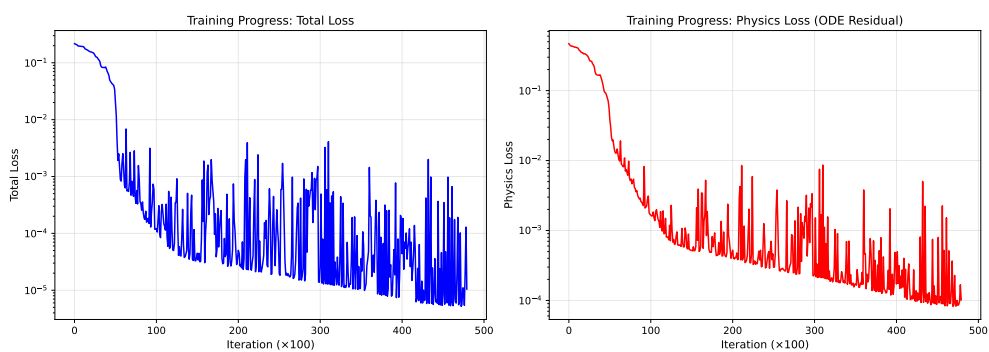
33.1. Training Progress

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

# Plot total loss
ax1.plot(loss_history_pinn, 'b-', linewidth=1.5)
ax1.set_xlabel('Iteration (×100)', fontsize=11)
ax1.set_ylabel('Total Loss', fontsize=11)
ax1.set_title('Training Progress: Total Loss', fontsize=12)
ax1.grid(True, alpha=0.3)
ax1.set_yscale('log')

# Plot physics loss
ax2.plot(loss2_history_pinn, 'r-', linewidth=1.5)
ax2.set_xlabel('Iteration (×100)', fontsize=11)
ax2.set_ylabel('Physics Loss', fontsize=11)
ax2.set_title('Training Progress: Physics Loss (ODE Residual)', fontsize=12)
ax2.grid(True, alpha=0.3)
ax2.set_yscale('log')

plt.tight_layout()
plt.show()
```



33.2. Solution Evolution During Training

Visualize how the neural network solution evolves during training:

```

xp_plot = x_physics.detach()

for plot_info in plot_data_points_pinn:
    plt.figure(figsize=(10, 6))

    # Plot exact solution
    plt.plot(x.numpy(), y.numpy(), 'b-', linewidth=2,
             label='Exact solution', alpha=0.7)

    # Plot training data
    plt.scatter(x_data.numpy(), y_data.numpy(), color='tab:orange',
               s=50, label='Training data', zorder=5)

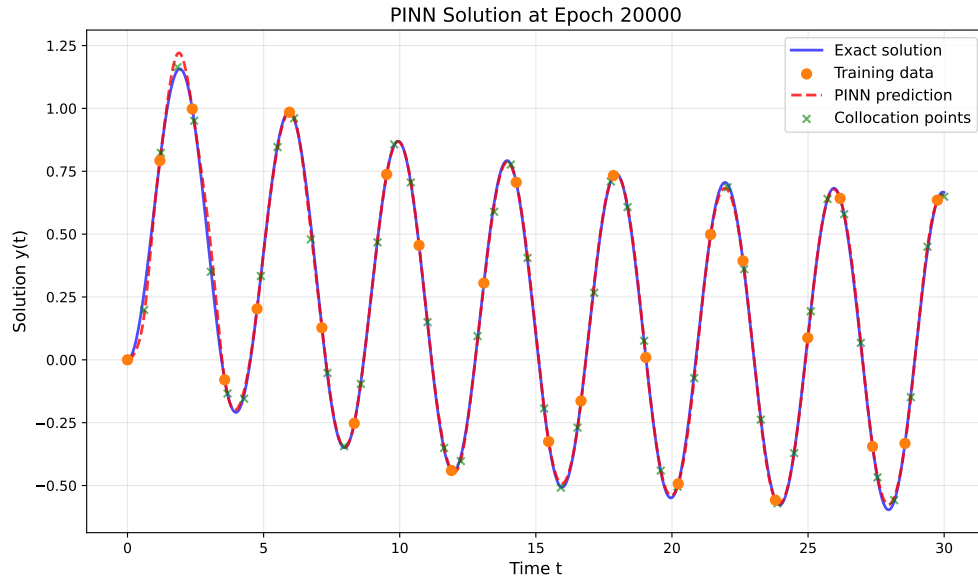
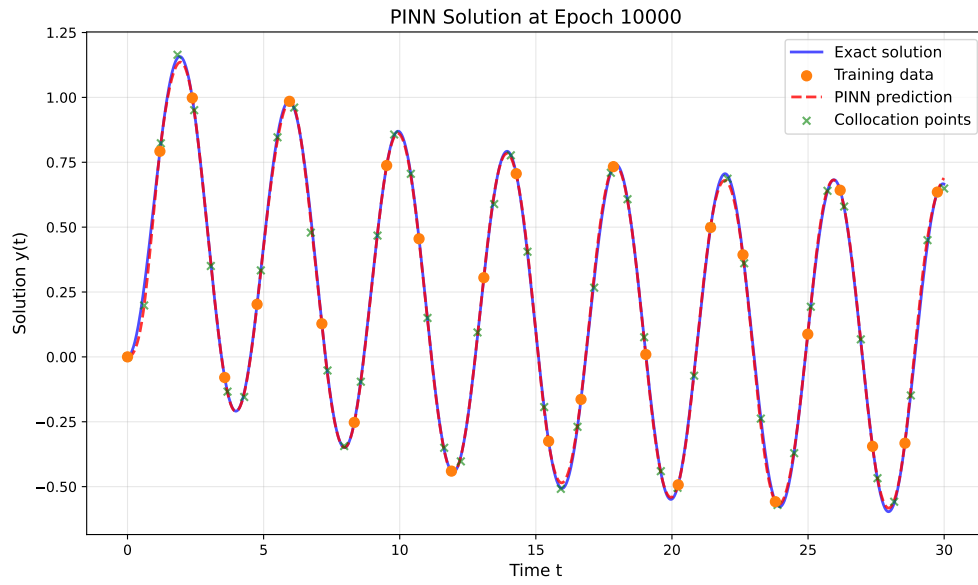
    # Plot neural network prediction
    plt.plot(x.numpy(), plot_info['yh'].numpy(), 'r--', linewidth=2,
             label='PINN prediction', alpha=0.8)

    # Plot collocation points
    plt.scatter(xp_plot.numpy(),
               model(xp_plot).detach().numpy(),
               color='green', marker='x', s=30,
               label='Collocation points', alpha=0.6, zorder=4)

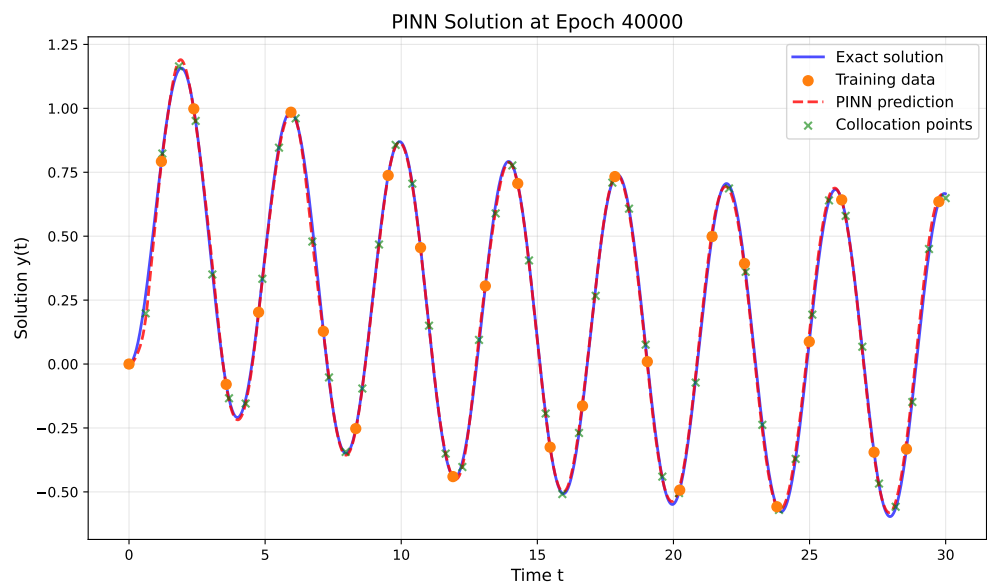
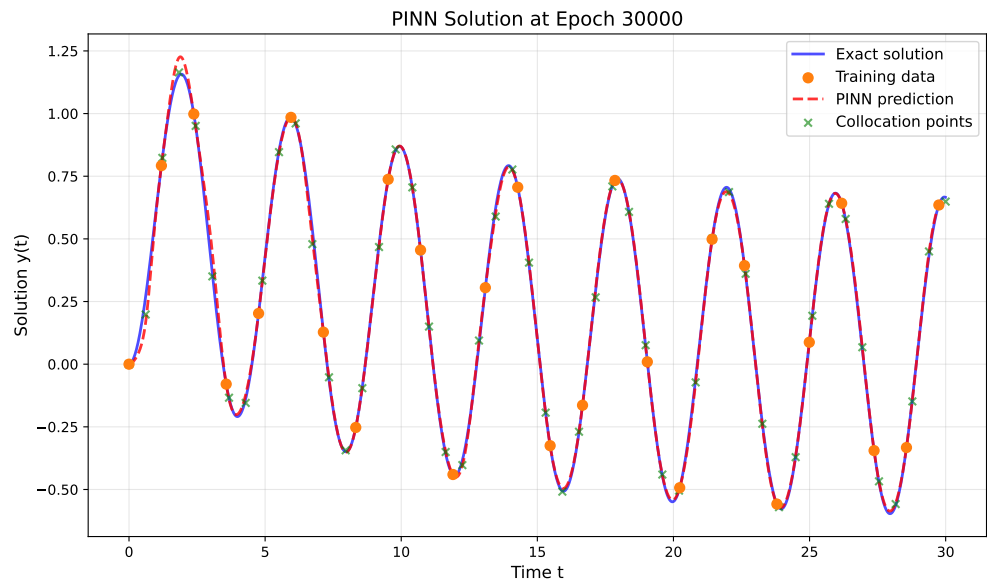
    plt.xlabel('Time t', fontsize=12)
    plt.ylabel('Solution y(t)', fontsize=12)
    plt.title(f'PINN Solution at Epoch {plot_info["step"]}', fontsize=14)
    plt.legend(fontsize=11)
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.show()

```

33.2. Solution Evolution During Training



33. Results Visualization



33.3. Final Solution Comparison

33.3. Final Solution Comparison

```
# Final prediction
model.eval()
with torch.no_grad():
    y_final = model(x)

plt.figure(figsize=(12, 7))

# Plot exact solution
plt.plot(x.numpy(), y.numpy(), 'b-', linewidth=2.5,
         label='Exact solution', alpha=0.8)

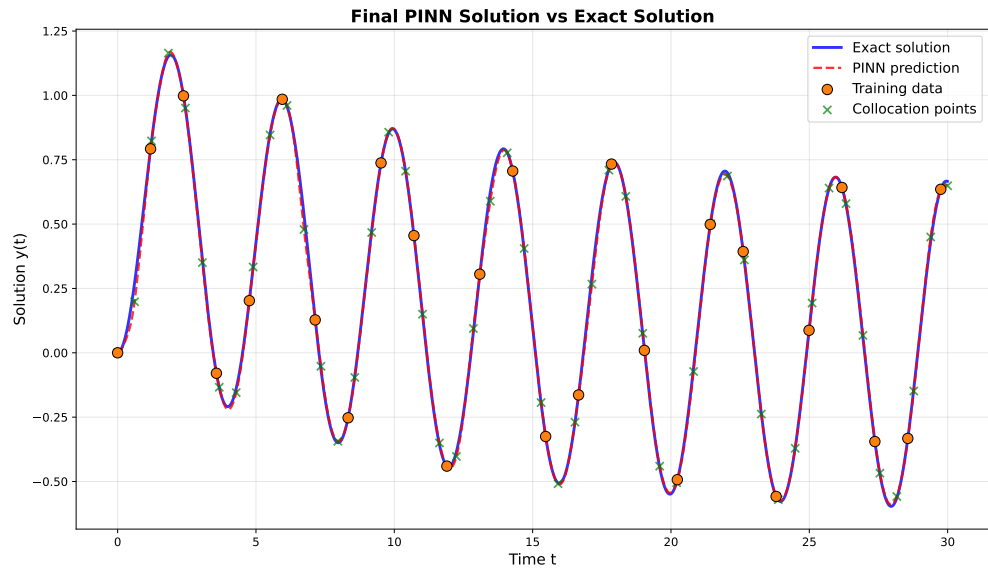
# Plot PINN prediction
plt.plot(x.numpy(), y_final.numpy(), 'r--', linewidth=2,
         label='PINN prediction', alpha=0.8)

# Plot training data
plt.scatter(x_data.numpy(), y_data.numpy(), color='tab:orange',
           s=80, label='Training data', zorder=5, edgecolors='black', linewidth=0.5)

# Plot collocation points
plt.scatter(x_physics.detach().numpy(),
           model(x_physics).detach().numpy(),
           color='green', marker='x', s=50,
           label='Collocation points', alpha=0.7, zorder=4)

plt.xlabel('Time t', fontsize=13)
plt.ylabel('Solution y(t)', fontsize=13)
plt.title('Final PINN Solution vs Exact Solution', fontsize=15, fontweight='bold')
plt.legend(fontsize=12, loc='best')
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

33. Results Visualization



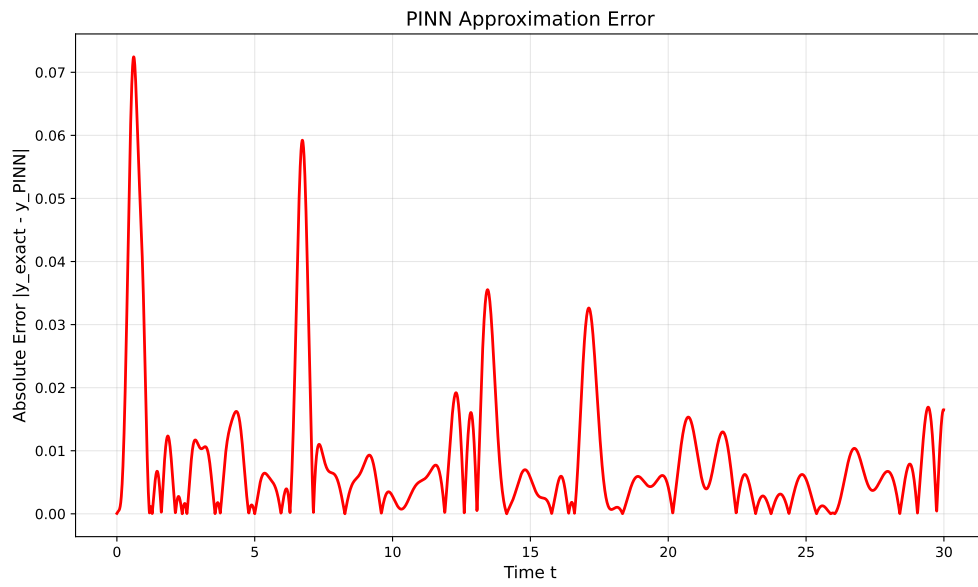
33.4. Error Analysis

```
# Compute absolute error
error = torch.abs(y_final - y)

plt.figure(figsize=(10, 6))
plt.plot(x.numpy(), error.numpy(), 'r-', linewidth=2)
plt.xlabel('Time t', fontsize=12)
plt.ylabel('Absolute Error |y_exact - y_PINN|', fontsize=12)
plt.title('PINN Approximation Error', fontsize=14)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print("\nError Statistics:")
print(f"Maximum absolute error: {error.max().item():.6f}")
print(f"Mean absolute error: {error.mean().item():.6f}")
print(f"Root mean squared error: {torch.sqrt(torch.mean(error**2)).item():.6f}")
```

33.4. Error Analysis



Error Statistics:

Maximum absolute error: 0.072458

Mean absolute error: 0.008300

Root mean squared error: 0.013641

34. Summary

This tutorial demonstrated how to use SpotOptim's `LinearRegressor` class to implement a Physics-Informed Neural Network (PINN) for solving ordinary differential equations. Key takeaways:

1. **Network Architecture:** Used a 3-layer neural network with 32 neurons per layer and Tanh activation
2. **Dual Loss Function:** Combined data fitting loss with physics constraint loss
3. **Automatic Differentiation:** Leveraged PyTorch's autograd to compute derivatives for the ODE
4. **Collocation Method:** Enforced the ODE at specific points in the domain
5. **Training Strategy:** Balanced data-driven and physics-informed learning with weight

The PINN successfully learned to approximate the ODE solution using only sparse training data (~25 points) by incorporating the underlying physics through the differential equation constraint.

34.1. Key Advantages of PINNs

- **Data Efficiency:** Can learn with very few data points
- **Physics Consistency:** Solutions automatically satisfy the governing equations
- **Generalization:** Better extrapolation beyond training data
- **Flexibility:** Can handle complex geometries and boundary conditions

34.2. Using SpotOptim for Hyperparameter Optimization

The `LinearRegressor` class integrates seamlessly with SpotOptim for hyperparameter tuning:

```
from spotoptim import SpotOptim

def train_pinn(X):
    """Objective function for hyperparameter optimization."""
    results = []
```

34. Summary

```
for params in X:
    l1 = int(params[0])          # Hidden layer size
    num_layers = int(params[1])  # Number of layers
    lr_unified = 10 ** params[2] # Learning rate (log scale)

    # Create model
    model = LinearRegressor(
        input_dim=1, output_dim=1,
        l1=l1, num_hidden_layers=num_layers,
        activation="Tanh", lr=lr_unified
    )

    # Train PINN and compute validation error
    # ... training code ...

    results.append(validation_error)
return np.array(results)

# Optimize hyperparameters
optimizer = SpotOptim(
    fun=train_pinn,
    bounds=[(16, 128), (1, 5), (-4, 0)],
    var_type=["int", "int", "float"],
    var_name=["layer_size", "num_layers", "log_lr"],
    max_iter=50
)
result = optimizer.optimize()
```

This approach allows you to systematically find the best network architecture and learning rate for your specific PINN problem.

34.3. Jupyter Notebook

Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

35. Hyperparameter Tuning for Physics-Informed Neural Networks

Using SpotOptim to Optimize PINN Architecture and Training

36. Overview

This tutorial demonstrates how to use SpotOptim for hyperparameter optimization of Physics-Informed Neural Networks (PINNs). We'll optimize the network architecture and training parameters to find the best configuration for solving an ordinary differential equation.

Building on the basic PINN demo, we'll now systematically search for optimal:

- Number of neurons per hidden layer
- Number of hidden layers
- Activation function (categorical)
- Optimizer algorithm (categorical)
- Learning rate (log-scale)
- Physics loss weight (log-scale)

36.1. Key Features

36.1.1. 1. PyTorch Dataset and DataLoader

Following PyTorch best practices from the official tutorial, this tutorial implements:

- **Custom Dataset Classes:** Separate classes for supervised data (`PINNDataset`) and collocation points (`CollocationDataset`)
- **DataLoader Integration:** Efficient batch processing with configurable batch size, shuffling, and parallel loading
- **Proper Data Separation:** Clean separation of training, validation, and collocation data
- **Gradient Tracking:** Automatic gradient handling for collocation points needed in physics loss

Benefits:

- **Modularity:** Clean separation between data and model code
- **Efficiency:** Batch processing and optional parallel data loading
- **Scalability:** Easy to extend to larger datasets
- **Best Practices:** Follows PyTorch conventions used across the ecosystem

36.1.2. 2. Automatic Transformation Handling

This tutorial also showcases SpotOptim's `var_trans` feature for automatic variable transformations. Learning rates and regularization parameters are often best explored on a log scale, but manually transforming values is tedious and error-prone. With `var_trans`, you simply specify:

```
var_trans = [None, None, "log10", "log10"]
```

SpotOptim then:

- Optimizes internally in log-transformed space (efficient exploration)
- Passes original-scale values to your objective function (no manual conversion needed)
- Displays all results in original scale (easy interpretation)

This eliminates the need for manual `10**x` conversions throughout your code!

37. The Problem

We're solving the same ODE as in the basic PINN demo:

$$\frac{dy}{dt} + 0.1y - \sin\left(\frac{\pi t}{2}\right) = 0$$

with initial condition $y(0) = 0$.

38. Setup

```
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
from typing import Tuple
from spotoptim import SpotOptim
from spotoptim.nn.linear_regressor import LinearRegressor
```

```
# Set random seed for reproducibility
torch.manual_seed(42)
np.random.seed(42)
```

Set number of epochs for training, maximum number of iterations for hyperparameter optimization, and initial design size. For this example we use a small number of epochs and iterations to keep the runtime short.

```
N_EPOCHS = 500
MAX_ITER = 15
N_INITIAL = 10
```


39. Data Generation

Following PyTorch best practices, we'll create custom Dataset classes for our PINN data.

39.1. Custom Dataset Classes

We'll create two dataset types:

1. PINNDataset for supervised data (training and validation)
2. CollocationDataset for physics-informed collocation points

```
from torch.utils.data import Dataset, DataLoader

def oscillator(
    n_steps: int = 3000,
    t_min: float = 0.0,
    t_max: float = 30.0,
    y0: float = 0.0,
    alpha: float = 0.1,
    omega: float = np.pi / 2
) -> Tuple[torch.Tensor, torch.Tensor]:
    """
    Solve ODE:  $dy/dt + \alpha y - \sin(\omega t) = 0$ 
    using RK2 (midpoint method).

    Returns:
        t_tensor: Time points, shape (n_steps, 1)
        y_tensor: Solution values, shape (n_steps, 1)
    """
    t_step = (t_max - t_min) / n_steps
    t_points = np.arange(t_min, t_min + n_steps * t_step, t_step)[:n_steps]

    y = [y0]

    for t_current_step_end in t_points[1:]:
        t_midpoint = t_current_step_end - t_step / 2.0
```

39. Data Generation

```
y_prev = y[-1]

slope_at_t_mid = -alpha * y_prev + np.sin(omega * t_midpoint)
y_intermediate = y_prev + (t_step / 2.0) * slope_at_t_mid

slope_at_t_end = -alpha * y_intermediate + np.sin(omega * t_current_step_end)
y_next = y_prev + t_step * slope_at_t_end
y.append(y_next)

t_tensor = torch.tensor(t_points, dtype=torch.float32).view(-1, 1)
y_tensor = torch.tensor(y, dtype=torch.float32).view(-1, 1)

return t_tensor, y_tensor
```

```
class PINNDataset(Dataset):
    """PyTorch Dataset for PINN supervised data (training/validation).

    This dataset stores time-solution pairs (t, y) for supervised learning.

    Args:
        t (torch.Tensor): Time points, shape (n_samples, 1)
        y (torch.Tensor): Solution values, shape (n_samples, 1)
    """

    def __init__(self, t: torch.Tensor, y: torch.Tensor):
        self.t = t
        self.y = y

    def __len__(self) -> int:
        return len(self.t)

    def __getitem__(self, idx: int) -> Tuple[torch.Tensor, torch.Tensor]:
        return self.t[idx], self.y[idx]
```

```
class CollocationDataset(Dataset):
    """PyTorch Dataset for PINN collocation points.

    This dataset stores time points where physics loss is evaluated.
    Gradients are required for computing derivatives in the PDE.

    Args:
        t (torch.Tensor): Collocation time points, shape (n_points, 1)
```



```

"""

def __init__(self, t: torch.Tensor):
    # Store collocation points with gradient tracking
    self.t = t.requires_grad_(True)

def __len__(self) -> int:
    return len(self.t)

def __getitem__(self, idx: int) -> torch.Tensor:
    # Return single collocation point (still requires_grad)
    return self.t[idx].unsqueeze(0)

```

Generate exact solution using Runge-Kutta 2 (RK2) method:

```

x_exact, y_exact = oscillator()

# Create training data (sparse sampling)
t_train = x_exact[0:3000:119]
y_train = y_exact[0:3000:119]

# Create validation data (different sampling for unbiased evaluation)
t_val = x_exact[50:3000:120]
y_val = y_exact[50:3000:120]

# Create collocation points for physics loss
t_physics = torch.linspace(0, 30, 50).view(-1, 1)

# Create Dataset objects
train_dataset = PINNDataset(t_train, y_train)
val_dataset = PINNDataset(t_val, y_val)
collocation_dataset = CollocationDataset(t_physics)

print(f"Training dataset size: {len(train_dataset)}")
print(f"Validation dataset size: {len(val_dataset)}")
print(f"Collocation dataset size: {len(collocation_dataset)}")
print(f"\nSample from training dataset:")
t_sample, y_sample = train_dataset[0]
print(f"  t: {t_sample.item():.4f}, y: {y_sample.item():.4f}")

```

```

Training dataset size: 26
Validation dataset size: 25
Collocation dataset size: 50

```

39. Data Generation

```
Sample from training dataset:  
  t: 0.0000, y: 0.0000
```

40. Define the PINN Training Function

This function creates DataLoaders and trains a PINN with given hyperparameters. Following PyTorch best practices, we use DataLoader for efficient batch processing.

```
def train_pinn(
    l1: int,
    num_layers: int,
    activation: str,
    optimizer_name: str,
    lr_unified: float,
    alpha: float,
    n_epochs: int = N_EPOCHS,
    batch_size: int = 16,
    verbose: bool = False
) -> float:
    """
    Train a PINN with specified hyperparameters using DataLoaders.

    Args:
        l1: Number of neurons per hidden layer
        num_layers: Number of hidden layers
        activation: Activation function ("Tanh", "ReLU", "Sigmoid", "GELU")
        optimizer_name: Optimizer algorithm ("Adam", "SGD", "RMSprop", "AdamW")
        lr_unified: Unified learning rate multiplier
        alpha: Weight for physics loss
        n_epochs: Number of training epochs
        batch_size: Batch size for DataLoader
        verbose: Whether to print progress

    Returns:
        Validation mean squared error
    """
    # Set seed for reproducibility
    torch.manual_seed(42)

    # Create DataLoaders
    train_loader = DataLoader(
```

40. Define the PINN Training Function

```
        train_dataset,
        batch_size=batch_size,
        shuffle=True,
        num_workers=0,
        pin_memory=False
    )

    val_loader = DataLoader(
        val_dataset,
        batch_size=batch_size,
        shuffle=False,
        num_workers=0,
        pin_memory=False
    )

    # For collocation points, we can use full batch since it's small
    collocation_loader = DataLoader(
        collocation_dataset,
        batch_size=len(collocation_dataset),
        shuffle=False,
        num_workers=0
    )

    # Create model
    model = LinearRegressor(
        input_dim=1,
        output_dim=1,
        l1=l1,
        num_hidden_layers=num_layers,
        activation=activation,
        lr=lr_unified
    )

    # Get optimizer
    optimizer = model.get_optimizer(optimizer_name)

    # Training loop
    for epoch in range(n_epochs):
        model.train()
        epoch_loss = 0.0

        # Get collocation points (full batch)
        t_physics_batch = next(iter(collocation_loader))
        # Ensure gradients are enabled
```

```

t_physics_batch = t_physics_batch.requires_grad_(True)

# Iterate over training batches
for batch_t, batch_y in train_loader:
    optimizer.zero_grad()

    # Data Loss
    y_pred = model(batch_t)
    loss_data = torch.mean((y_pred - batch_y)**2)

    # Physics Loss (computed on full collocation set)
    y_physics = model(t_physics_batch)
    dy_dt = torch.autograd.grad(
        y_physics,
        t_physics_batch,
        torch.ones_like(y_physics),
        create_graph=True,
        retain_graph=True
    )[0]

    # PDE residual:  $dy/dt + 0.1*y - \sin(\pi*t/2) = 0$ 
    physics_residual = dy_dt + 0.1 * y_physics - torch.sin(np.pi * t_physics_batch / 2)
    loss_physics = torch.mean(physics_residual**2)

    # Total Loss
    loss = loss_data + alpha * loss_physics
    loss.backward()
    optimizer.step()

    epoch_loss += loss.item()

if verbose and (epoch + 1) % 2000 == 0:
    avg_loss = epoch_loss / len(train_loader)
    print(f" Epoch {epoch+1}/{n_epochs}: Avg Loss = {avg_loss:.6f}")

# Evaluate on validation set
model.eval()
val_mse = 0.0
with torch.no_grad():
    for batch_t, batch_y in val_loader:
        y_pred = model(batch_t)
        val_mse += torch.mean((batch_y - y_pred)**2).item()

val_mse /= len(val_loader)

```

40. Define the PINN Training Function

```
        return val_mse

# Test the function with default parameters
print("Testing PINN training function with DataLoaders...")
test_error = train_pinn(
    l1=32,
    num_layers=2,
    activation="Tanh",
    optimizer_name="Adam",
    lr_unified=3.0,
    alpha=0.06,
    batch_size=16,
    n_epochs=N_EPOCHS,
    verbose=True
)
print(f"\nTest validation MSE: {test_error:.6f}")
```

Testing PINN training function with DataLoaders...

Test validation MSE: 0.188752

41. Hyperparameter Optimization with SpotOptim

Now we'll use SpotOptim to find the best hyperparameters:

41.1. Define the Objective Function

```
def objective_pinn(X):
    """
    Objective function for SpotOptim.

    Args:
        X: Array of hyperparameter configurations, shape (n_configs, 6)
        Each row: [l1, num_layers, activation, optimizer, lr_unified, alpha]
        Note: SpotOptim handles log transformations and factor mapping automatically

    Returns:
        Array of validation errors
    """
    results = []

    for i, params in enumerate(X):
        # Extract parameters (already in original scale thanks to var_trans)
        # Factor variables (activation, optimizer) are returned as strings
        l1 = int(params[0])                # Number of neurons
        num_layers = int(params[1])        # Number of hidden layers
        activation = params[2]              # Activation function
        optimizer_name = params[3]         # Optimizer algorithm
        lr_unified = params[4]              # Learning rate
        alpha = params[5]                   # Physics weight

        print(f"\nConfiguration {i+1}/{len(X)}:")
        print(f"  l1={l1}, num_layers={num_layers}, activation={activation}, ")
        print(f"  optimizer={optimizer_name}, lr_unified={lr_unified:.4f}, alpha={alpha:.4f}")
```

41. Hyperparameter Optimization with SpotOptim

```
# Train PINN with these hyperparameters
val_error = train_pinn(
    l1=l1,
    num_layers=num_layers,
    activation=activation,
    optimizer_name=optimizer_name,
    lr_unified=lr_unified,
    alpha=alpha,
    n_epochs=N_EPOCHS,
    verbose=False
)

print(f" Validation MSE: {val_error:.6f}")
results.append(val_error)

return np.array(results)
```

We test the objective function with two configurations:

```
X_test = np.array([
    [32, 2, "Tanh", "Adam", 3.0, 0.06], # Baseline config
    [64, 3, "ReLU", "AdamW", 2.0, 0.04] # Alternative config
], dtype=object)
test_results = objective_pinn(X_test)
print(f"\nTest results: {test_results}")
```

Configuration 1/2:

```
l1=32, num_layers=2, activation=Tanh,
optimizer=Adam, lr_unified=3.0000, alpha=0.0600
Validation MSE: 0.188752
```

Configuration 2/2:

```
l1=64, num_layers=3, activation=ReLU,
optimizer=AdamW, lr_unified=2.0000, alpha=0.0400
Validation MSE: 0.169017
```

Test results: [0.18875158 0.16901682]

41.2. Run the Optimization

Use `tensorboard --logdir=runs` from a shell in the current directory (where this notebook is located) to visualize the optimization process.

Setting the bounds for the search space:

```
# Define search space with var_trans for automatic log-scale handling
bounds = [
    (16, 128),           # l1: neurons per layer (16 to 128)
    (1, 4),             # num_layers: 1 to 4 hidden layers
    ("Tanh", "ReLU", "Sigmoid", "GELU"), # activation: activation function
    ("Adam", "SGD", "RMSprop", "AdamW"), # optimizer: optimizer algorithm
    (0.1, 10.0),        # lr_unified: learning rate (0.1 to 10)
    (0.01, 1.0)         # alpha: physics weight (0.01 to 1.0)
]
```

Specify the variable types and transformations. Use `var_trans` to handle log-scale transformations automatically, factor variables don't need transformations (None):

```
var_type = ["int", "int", "factor", "factor", "float", "float"]
var_name = ["l1", "num_layers", "activation", "optimizer", "lr_unified", "alpha"]
var_trans = [None, None, None, None, "log10", "log10"]
```

```
# Create optimizer
optimizer = SpotOptim(
    fun=objective_pinn,
    bounds=bounds,
    var_type=var_type,
    var_name=var_name,
    var_trans=var_trans, # Automatic log-scale handling!
    max_iter=MAX_ITER,
    n_initial=N_INITIAL,
    seed=42,
    verbose=True,
    tensorboard_clean=True,
    tensorboard_log=True
)
```

Factor variable at dimension 2:

Levels: ['Tanh', 'ReLU', 'Sigmoid', 'GELU']

Mapped to integers: 0 to 3

Factor variable at dimension 3:

41. Hyperparameter Optimization with SpotOptim

```
Levels: ['Adam', 'SGD', 'RMSprop', 'AdamW']
Mapped to integers: 0 to 3
Removed old TensorBoard logs: runs/spotoptim_20251220_114016
Cleaned 1 old TensorBoard log directory
TensorBoard logging enabled: runs/spotoptim_20251220_114133
```

Display search space configuration. The `transcolumn` shows applied transformations. `lr_unified` and `alpha` use `log10` transformation internally. This enables efficient exploration of log-scale parameters. All values shown are in original scale (not transformed).

```
design_table = optimizer.print_design_table(tablefmt="github")
print(design_table)
```

name	type	lower	upper	default	transform
l1	int	16.0	128.0	72	-
num_layers	int	1.0	4.0	2	-
activation	factor	Tanh	GELU	Sigmoid	-
optimizer	factor	Adam	AdamW	RMSprop	-
lr_unified	float	0.1	10.0	5.05	log10
alpha	float	0.01	1.0	0.505	log10
name	type	lower	upper	default	transform
l1	int	16.0	128.0	72	-
num_layers	int	1.0	4.0	2	-
activation	factor	Tanh	GELU	Sigmoid	-
optimizer	factor	Adam	AdamW	RMSprop	-
lr_unified	float	0.1	10.0	5.05	log10
alpha	float	0.01	1.0	0.505	log10

Run optimization

```
result = optimizer.optimize()
```

Configuration 1/10:

```
l1=19, num_layers=3, activation=ReLU,
optimizer=SGD, lr_unified=9.5756, alpha=0.1603
Validation MSE: 0.196427
```

Configuration 2/10:

```
l1=30, num_layers=3, activation=ReLU,
```

41.2. Run the Optimization

```
optimizer=Adam, lr_unified=0.8430, alpha=0.0412  
Validation MSE: 0.190747
```

Configuration 3/10:

```
l1=110, num_layers=4, activation=ReLU,  
optimizer=AdamW, lr_unified=1.9457, alpha=0.3866  
Validation MSE: 0.172875
```

Configuration 4/10:

```
l1=74, num_layers=1, activation=Sigmoid,  
optimizer=RMSprop, lr_unified=0.1014, alpha=0.1050  
Validation MSE: 0.197574
```

Configuration 5/10:

```
l1=41, num_layers=3, activation=Sigmoid,  
optimizer=RMSprop, lr_unified=0.5877, alpha=0.7301  
Validation MSE: 0.196817
```

Configuration 6/10:

```
l1=52, num_layers=1, activation=Sigmoid,  
optimizer=RMSprop, lr_unified=0.2023, alpha=0.0145  
Validation MSE: 0.198142
```

Configuration 7/10:

```
l1=71, num_layers=2, activation=Sigmoid,  
optimizer=SGD, lr_unified=3.2551, alpha=0.4300  
Validation MSE: 0.204070
```

Configuration 8/10:

```
l1=120, num_layers=3, activation=ReLU,  
optimizer=AdamW, lr_unified=0.3330, alpha=0.0220  
Validation MSE: 0.185089
```

Configuration 9/10:

```
l1=98, num_layers=2, activation=Tanh,  
optimizer=SGD, lr_unified=4.3915, alpha=0.0293  
Validation MSE: 0.193439
```

Configuration 10/10:

```
l1=87, num_layers=2, activation=GELU,  
optimizer=SGD, lr_unified=1.4861, alpha=0.0949  
Validation MSE: nan
```

Warning: 1 initial design point(s) returned NaN/inf and will be ignored (reduced from 10 to 9 points)

Note: Initial design size (9) is smaller than requested (10) due to NaN/inf values

Initial best: $f(x) = 0.172875$

41. Hyperparameter Optimization with SpotOptim

Configuration 1/1:

l1=110, num_layers=4, activation=ReLU,
optimizer=AdamW, lr_unified=1.8630, alpha=0.3873
Validation MSE: 0.167599

Iteration 1: New best f(x) = 0.167599

Configuration 1/1:

l1=52, num_layers=3, activation=ReLU,
optimizer=RMSprop, lr_unified=0.1454, alpha=0.0135
Validation MSE: 0.184578

Iteration 2: f(x) = 0.184578

Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.

Configuration 1/1:

l1=115, num_layers=1, activation=Tanh,
optimizer=SGD, lr_unified=2.9394, alpha=0.0115
Validation MSE: nan

Warning: Found 1 NaN/inf value(s), replacing with adaptive penalty (max + 3*std = 0.2)

Iteration 3: f(x) = 0.287547

Configuration 1/1:

l1=111, num_layers=3, activation=ReLU,
optimizer=RMSprop, lr_unified=0.6980, alpha=0.1358
Validation MSE: 0.202440

Iteration 4: f(x) = 0.202440

Configuration 1/1:

l1=53, num_layers=1, activation=GELU,
optimizer=RMSprop, lr_unified=0.4545, alpha=0.0140
Validation MSE: 0.809881

Iteration 5: f(x) = 0.809881

Configuration 1/1:

l1=110, num_layers=4, activation=ReLU,
optimizer=AdamW, lr_unified=0.9493, alpha=0.3979
Validation MSE: 0.168582

Iteration 6: f(x) = 0.168582

TensorBoard writer closed. View logs with: tensorboard --logdir=runs/spotoptim_202512

42. Results Analysis

42.1. Best Configuration

Display best hyperparameters using `print_best()` method. With `var_trans`, results are already in original scale!

```
optimizer.print_best(result)
```

Best Solution Found:

```
-----  
l1: 110  
num_layers: 4  
activation: ReLU  
optimizer: AdamW  
lr_unified: 1.8630  
alpha: 0.3873  
Objective Value: 0.1676  
Total Evaluations: 15
```

Store values for later use in visualizations. Values are already in original scale thanks to `var_trans`. Factor variables are returned as strings.

```
best_l1 = int(result.x[0])  
best_num_layers = int(result.x[1])  
best_activation = result.x[2]  
best_optimizer = result.x[3]  
best_lr_unified = result.x[4]  
best_alpha = result.x[5]  
best_val_error = result.fun  
  
print(f"Best activation: {best_activation}")  
print(f"Best optimizer: {best_optimizer}")
```

```
Best activation: ReLU  
Best optimizer: AdamW
```

42.1.1. Results Table with Importance Scores

Display comprehensive results table with importance scores

```
table = optimizer.print_results_table(show_importance=True, tablefmt="github")
print(table)
```

name	type	default	lower	upper	tuned	transformer
l1	int	72	16.0	128.0	110	-
num_layers	int	2	1.0	4.0	4	-
activation	factor	Sigmoid	Tanh	GELU	ReLU	-
optimizer	factor	RMSprop	Adam	AdamW	AdamW	-
lr_unified	float	5.05	0.1	10.0	1.8630186679138154	log10
alpha	float	0.505	0.01	1.0	0.38731466430172934	log10

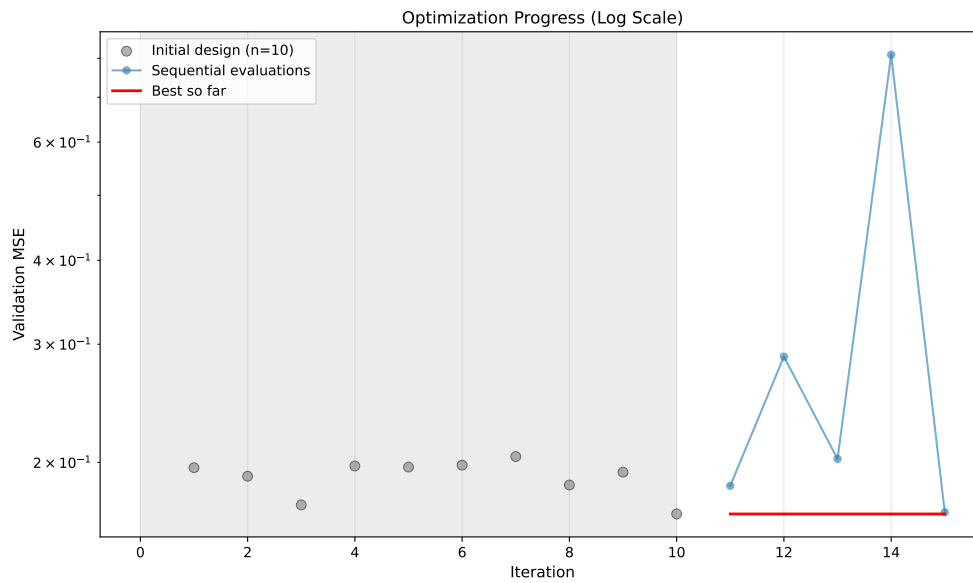
Interpretation: ***: >99%, **: >75%, *: >50%, .: >10%

name	type	default	lower	upper	tuned	transformer
l1	int	72	16.0	128.0	110	-
num_layers	int	2	1.0	4.0	4	-
activation	factor	Sigmoid	Tanh	GELU	ReLU	-
optimizer	factor	RMSprop	Adam	AdamW	AdamW	-
lr_unified	float	5.05	0.1	10.0	1.8630186679138154	log10
alpha	float	0.505	0.01	1.0	0.38731466430172934	log10

Interpretation: ***: >99%, **: >75%, *: >50%, .: >10%

42.2. Optimization History

```
optimizer.plot_progress(log_y=True, ylabel="Validation MSE")
```



42.3. Surrogate Visualization

Visualize the surrogate model's learned response surface for the most important hyperparameter combinations:

```
# Plot top 3 most important hyperparameter combinations
optimizer.plot_important_hyperparameter_contour(max_imp=3)
```

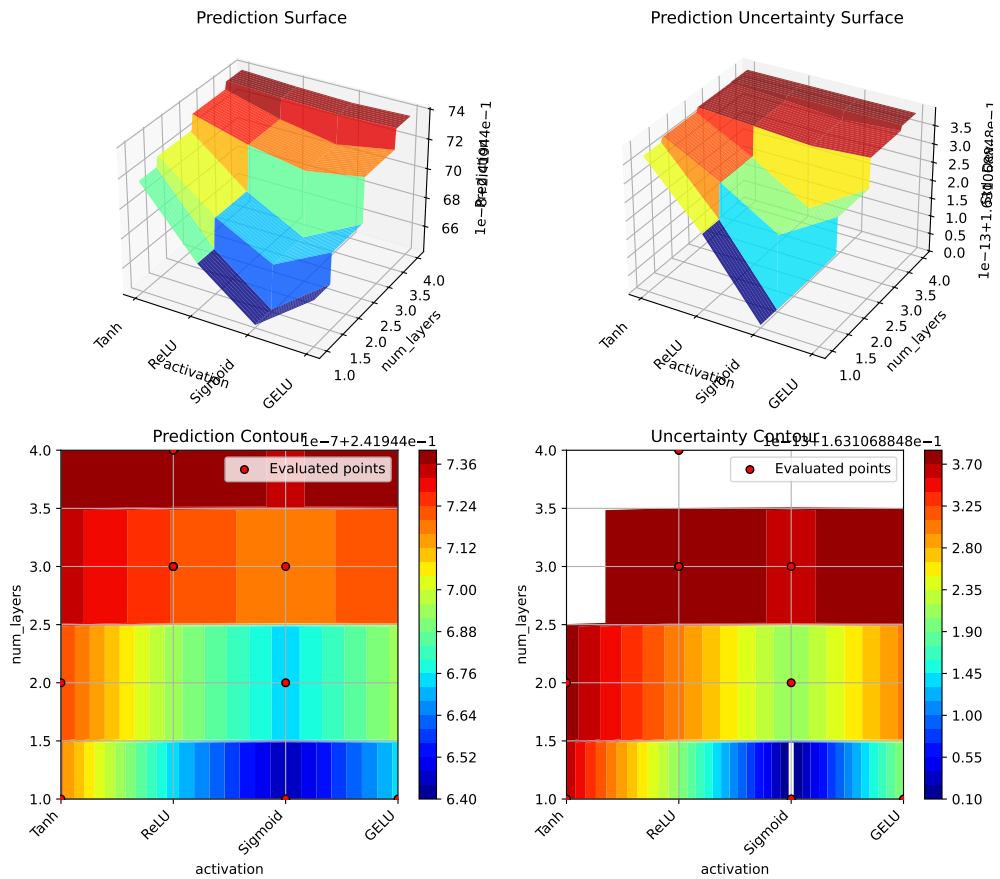
Plotting surrogate contours for top 3 most important parameters:

```
activation: importance = 33.13% (type: factor)
num_layers: importance = 28.95% (type: int)
alpha: importance = 16.38% (type: float)
```

Generating 3 surrogate plots...

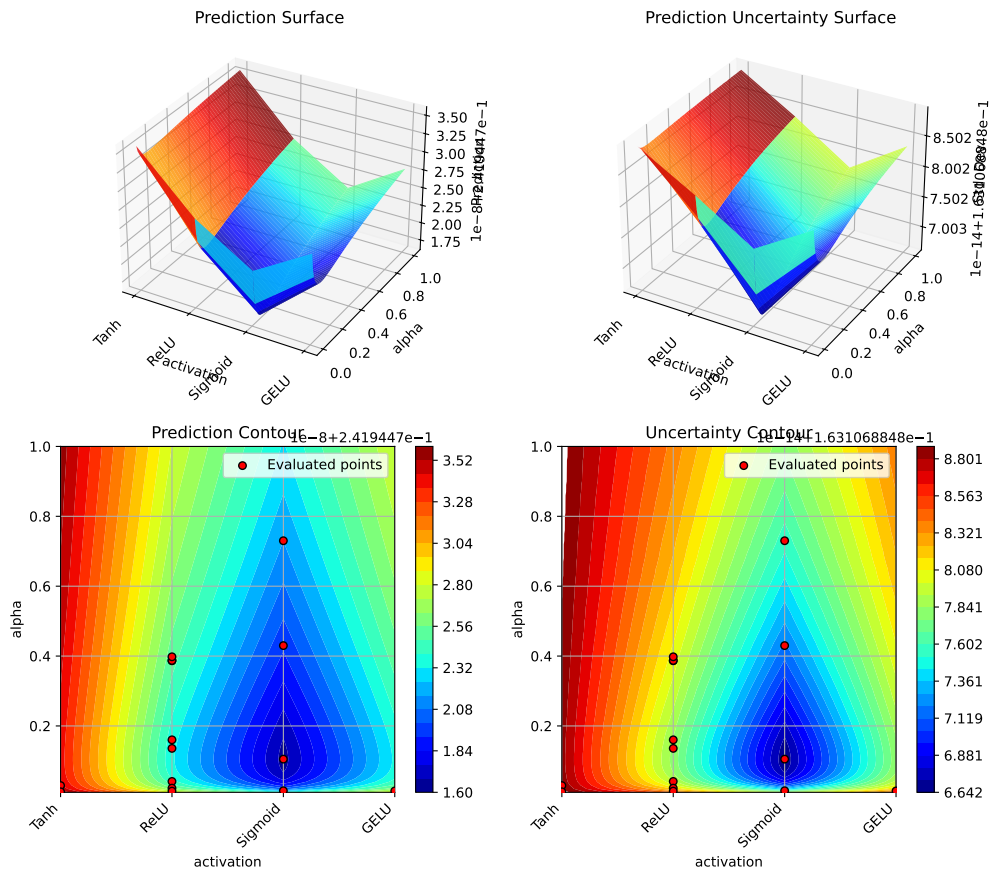
```
Plotting activation vs num_layers
```

42. Results Analysis



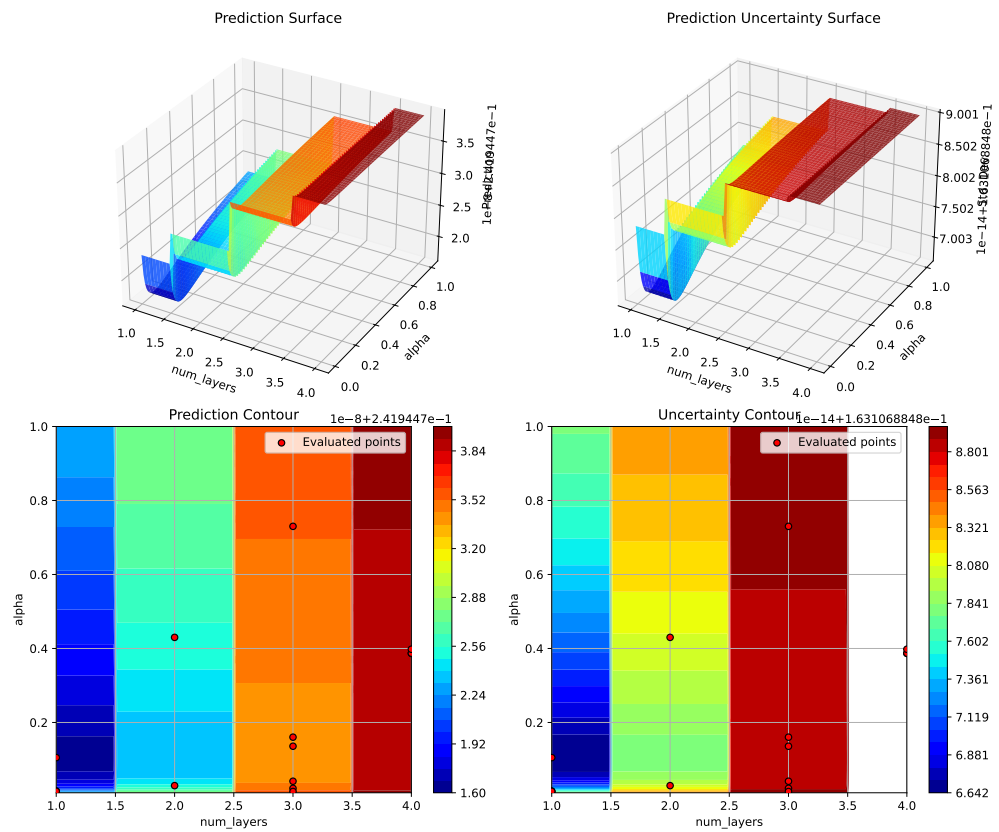
Plotting activation vs alpha

42.3. Surrogate Visualization



Plotting num_layers vs alpha

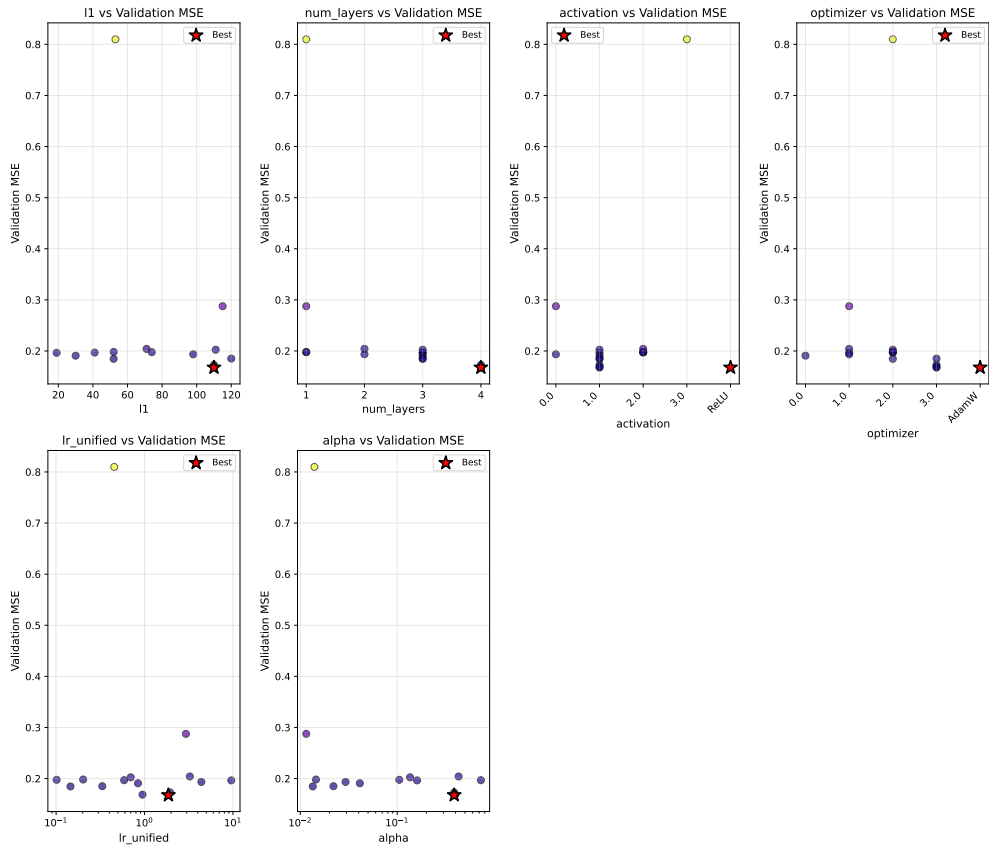
42. Results Analysis



42.4. Parameter Distribution Analysis

```
optimizer.plot_parameter_scatter(  
    result,  
    ylabel="Validation MSE",  
    cmap="plasma",  
    figsize=(14, 12)  
)
```

42.4. Parameter Distribution Analysis



43. Train Final Model with Best Hyperparameters

Now let's train a final model with the optimized hyperparameters using DataLoaders:

```
print("Training final model with best hyperparameters using DataLoaders...")
print(f"Training for 30,000 epochs...")

# Set seed for reproducibility
torch.manual_seed(42)

# Create DataLoaders for final training
final_batch_size = 16
train_loader_final = DataLoader(
    train_dataset,
    batch_size=final_batch_size,
    shuffle=True,
    num_workers=0
)

collocation_loader_final = DataLoader(
    collocation_dataset,
    batch_size=len(collocation_dataset),
    shuffle=False,
    num_workers=0
)

# Create model with best hyperparameters
final_model = LinearRegressor(
    input_dim=1,
    output_dim=1,
    l1=best_l1,
    num_hidden_layers=best_num_layers,
    activation=best_activation,
    lr=best_lr_unified
)
```

43. Train Final Model with Best Hyperparameters

```
optimizer_final = final_model.get_optimizer(best_optimizer)

# Training with history tracking
loss_history = []
n_epochs_final = 30000

for epoch in range(n_epochs_final):
    final_model.train()
    epoch_loss = 0.0

    # Get collocation points
    t_physics_batch = next(iter(collocation_loader_final))
    t_physics_batch = t_physics_batch.requires_grad_(True)

    # Iterate over training batches
    for batch_t, batch_y in train_loader_final:
        optimizer_final.zero_grad()

        # Data Loss
        y_pred = final_model(batch_t)
        loss_data = torch.mean((y_pred - batch_y)**2)

        # Physics Loss
        y_physics = final_model(t_physics_batch)
        dy_dt = torch.autograd.grad(
            y_physics,
            t_physics_batch,
            torch.ones_like(y_physics),
            create_graph=True,
            retain_graph=True
        )[0]

        physics_residual = dy_dt + 0.1 * y_physics - torch.sin(np.pi * t_physics_batch)
        loss_physics = torch.mean(physics_residual**2)

        # Total Loss
        loss = loss_data + best_alpha * loss_physics
        loss.backward()
        optimizer_final.step()

        epoch_loss += loss.item()

    # Record average loss every 100 epochs
    if (epoch + 1) % 100 == 0:
```

```

        avg_loss = epoch_loss / len(train_loader_final)
        loss_history.append(avg_loss)

    if (epoch + 1) % 5000 == 0:
        avg_loss = epoch_loss / len(train_loader_final)
        print(f"   Epoch {epoch+1}/{n_epochs_final}: Avg Loss = {avg_loss:.6f}")

print("Training completed!")

# Plot training history
plt.figure(figsize=(10, 5))
plt.plot(loss_history, linewidth=1.5)
plt.xlabel('Iteration (×100)', fontsize=11)
plt.ylabel('Average Total Loss per Batch', fontsize=11)
plt.title('Final Model Training History', fontsize=12)
plt.grid(True, alpha=0.3)
plt.yscale('log')
plt.tight_layout()
plt.show()

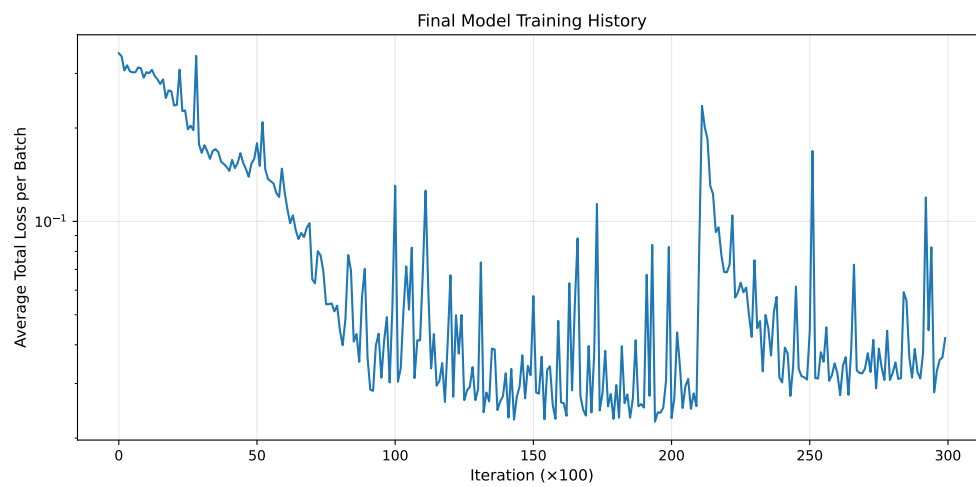
```

```

Training final model with best hyperparameters using DataLoaders...
Training for 30,000 epochs...
  Epoch 5000/30000: Avg Loss = 0.159071
  Epoch 10000/30000: Avg Loss = 0.053157
  Epoch 15000/30000: Avg Loss = 0.031900
  Epoch 20000/30000: Avg Loss = 0.082692
  Epoch 25000/30000: Avg Loss = 0.030881
  Epoch 30000/30000: Avg Loss = 0.042042
Training completed!

```

43. Train Final Model with Best Hyperparameters



44. Evaluate Final Model

```
# Create validation DataLoader for evaluation
val_loader_final = DataLoader(
    val_dataset,
    batch_size=len(val_dataset),
    shuffle=False,
    num_workers=0
)

# Evaluate on validation set
final_model.eval()
with torch.no_grad():
    # Validation MSE using DataLoader
    val_mse_total = 0.0
    for batch_t, batch_y in val_loader_final:
        y_pred = final_model(batch_t)
        val_mse_total += torch.mean((y_pred - batch_y)**2).item()
    final_val_mse = val_mse_total / len(val_loader_final)

    # Predict on full domain for visualization
    y_pred_full = final_model(x_exact)
    full_mse = torch.mean((y_pred_full - y_exact)**2).item()

    # Compute maximum absolute error
    max_error = torch.max(torch.abs(y_pred_full - y_exact)).item()

print("\nFinal Model Performance:")
print("-" * 50)
print(f"  Validation MSE: {final_val_mse:.6f}")
print(f"  Full domain MSE: {full_mse:.6f}")
print(f"  Maximum absolute error: {max_error:.6f}")
```

Final Model Performance:

Validation MSE: 0.014159

44. Evaluate Final Model

Full domain MSE: 0.009609
Maximum absolute error: 0.213946

45. Visualize Final Solution

```
# Generate predictions
final_model.eval()
with torch.no_grad():
    y_pred = final_model(x_exact)

fig, axes = plt.subplots(2, 1, figsize=(12, 10))

# Plot 1: Solution comparison
ax1 = axes[0]
ax1.plot(x_exact.numpy(), y_exact.numpy(), 'b-', linewidth=2.5,
        label='Exact solution', alpha=0.8)
ax1.plot(x_exact.numpy(), y_pred.numpy(), 'r--', linewidth=2,
        label='PINN prediction (optimized)', alpha=0.8)

# Plot training data from dataset
ax1.scatter(train_dataset.t.numpy(), train_dataset.y.numpy(),
        color='tab:orange', s=80, label='Training data',
        zorder=5, edgecolors='black', linewidth=0.5)

# Plot collocation points
t_collocation = collocation_dataset.t.detach()
ax1.scatter(t_collocation.numpy(),
        final_model(t_collocation).detach().numpy(),
        color='green', marker='x', s=50,
        label='Collocation points', alpha=0.7, zorder=4)

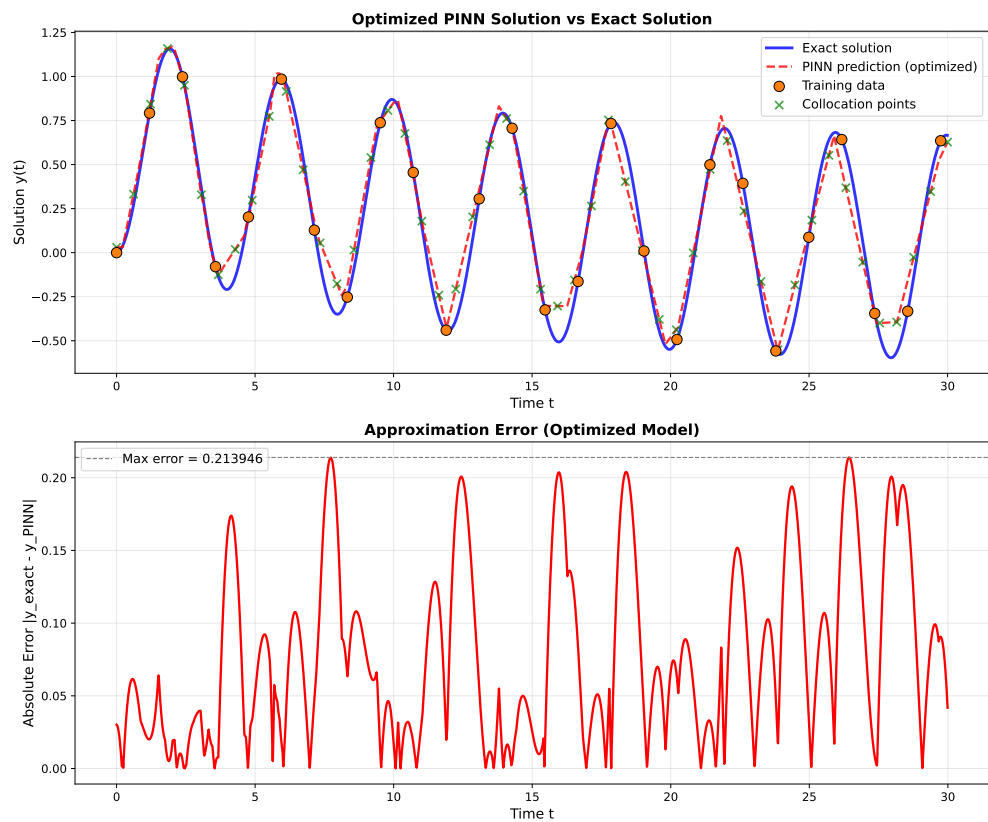
ax1.set_xlabel('Time t', fontsize=12)
ax1.set_ylabel('Solution y(t)', fontsize=12)
ax1.set_title('Optimized PINN Solution vs Exact Solution', fontsize=13, fontweight='bold')
ax1.legend(fontsize=11, loc='best')
ax1.grid(True, alpha=0.3)

# Plot 2: Error
ax2 = axes[1]
error = torch.abs(y_pred - y_exact)
```

45. Visualize Final Solution

```
ax2.plot(x_exact.numpy(), error.numpy(), 'r-', linewidth=2)
ax2.axhline(y=max_error, color='gray', linestyle='--', linewidth=1,
            label=f'Max error = {max_error:.6f}')
ax2.set_xlabel('Time t', fontsize=12)
ax2.set_ylabel('Absolute Error |y_exact - y_PINN|', fontsize=12)
ax2.set_title('Approximation Error (Optimized Model)', fontsize=13, fontweight='bold')
ax2.legend(fontsize=11)
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```



46. Comparison with Baseline

Let's compare the optimized configuration with a baseline:

```
print("Training baseline model for comparison...")

# Baseline configuration (from basic PINN demo)
baseline_config = {
    'l1': 32,
    'num_layers': 3,
    'activation': 'Tanh',
    'optimizer': 'Adam',
    'lr_unified': 3.0,
    'alpha': 0.06
}

print(f"\nBaseline Configuration:")
for key, val in baseline_config.items():
    print(f"  {key}: {val}")

# Train baseline
baseline_error = train_pinn(
    l1=baseline_config['l1'],
    num_layers=baseline_config['num_layers'],
    activation=baseline_config['activation'],
    optimizer_name=baseline_config['optimizer'],
    lr_unified=baseline_config['lr_unified'],
    alpha=baseline_config['alpha'],
    n_epochs=N_EPOCHS,
    verbose=False
)

print(f"\nValidation MSE Comparison:")
print("-" * 50)
print(f"  Baseline: {baseline_error:.6f}")
print(f"  Optimized: {best_val_error:.6f}")
print(f"  Improvement: {(1 - best_val_error/baseline_error)*100:.1f}%")
```

46. Comparison with Baseline

```
# Bar plot comparison
fig, ax = plt.subplots(figsize=(8, 6))
configs = ['Baseline', 'Optimized']
errors = [baseline_error, best_val_error]
colors = ['tab:blue', 'tab:green']

bars = ax.bar(configs, errors, color=colors, alpha=0.7, edgecolor='black', linewidth=1)

# Add value labels on bars
for bar, error in zip(bars, errors):
    height = bar.get_height()
    ax.text(bar.get_x() + bar.get_width()/2., height,
            f'{error:.6f}',
            ha='center', va='bottom', fontsize=11, fontweight='bold')

ax.set_ylabel('Validation MSE', fontsize=12)
ax.set_title('Model Comparison: Baseline vs Optimized', fontsize=13, fontweight='bold')
ax.grid(True, alpha=0.3, axis='y')
plt.tight_layout()
plt.show()
```

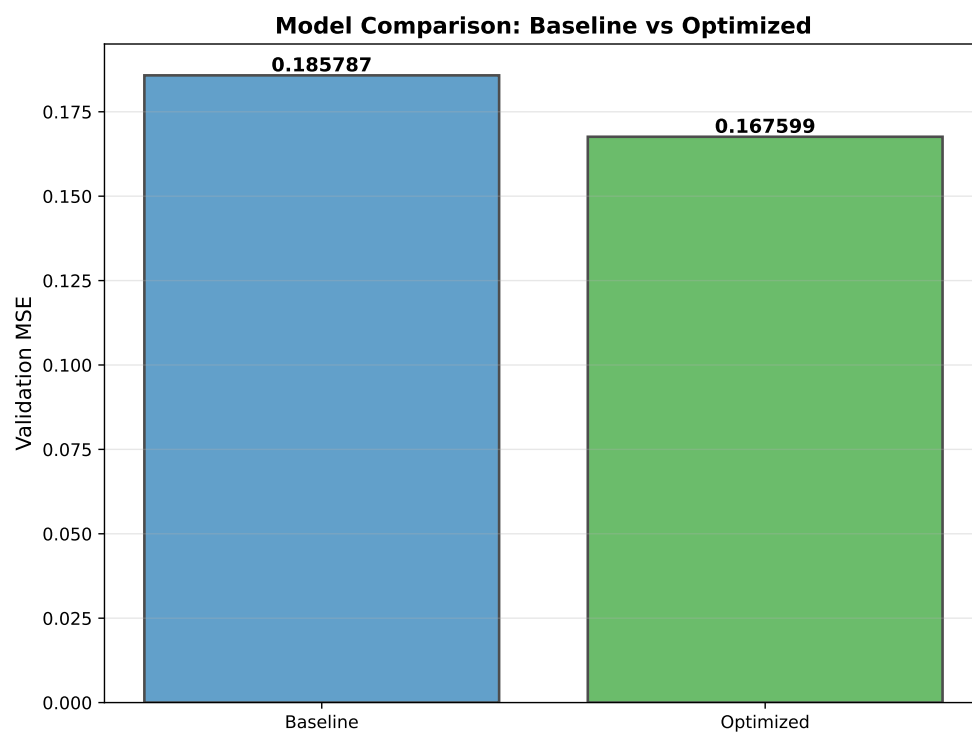
Training baseline model for comparison...

Baseline Configuration:

```
l1: 32
num_layers: 3
activation: Tanh
optimizer: Adam
lr_unified: 3.0
alpha: 0.06
```

Validation MSE Comparison:

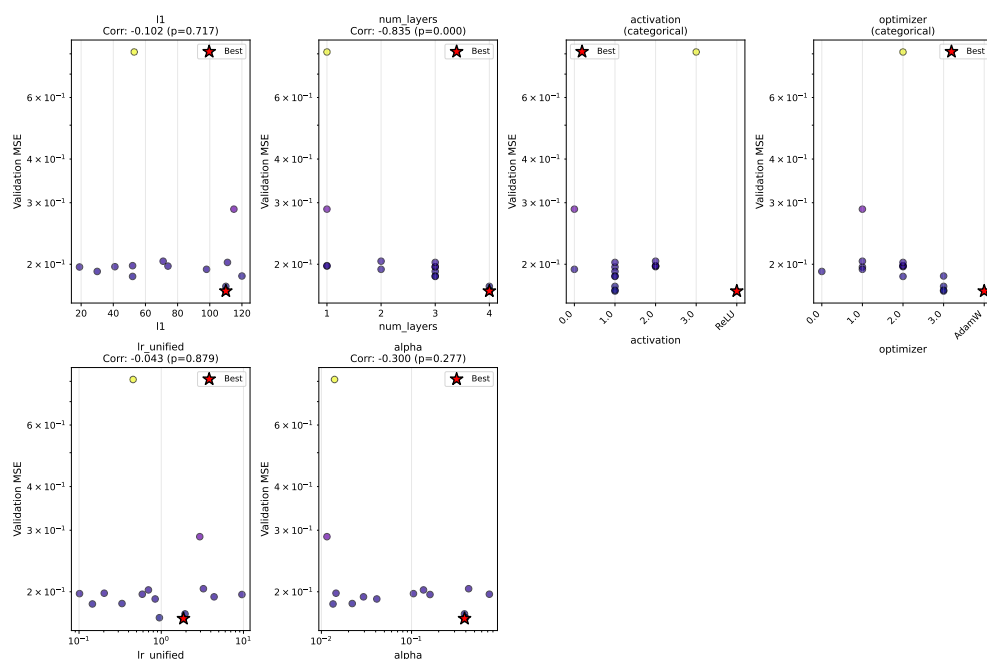
```
-----
Baseline: 0.185787
Optimized: 0.167599
Improvement: 9.8%
```



47. Hyperparameter Sensitivity Analysis

Let's analyze how sensitive the model is to each hyperparameter using the enhanced `plot_parameter_scatter()` method with Spearman correlation:

```
# Use the enhanced plot_parameter_scatter method with correlation display
optimizer.plot_parameter_scatter(
    result,
    ylabel="Validation MSE",
    cmap="plasma",
    figsize=(15, 10),
    show_correlation=True,
    log_y=True
)
```



47.1. Sensitivity Analysis (Spearman Correlation)

```
# Use the new sensitivity_spearman() method for tabular output
optimizer.sensitivity_spearman()
```

Sensitivity Analysis (Spearman Correlation):

```
-----
l1                : -0.102 (p=0.717)
num_layers        : -0.835 (p=0.000) ***
activation        : (categorical variable, use visual inspection)
optimizer         : (categorical variable, use visual inspection)
lr_unified        : -0.043 (p=0.879)
alpha            : -0.300 (p=0.277)
```

48. Summary

48.1. Key Findings

```
# Get optimization history for statistics
history = optimizer.y_

print("\n" + "="*70)
print("HYPERPARAMETER OPTIMIZATION SUMMARY")
print("="*70)

print("\n1. BEST CONFIGURATION FOUND:")
print(f"  - Neurons per layer (l1): {best_l1}")
print(f"  - Number of hidden layers: {best_num_layers}")
print(f"  - Activation function: {best_activation}")
print(f"  - Optimizer: {best_optimizer}")
print(f"  - Learning rate: {best_lr_unified:.4f}")
print(f"  - Physics weight (alpha): {best_alpha:.4f}")

print("\n2. PERFORMANCE:")
print(f"  - Validation MSE: {best_val_error:.6f}")
print(f"  - Full domain MSE: {full_mse:.6f}")
print(f"  - Maximum absolute error: {max_error:.6f}")

print("\n3. OPTIMIZATION STATISTICS:")
print(f"  - Total evaluations: {result.nfev}")
print(f"  - Initial best: {history[0]:.6f}")
print(f"  - Final best: {best_val_error:.6f}")
print(f"  - Improvement: {(1 - best_val_error/history[0])*100:.1f}%")

print("\n4. COMPARISON TO BASELINE:")
print(f"  - Baseline MSE: {baseline_error:.6f}")
print(f"  - Optimized MSE: {best_val_error:.6f}")
print(f"  - Improvement: {(1 - best_val_error/baseline_error)*100:.1f}%")

print("\n" + "="*70)
```

48. Summary

=====

HYPERPARAMETER OPTIMIZATION SUMMARY

=====

1. BEST CONFIGURATION FOUND:
 - Neurons per layer (l1): 110
 - Number of hidden layers: 4
 - Activation function: ReLU
 - Optimizer: AdamW
 - Learning rate: 1.8630
 - Physics weight (alpha): 0.3873
2. PERFORMANCE:
 - Validation MSE: 0.167599
 - Full domain MSE: 0.009609
 - Maximum absolute error: 0.213946
3. OPTIMIZATION STATISTICS:
 - Total evaluations: 15
 - Initial best: 0.196427
 - Final best: 0.167599
 - Improvement: 14.7%
4. COMPARISON TO BASELINE:
 - Baseline MSE: 0.185787
 - Optimized MSE: 0.167599
 - Improvement: 9.8%

=====

48.2. Recommendations

Based on the hyperparameter optimization results:

1. Network Architecture:

- The optimal architecture was found with {best_l1} neurons and {best_num_layers} hidden layers
- Best activation function: {best_activation}
- This balances model capacity with training efficiency

2. Optimizer Selection:

- Best optimizer: {best_optimizer}
- Different optimizers have different convergence characteristics for PINNs

3. Learning Rate:

- Optimal unified learning rate: {best_lr_unified:.4f}
- This translates to an actual Adam learning rate of {best_lr_unified * 0.001:.6f}

4. Physics Loss Weight:

- Optimal alpha: {best_alpha:.4f}
- This balances data fitting with physics constraint satisfaction

5. Training Strategy:

- Start with a broad search space to explore different architectures
- Use `var_trans` with “log10” for learning rate and physics weight parameters
- This enables efficient exploration of log-scale parameters without manual transformations
- Validate on held-out data to prevent overfitting to training points

6. Benefits of `var_trans` and Factor Variables:

- **Factor variables:** Categorical choices (activation, optimizer) handled automatically
- `SpotOptim` maps strings to integers internally and back to strings in results
- **Cleaner code:** No manual `10**x` conversions in objective function
- **Fewer errors:** Eliminates confusion about which scale values are in
- **Better optimization:** Searches efficiently in transformed space
- **Easier interpretation:** All results displayed in original scale

48.3. Using These Results

To use the optimized configuration in your own PINN problems:

```
# Create optimized PINN
model = LinearRegressor(
    input_dim=1,
    output_dim=1,
    l1={best_l1},
    num_hidden_layers={best_num_layers},
    activation="{best_activation}",
    lr={best_lr_unified:.4f}
)
```

```
optimizer = model.get_optimizer("{best_optimizer}")

# Use alpha={best_alpha:.4f} for physics loss weight
loss = data_loss + {best_alpha:.4f} * physics_loss
```

48.4. Using var_trans for Your Hyperparameter Optimization

When setting up optimization for your own PINN problems:

```
from spotoptim import SpotOptim

# Define search space with factor variables and log-scale parameters
bounds = [
    (16, 128),                # neurons (integer)
    (1, 4),                  # layers (integer)
    ("Tanh", "ReLU", "Sigmoid", "GELU"), # activation (factor)
    ("Adam", "SGD", "RMSprop", "AdamW"), # optimizer (factor)
    (0.1, 10.0),             # learning rate (log-scale)
    (0.01, 1.0)              # physics weight (log-scale)
]

var_type = ["int", "int", "factor", "factor", "float", "float"]
var_trans = [None, None, None, None, "log10", "log10"]

opt = SpotOptim(
    fun=your_objective_function,
    bounds=bounds,
    var_type=var_type,
    var_trans=var_trans, # Automatic log-scale and factor handling!
    max_iter=MAX_ITER,
    n_initial=N_INITIAL
)

result = opt.optimize()
```

Your objective function receives parameters in **original scale** - no manual transformations needed!

48.5. Future Directions

Consider exploring:

1. **Adaptive physics weights** that change during training
2. **Architecture search** including skip connections or residual blocks
3. **Batch size optimization** as an additional hyperparameter
4. **Multi-objective optimization** balancing accuracy and computational cost
5. **Transfer learning** from pre-optimized configurations
6. **Learning rate schedules** with different decay strategies

Note: The specific optimal values depend on the problem, data distribution, and computational budget. Always validate results on held-out test data.

48.6. Jupyter Notebook

Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

49. Learning Rate Mapping for Unified Optimizer Interface

SpotOptim provides a sophisticated learning rate mapping system through the `map_lr()` function, enabling a unified interface for learning rates across different PyTorch optimizers. This solves the challenge that different optimizers operate on vastly different learning rate scales.

49.1. Overview

Different PyTorch optimizers use different default learning rates and optimal ranges:

- **Adam**: default 0.001, typical range 0.0001-0.01
- **SGD**: default 0.01, typical range 0.001-0.1
- **RMSprop**: default 0.01, typical range 0.001-0.1

This makes it difficult to compare optimizer performance fairly or optimize learning rates across different optimizers. The `map_lr()` function provides a unified scale where **`lr_unified=1.0` corresponds to each optimizer's PyTorch default.**

Module: `spotoptim.utils.mapping`

Key Features:

- Unified learning rate scale across all optimizers
- Fair comparison when evaluating different optimizers
- Simplified hyperparameter optimization
- Based on official PyTorch default learning rates
- Supports 13 major PyTorch optimizers

49.2. Quick Start

49.2.1. Basic Usage

49. Learning Rate Mapping for Unified Optimizer Interface

```
from spotoptim.utils.mapping import map_lr

# Get optimizer-specific learning rate from unified scale
lr_adam = map_lr(1.0, "Adam")      # Returns 0.001 (Adam's default)
lr_sgd = map_lr(1.0, "SGD")        # Returns 0.01 (SGD's default)
lr_rmsprop = map_lr(1.0, "RMSprop") # Returns 0.01 (RMSprop's default)

print(f"Unified lr=1.0:")
print(f"  Adam:    {lr_adam}")
print(f"  SGD:      {lr_sgd}")
print(f"  RMSprop: {lr_rmsprop}")
```

```
Unified lr=1.0:
  Adam:    0.001
  SGD:      0.01
  RMSprop: 0.01
```

49.2.2. Scaling Learning Rates

```
from spotoptim.utils.mapping import map_lr

# Scale all learning rates by the same factor
unified_lr = 0.5

lr_adam = map_lr(unified_lr, "Adam")      # 0.5 * 0.001 = 0.0005
lr_sgd = map_lr(unified_lr, "SGD")        # 0.5 * 0.01 = 0.005
lr_rmsprop = map_lr(unified_lr, "RMSprop") # 0.5 * 0.01 = 0.005

print(f"Unified lr={unified_lr}:")
print(f"  Adam:    {lr_adam}")
print(f"  SGD:      {lr_sgd}")
print(f"  RMSprop: {lr_rmsprop}")
```

```
Unified lr=0.5:
  Adam:    0.0005
  SGD:      0.005
  RMSprop: 0.005
```

49.2.3. Integration with LinearRegressor

```
from spotoptim.nn.linear_regressor import LinearRegressor

# Create model with unified learning rate
model = LinearRegressor(
    input_dim=10,
    output_dim=1,
    l1=32,
    num_hidden_layers=2,
    lr=1.0 # Unified learning rate
)

# Get optimizer - automatically uses mapped learning rate
optimizer_adam = model.get_optimizer("Adam")      # Gets 1.0 * 0.001 = 0.001
optimizer_sgd = model.get_optimizer("SGD")        # Gets 1.0 * 0.01 = 0.01

# Verify the actual learning rates
print(f"Adam actual lr: {optimizer_adam.param_groups[0]['lr']}")
print(f"SGD actual lr: {optimizer_sgd.param_groups[0]['lr']}")
```

Adam actual lr: 0.001

SGD actual lr: 0.01

49.3. Function Reference

49.3.1. `map_lr(lr_unified, optimizer_name, use_default_scale=True)`

Maps a unified learning rate to an optimizer-specific learning rate.

Parameters:

- `lr_unified` (float): Unified learning rate multiplier. A value of 1.0 corresponds to the optimizer's default learning rate. Typical range: [0.001, 100.0].
- `optimizer_name` (str): Name of the PyTorch optimizer. Must be one of: "Adadelata", "Adagrad", "Adam", "AdamW", "SparseAdam", "Adamax", "ASGD", "LBFGS", "NAdam", "RAdam", "RMSprop", "Rprop", "SGD".
- `use_default_scale` (bool, optional): Whether to scale by the optimizer's default learning rate. If `True` (default), `lr_unified` is multiplied by the default lr. If `False`, returns `lr_unified` directly.

Returns:

- **float:** The optimizer-specific learning rate.

Raises:

- **ValueError:** If `optimizer_name` is not supported.
- **ValueError:** If `lr_unified` is not positive.

Example:

```
from spotoptim.utils.mapping import map_lr

# Get default learning rates (unified lr = 1.0)
lr = map_lr(1.0, "Adam")      # 0.001
lr = map_lr(1.0, "SGD")      # 0.01
lr = map_lr(1.0, "RMSprop")  # 0.01

# Scale learning rates
lr = map_lr(0.5, "Adam")     # 0.0005
lr = map_lr(2.0, "SGD")     # 0.02

# Without default scaling
lr = map_lr(0.01, "Adam", use_default_scale=False) # 0.01 (direct)
```

49.4. Supported Optimizers

All major PyTorch optimizers are supported with their default learning rates:

Optimizer	Default LR	Typical Range	Notes
Adam	0.001	0.0001-0.01	Most popular, good default
AdamW	0.001	0.0001-0.01	Adam with weight decay
Adamax	0.002	0.0001-0.01	Adam variant with infinity norm
NAdam	0.002	0.0001-0.01	Adam with Nesterov momentum
RAdam	0.001	0.0001-0.01	Rectified Adam
SparseAdam	0.001	0.0001-0.01	For sparse gradients
SGD	0.01	0.001-0.1	Classic, needs momentum
RMSprop	0.01	0.001-0.1	Good for RNNs
Adagrad	0.01	0.001-0.1	Adaptive learning rate
Adadelat	1.0	0.1-10.0	Extension of Adagrad
ASGD	0.01	0.001-0.1	Averaged SGD
LBFGS	1.0	0.1-10.0	Second-order optimizer
Rprop	0.01	0.001-0.1	Resilient backpropagation

49.5. Use Cases

49.5.1. Comparing Different Optimizers

```

import torch
import torch.nn as nn
from spotoptim.nn.linear_regressor import LinearRegressor
from spotoptim.data import get_diabetes_dataloaders

# Load data
train_loader, test_loader, _ = get_diabetes_dataloaders(batch_size=32, random_state=42)

# Test different optimizers with unified learning rate
unified_lr = 1.0
optimizers_to_test = ["Adam", "SGD", "RMSprop", "AdamW"]
results = {}

for opt_name in optimizers_to_test:
    # Reset for fair comparison
    torch.manual_seed(42)
    model = LinearRegressor(input_dim=10, output_dim=1, l1=32,
                           num_hidden_layers=2, lr=unified_lr)

    # Create optimizer with mapped learning rate
    if opt_name == "SGD":
        optimizer = model.get_optimizer(opt_name, momentum=0.9)
    else:
        optimizer = model.get_optimizer(opt_name)

    criterion = nn.MSELoss()

    # Train
    model.train()
    for epoch in range(50):
        for batch_X, batch_y in train_loader:
            optimizer.zero_grad()
            predictions = model(batch_X)
            loss = criterion(predictions, batch_y)
            loss.backward()
            optimizer.step()

    # Evaluate
    model.eval()

```

```

test_loss = 0.0
with torch.no_grad():
    for batch_X, batch_y in test_loader:
        predictions = model(batch_X)
        test_loss += criterion(predictions, batch_y).item()

avg_test_loss = test_loss / len(test_loader)
results[opt_name] = avg_test_loss

print(f"{opt_name:10s}: Test MSE = {avg_test_loss:.4f} "
      f"(actual lr = {optimizer.param_groups[0]['lr']:.6f})")

# Find best optimizer
best_opt = min(results, key=results.get)
print(f"\nBest optimizer: {best_opt} with MSE = {results[best_opt]:.4f}")

```

```

Adam      : Test MSE = 3859.4478 (actual lr = 0.001000)
SGD       : Test MSE = 5271.6840 (actual lr = 0.010000)
RMSprop   : Test MSE = 2769.5164 (actual lr = 0.010000)
AdamW     : Test MSE = 3866.1197 (actual lr = 0.001000)

```

Best optimizer: RMSprop with MSE = 2769.5164

49.5.2. Hyperparameter Optimization

```

from spotoptim import SpotOptim
from spotoptim.nn.linear_regressor import LinearRegressor
from spotoptim.data import get_diabetes_data loaders
import torch
import torch.nn as nn
import numpy as np

def train_model(X):
    """Objective function for hyperparameter optimization."""
    results = []

    # Load data once
    train_loader, test_loader, _ = get_diabetes_data loaders(batch_size=32, random_state=42)

    for params in X:
        lr_unified = 10 ** params[0] # Log scale: [-4, 0]
        optimizer_name = params[1]   # Factor: "Adam", "SGD", "RMSprop"

```

```

# Create model with unified lr - automatically scaled per optimizer
torch.manual_seed(42)
model = LinearRegressor(input_dim=10, output_dim=1, l1=32, num_hidden_layers=2, lr=lr_unified)
optimizer = model.get_optimizer(optimizer_name)

criterion = nn.MSELoss()

# Train
model.train()
for epoch in range(30):
    for batch_X, batch_y in train_loader:
        optimizer.zero_grad()
        predictions = model(batch_X)
        loss = criterion(predictions, batch_y)
        loss.backward()
        optimizer.step()

# Evaluate
model.eval()
test_loss = 0.0
with torch.no_grad():
    for batch_X, batch_y in test_loader:
        predictions = model(batch_X)
        test_loss += criterion(predictions, batch_y).item()

avg_test_loss = test_loss / len(test_loader)
results.append(avg_test_loss)

return np.array(results)

# Optimize unified lr across different optimizers
spot_optimizer = SpotOptim(
    fun=train_model,
    bounds=[(-4, 0), ("Adam", "SGD", "RMSprop")],
    var_type=["float", "factor"],
    max_iter=10, # Small for demo
    n_initial=5,
    seed=42
)
result = spot_optimizer.optimize()

print(f"\nBest unified lr: {10**result.x[0]:.6f}")
print(f"Best optimizer: {result.x[1]}")
print(f"Best test MSE: {result.fun:.4f}")

```

49. Learning Rate Mapping for Unified Optimizer Interface

```
# Show actual learning rate used
from spotoptim.utils.mapping import map_lr
actual_lr = map_lr(10**result.x[0], result.x[1])
print(f"Actual {result.x[1]} learning rate: {actual_lr:.6f}")
```

```
Best unified lr: 0.003144
Best optimizer: SGD
Best test MSE: 4135.5200
Actual SGD learning rate: 0.000031
```

49.5.3. Hyperparameter Optimization with SpotOptim

Note, N_INITIAL and MAX_ITER are kept small for demonstration; increase for real use.

```
from spotoptim import SpotOptim
from spotoptim.nn.linear_regressor import LinearRegressor
from spotoptim.data import get_diabetes_data loaders
import torch.nn as nn
import torch
import numpy as np

MAX_ITER = 10
N_INITIAL = 5

def train_and_evaluate(X):
    """Objective function for hyperparameter optimization."""
    results = []

    # Load data once
    train_loader, test_loader, _ = get_diabetes_data loaders(
        batch_size=32, random_state=42
    )

    for params in X:
        # Extract hyperparameters
        lr_unified = 10 ** params[0] # Log scale
        optimizer_name = params[1] # Factor variable
        l1 = int(params[2]) # Integer
        num_layers = int(params[3]) # Integer
```



```

# Create model with unified learning rate
model = LinearRegressor(
    input_dim=10,
    output_dim=1,
    l1=l1,
    num_hidden_layers=num_layers,
    lr=lr_unified # Automatically mapped per optimizer
)

# Get optimizer (lr already mapped internally)
if optimizer_name == "SGD":
    optimizer = model.get_optimizer(optimizer_name, momentum=0.9)
else:
    optimizer = model.get_optimizer(optimizer_name)

criterion = nn.MSELoss()

# Train
model.train()
for epoch in range(30):
    for batch_X, batch_y in train_loader:
        optimizer.zero_grad()
        predictions = model(batch_X)
        loss = criterion(predictions, batch_y)
        loss.backward()
        optimizer.step()

# Evaluate
model.eval()
test_loss = 0.0
with torch.no_grad():
    for batch_X, batch_y in test_loader:
        predictions = model(batch_X)
        test_loss += criterion(predictions, batch_y).item()

avg_test_loss = test_loss / len(test_loader)
results.append(avg_test_loss)

return np.array(results)

# Optimize learning rate, optimizer choice, and architecture
optimizer = SpotOptim(
    fun=train_and_evaluate,
    bounds=[

```

49. Learning Rate Mapping for Unified Optimizer Interface

```
        (-4, 0), # log10(lr_unified): [0.0001, 1.0]
        ("Adam", "SGD", "RMSprop", "AdamW"), # Optimizer choice
        (16, 128), # Layer size
        (1, 3) # Number of hidden layers
    ],
    var_type=["float", "factor", "int", "int"],
    max_iter=MAX_ITER,
    n_initial=N_INITIAL,
    seed=42
)

result = optimizer.optimize()

# Display results
print("\nOptimization Results:")
print(f"Best unified lr: {10**result.x[0]:.6f}")
print(f"Best optimizer: {result.x[1]}")
print(f"Best layer size: {int(result.x[2])}")
print(f"Best num layers: {int(result.x[3])}")
print(f"Best test MSE: {result.fun:.4f}")

# Show actual learning rate used
from spotoptim.utils.mapping import map_lr
actual_lr = map_lr(10**result.x[0], result.x[1])
print(f"Actual {result.x[1]} learning rate: {actual_lr:.6f}")
```

```
Optimization Results:
Best unified lr: 0.305427
Best optimizer: RMSprop
Best layer size: 118
Best num layers: 2
Best test MSE: 2874.4881
Actual RMSprop learning rate: 0.003054
```

49.5.4. Log-Scale Hyperparameter Search

```
from spotoptim.utils.mapping import map_lr
import numpy as np

# Common pattern: sample unified lr from log scale
log_lr_range = np.linspace(-4, 0, 10) # [-4, -3.56, ..., 0]
```

```

optimizers = ["Adam", "SGD", "RMSprop"]

print("Log-scale learning rate search:")
print()
print(f"{'log_lr':<10} {'unified_lr':<12} {'Adam':<12} {'SGD':<12} {'RMSprop':<12}")
print("-" * 60)

for log_lr in log_lr_range:
    lr_unified = 10 ** log_lr
    lr_adam = map_lr(lr_unified, "Adam")
    lr_sgd = map_lr(lr_unified, "SGD")
    lr_rmsprop = map_lr(lr_unified, "RMSprop")

    print(f"{'log_lr':<10.2f} {'lr_unified':<12.6f} {'lr_adam':<12.8f} "
          f"{'lr_sgd':<12.8f} {'lr_rmsprop':<12.8f}")

```

Log-scale learning rate search:

log_lr	unified_lr	Adam	SGD	RMSprop
-4.00	0.000100	0.00000010	0.00000100	0.00000100
-3.56	0.000278	0.00000028	0.00000278	0.00000278
-3.11	0.000774	0.00000077	0.00000774	0.00000774
-2.67	0.002154	0.00000215	0.00002154	0.00002154
-2.22	0.005995	0.00000599	0.00005995	0.00005995
-1.78	0.016681	0.00001668	0.00016681	0.00016681
-1.33	0.046416	0.00004642	0.00046416	0.00046416
-0.89	0.129155	0.00012915	0.00129155	0.00129155
-0.44	0.359381	0.00035938	0.00359381	0.00359381
0.00	1.000000	0.00100000	0.01000000	0.01000000

49.5.5. Custom Learning Rate Schedules

```

import torch
import torch.nn as nn
from spotoptim.nn.linear_regressor import LinearRegressor
from spotoptim.utils.mapping import map_lr

# Create model with unified lr
model = LinearRegressor(input_dim=10, output_dim=1, lr=1.0)

# Get initial optimizer

```

49. Learning Rate Mapping for Unified Optimizer Interface

```
optimizer = model.get_optimizer("Adam")
initial_lr = optimizer.param_groups[0]['lr']
print(f"Initial learning rate: {initial_lr}")

# Use PyTorch learning rate scheduler
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=30, gamma=0.1)

# Training with scheduler
for epoch in range(100):
    # ... training code ...
    scheduler.step()

    if (epoch + 1) % 30 == 0:
        current_lr = optimizer.param_groups[0]['lr']
        print(f"Epoch {epoch+1}: lr = {current_lr:.8f}")
```

```
Initial learning rate: 0.001
Epoch 30: lr = 0.00010000
Epoch 60: lr = 0.00001000
Epoch 90: lr = 0.00000100
```

49.5.6. Direct Usage Without LinearRegressor

```
import torch
import torch.nn as nn
from spotoptim.utils.mapping import map_lr

# Define your own model
class MyModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(10, 32)
        self.fc2 = nn.Linear(32, 1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        return self.fc2(x)

model = MyModel()

# Use map_lr to get optimizer-specific learning rate
unified_lr = 2.0
```

```
optimizer_name = "Adam"

actual_lr = map_lr(unified_lr, optimizer_name)
optimizer = torch.optim.Adam(model.parameters(), lr=actual_lr)

print(f"Unified lr: {unified_lr}")
print(f"Actual {optimizer_name} lr: {actual_lr}")
```

```
Unified lr: 2.0
Actual Adam lr: 0.002
```

49.6. Best Practices

49.6.1. Choosing Unified Learning Rate

For initial experiments:

- Start with `lr=1.0` (gives defaults for all optimizers)
- Test with `lr=0.1`, `lr=1.0`, `lr=10.0` to get a sense of scale

For hyperparameter optimization:

- Use log scale: sample from `[-4, 0]` or `[-3, 1]`
- Convert with `lr_unified = 10 ** log_lr`
- This gives reasonable ranges for all optimizers

For fine-tuning:

- If training is unstable: try smaller `lr` (e.g., 0.1 or 0.5)
- If training is too slow: try larger `lr` (e.g., 2.0 or 5.0)
- Monitor loss curves to adjust

49.6.2. Optimizer Selection Guidelines

Adam family (Adam, AdamW, NAdam, RAdam):

- Good default choice for most tasks
- Adaptive learning rates per parameter
- Works well out of the box
- Use `lr=1.0` as starting point

SGD:

- Good for large datasets

49. Learning Rate Mapping for Unified Optimizer Interface

- Often achieves better generalization
- Requires momentum (e.g., 0.9)
- Use `lr=1.0` with `momentum=0.9`

RMSprop:

- Good for recurrent networks
- Handles non-stationary objectives
- Use `lr=1.0` as starting point

Others (Adadelata, Adagrad, etc.):

- Specialized use cases
- Start with `lr=1.0` and adjust

49.6.3. Common Patterns

```
# Pattern 1: Quick optimizer comparison
model = LinearRegressor(input_dim=10, output_dim=1, lr=1.0)
for opt in ["Adam", "SGD", "RMSprop"]:
    optimizer = model.get_optimizer(opt)
    # ... train and compare ...

# Pattern 2: Hyperparameter optimization
def objective(X):
    lr_unified = 10 ** X[:, 0] # Log scale
    optimizer_name = X[:, 1]   # Factor
    # ... use unified lr ...

# Pattern 3: Override model's lr
model = LinearRegressor(input_dim=10, output_dim=1, lr=1.0)
optimizer = model.get_optimizer("Adam", lr=2.0) # Override with 2.0

# Pattern 4: Direct mapping
from spotoptim.utils.mapping import map_lr
lr_actual = map_lr(unified_lr, optimizer_name)
optimizer = torch.optim.Adam(params, lr=lr_actual)
```

49.7. Troubleshooting

49.7.1. Issue: Training is unstable (loss explodes)

Solution: Learning rate is too high. Try:

```
model = LinearRegressor(input_dim=10, output_dim=1, lr=0.1) # Reduce from 1.0
```

49.7.2. Issue: Training is too slow (loss decreases very slowly)

Solution: Learning rate is too low. Try:

```
model = LinearRegressor(input_dim=10, output_dim=1, lr=5.0) # Increase from 1.0
```

49.7.3. Issue: Different results across optimizer runs

Solution: Set random seed for reproducibility:

```
import torch
torch.manual_seed(42)
```

49.7.4. Issue: Want to use raw learning rate without mapping

Solution: Use `use_default_scale=False`:

```
from spotoptim.utils.mapping import map_lr
lr = map_lr(0.001, "Adam", use_default_scale=False) # Returns 0.001 directly
```

49.7.5. Issue: Optimizer not supported

Solution: Check supported optimizers:

```
from spotoptim.utils.mapping import OPTIMIZER_DEFAULT_LR
print("Supported optimizers:", list(OPTIMIZER_DEFAULT_LR.keys()))
```

49.8. Technical Details

49.8.1. How It Works

The mapping is simple but effective:

```
actual_lr = lr_unified * default_lr[optimizer_name]
```

49. Learning Rate Mapping for Unified Optimizer Interface

For example:

- `map_lr(1.0, "Adam")` $\rightarrow 1.0 * 0.001 = 0.001$
- `map_lr(0.5, "SGD")` $\rightarrow 0.5 * 0.01 = 0.005$
- `map_lr(2.0, "RMSprop")` $\rightarrow 2.0 * 0.01 = 0.02$

This ensures that the same unified learning rate gives optimizer-specific learning rates in their typical working ranges.

49.8.2. Design Rationale

Why use defaults as scaling factors?

PyTorch's default learning rates are carefully chosen to work well for typical use cases. By using them as scaling factors:

1. `lr=1.0` always gives sensible defaults
2. Scaling preserves the relative relationships between optimizers
3. Each optimizer stays in its optimal range
4. Easy to understand and explain

Comparison with spotPython's approach:

spotPython uses `lr = lr_mult * default_lr` in `optimizer_handler()`. Our implementation:

- Separates mapping logic (testable, reusable)
- Provides standalone function (`map_lr()`)
- Comprehensive error handling and validation
- Extensive documentation and examples
- Full integration with `LinearRegressor`

49.8.3. Default Learning Rates

All values verified against PyTorch documentation:

```
OPTIMIZER_DEFAULT_LR = {
    "Adadelta": 1.0,
    "Adagrad": 0.01,
    "Adam": 0.001,
    "AdamW": 0.001,
    "SparseAdam": 0.001,
    "Adamax": 0.002,
    "ASGD": 0.01,
    "LBFGS": 1.0,
```



```

    "NAdam": 0.002,
    "RAdam": 0.001,
    "RMSprop": 0.01,
    "Rprop": 0.01,
    "SGD": 0.01,
}

```

49.9. Examples

49.9.1. Complete Example: Optimizer Comparison Study

```

"""
Complete example: Compare optimizers with unified learning rate interface.
"""

import torch
import torch.nn as nn
from spotoptim.nn.linear_regressor import LinearRegressor
from spotoptim.data import get_diabetes_dataloaders
from spotoptim.utils.mapping import map_lr
import matplotlib.pyplot as plt

# Set seed for reproducibility
torch.manual_seed(42)

# Load data
train_loader, test_loader, _ = get_diabetes_dataloaders(
    batch_size=32,
    random_state=42
)

# Test configurations
optimizers = ["Adam", "SGD", "RMSprop", "AdamW"]
unified_lrs = [0.5, 1.0, 2.0]

# Store results
results = {}

print("Training models with different optimizers and learning rates...")
print()

for unified_lr in unified_lrs:

```

```

results[unified_lr] = {}

for opt_name in optimizers:
    # Reset model for fair comparison
    torch.manual_seed(42)

    # Create model with unified lr
    model = LinearRegressor(
        input_dim=10,
        output_dim=1,
        l1=32,
        num_hidden_layers=2,
        lr=unified_lr
    )

    # Get optimizer
    if opt_name == "SGD":
        optimizer = model.get_optimizer(opt_name, momentum=0.9)
    else:
        optimizer = model.get_optimizer(opt_name)

    actual_lr = optimizer.param_groups[0]['lr']
    criterion = nn.MSELoss()

    # Track training loss
    train_losses = []

    # Train
    model.train()
    for epoch in range(50):
        epoch_loss = 0.0
        for batch_X, batch_y in train_loader:
            optimizer.zero_grad()
            predictions = model(batch_X)
            loss = criterion(predictions, batch_y)
            loss.backward()
            optimizer.step()
            epoch_loss += loss.item()

        avg_epoch_loss = epoch_loss / len(train_loader)
        train_losses.append(avg_epoch_loss)

    # Evaluate on test set
    model.eval()

```

```

test_loss = 0.0
with torch.no_grad():
    for batch_X, batch_y in test_loader:
        predictions = model(batch_X)
        test_loss += criterion(predictions, batch_y).item()

avg_test_loss = test_loss / len(test_loader)
results[unified_lr][opt_name] = {
    'train_losses': train_losses,
    'test_loss': avg_test_loss,
    'actual_lr': actual_lr
}

print(f"Unified lr={unified_lr:.1f}, {opt_name:10s}: "
      f"actual_lr={actual_lr:.6f}, test_MSE={avg_test_loss:.4f}")

# Display summary
print()
print("=" * 70)
print("Summary: Best configurations")
print("=" * 70)

for unified_lr in unified_lrs:
    best_opt = min(results[unified_lr].items(),
                    key=lambda x: x[1]['test_loss'])
    opt_name, metrics = best_opt

    print(f"Unified lr={unified_lr:.1f}: {opt_name:10s} "
          f"(test MSE={metrics['test_loss']:.4f}, "
          f"actual lr={metrics['actual_lr']:.6f})")

# Find overall best
best_overall = None
best_overall_loss = float('inf')

for unified_lr in unified_lrs:
    for opt_name, metrics in results[unified_lr].items():
        if metrics['test_loss'] < best_overall_loss:
            best_overall_loss = metrics['test_loss']
            best_overall = (unified_lr, opt_name, metrics['actual_lr'])

print()
print(f"Overall best: unified_lr={best_overall[0]:.1f}, "
      f"optimizer={best_overall[1]}, ")

```

49. Learning Rate Mapping for Unified Optimizer Interface

```
f"test_MSE={best_overall_loss:.4f}")
print(f"Actual learning rate used: {best_overall[2]:.6f}")
```

Training models with different optimizers and learning rates...

Unified lr=0.5, Adam	: actual_lr=0.000500, test_MSE=5095.8747
Unified lr=0.5, SGD	: actual_lr=0.005000, test_MSE=5291.3903
Unified lr=0.5, RMSprop	: actual_lr=0.005000, test_MSE=2828.7084
Unified lr=0.5, AdamW	: actual_lr=0.000500, test_MSE=5107.9592
Unified lr=1.0, Adam	: actual_lr=0.001000, test_MSE=3859.4478
Unified lr=1.0, SGD	: actual_lr=0.010000, test_MSE=5271.6840
Unified lr=1.0, RMSprop	: actual_lr=0.010000, test_MSE=2769.5164
Unified lr=1.0, AdamW	: actual_lr=0.001000, test_MSE=3866.1197
Unified lr=2.0, Adam	: actual_lr=0.002000, test_MSE=3160.6882
Unified lr=2.0, SGD	: actual_lr=0.020000, test_MSE=nan
Unified lr=2.0, RMSprop	: actual_lr=0.020000, test_MSE=2883.3345
Unified lr=2.0, AdamW	: actual_lr=0.002000, test_MSE=3162.8034

=====
Summary: Best configurations
=====

Unified lr=0.5: RMSprop	(test MSE=2828.7084, actual lr=0.005000)
Unified lr=1.0: RMSprop	(test MSE=2769.5164, actual lr=0.010000)
Unified lr=2.0: RMSprop	(test MSE=2883.3345, actual lr=0.020000)

Overall best: unified_lr=1.0, optimizer=RMSprop, test_MSE=2769.5164
Actual learning rate used: 0.010000

49.10. Jupyter Notebook

Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

Part IX.

Data Sets

50. Diabetes Dataset Utilities

SpotOptim provides convenient utilities for working with the sklearn diabetes dataset, including PyTorch `Dataset` and `DataLoader` implementations. These utilities simplify data loading, preprocessing, and model training for regression tasks.

50.1. Overview

The diabetes dataset contains 10 baseline variables (age, sex, body mass index, average blood pressure, and six blood serum measurements) for 442 diabetes patients. The target is a quantitative measure of disease progression one year after baseline.

Module: `spotoptim.data.diabetes`

Key Components:

- `DiabetesDataset`: PyTorch Dataset class
- `get_diabetes_dataloaders()`: Convenience function for complete data pipeline

50.2. Quick Start

50.2.1. Basic Usage

```
from spotoptim.data import get_diabetes_dataloaders
from sklearn.datasets import load_diabetes
from spotoptim.data.diabetes import DiabetesDataset
import numpy as np

# Load data
diabetes = load_diabetes()
X = diabetes.data
y = diabetes.target.reshape(-1, 1)

# Now create the dataset
dataset = DiabetesDataset(X, y, transform=None, target_transform=None)
```

```
# Load data with default settings
train_loader, test_loader, scaler = get_diabetes_dataloaders()

# Iterate through batches
for batch_X, batch_y in train_loader:
    print(f"Batch features: {batch_X.shape}") # (32, 10)
    print(f"Batch targets: {batch_y.shape}")  # (32, 1)
    break
```

```
Batch features: torch.Size([32, 10])
Batch targets: torch.Size([32, 1])
```

50.2.2. Training a Model

```
import torch
import torch.nn as nn
from spotoptim.data import get_diabetes_dataloaders
from spotoptim.nn.linear_regressor import LinearRegressor

# Load data
train_loader, test_loader, scaler = get_diabetes_dataloaders(
    test_size=0.2,
    batch_size=32,
    scale_features=True,
    random_state=42
)

# Create model
model = LinearRegressor(
    input_dim=10,
    output_dim=1,
    l1=64,
    num_hidden_layers=2,
    activation="ReLU"
)

# Setup training
criterion = nn.MSELoss()
optimizer = model.get_optimizer("Adam", lr=0.01)

# Training loop
num_epochs = 100
```



```

for epoch in range(num_epochs):
    model.train()
    train_loss = 0.0

    for batch_X, batch_y in train_loader:
        # Forward pass
        predictions = model(batch_X)
        loss = criterion(predictions, batch_y)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        train_loss += loss.item()

    avg_train_loss = train_loss / len(train_loader)

    if (epoch + 1) % 20 == 0:
        print(f"Epoch {epoch+1}/{num_epochs}: Loss = {avg_train_loss:.4f}")

# Evaluation
model.eval()
test_loss = 0.0

with torch.no_grad():
    for batch_X, batch_y in test_loader:
        predictions = model(batch_X)
        loss = criterion(predictions, batch_y)
        test_loss += loss.item()

avg_test_loss = test_loss / len(test_loader)
print(f"Test MSE: {avg_test_loss:.4f}")

```

```

Epoch 20/100: Loss = 27494.3093
Epoch 40/100: Loss = 29340.1888
Epoch 60/100: Loss = 27948.9766
Epoch 80/100: Loss = 27642.7507
Epoch 100/100: Loss = 27600.7547
Test MSE: 26517.6699

```

50.3. Function Reference

50.3.1. get_diabetes_dataloaders()

Loads the sklearn diabetes dataset and returns configured PyTorch DataLoaders.

Signature:

```
get_diabetes_dataloaders(
    test_size=0.2,
    batch_size=32,
    shuffle_train=True,
    shuffle_test=False,
    random_state=42,
    scale_features=True,
    num_workers=0,
    pin_memory=False
)
```

```
(<torch.utils.data.dataloader.DataLoader at 0x11d0950f0>,
 <torch.utils.data.dataloader.DataLoader at 0x11d12deb0>,
 StandardScaler())
```

Parameters:

Parameter	Type	Default	Description
test_size	float	0.2	Proportion of dataset for testing (0.0 to 1.0)
batch_size	int	32	Number of samples per batch
shuffle_train	bool	True	Whether to shuffle training data
shuffle_test	bool	False	Whether to shuffle test data
random_state	int	42	Random seed for train/test split
scale_features	bool	True	Whether to standardize features
num_workers	int	0	Number of subprocesses for data loading
pin_memory	bool	False	Whether to pin memory (useful for GPU)

Returns:

- `train_loader` (DataLoader): Training data loader
- `test_loader` (DataLoader): Test data loader
- `scaler` (StandardScaler or None): Fitted scaler if `scale_features=True`, else None

Example:

```
from spotoptim.data import get_diabetes_dataloaders

# Custom configuration
train_loader, test_loader, scaler = get_diabetes_dataloaders(
    test_size=0.3,
    batch_size=64,
    shuffle_train=True,
    scale_features=True,
    random_state=123
)

print(f"Training batches: {len(train_loader)}")
print(f"Test batches: {len(test_loader)}")
print(f"Scaler mean: {scaler.mean_[0:3]}") # First 3 features
```

```
Training batches: 5
Test batches: 3
Scaler mean: [-0.00056537  0.00132258  0.00027836]
```

50.4. DiabetesDataset Class

PyTorch Dataset implementation for the diabetes dataset.

Signature:

```
DiabetesDataset(X, y, transform=None, target_transform=None)
```

```
<spotoptim.data.diabetes.DiabetesDataset at 0x11d06bb50>
```

Parameters:

- `X` (np.ndarray): Feature matrix of shape (n_samples, n_features)
- `y` (np.ndarray): Target values of shape (n_samples,) or (n_samples, 1)
- `transform` (callable, optional): Transform to apply to features
- `target_transform` (callable, optional): Transform to apply to targets

Attributes:

- `X` (torch.Tensor): Feature tensor (`n_samples`, `n_features`)
- `y` (torch.Tensor): Target tensor (`n_samples`, 1)
- `n_features` (int): Number of features (10 for diabetes)
- `n_samples` (int): Number of samples

Methods:

- `__len__()`: Returns number of samples
- `__getitem__(idx)`: Returns tuple (features, target) for given index

50.4.1. Manual Dataset Creation

```
from spotoptim.data import DiabetesDataset
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from torch.utils.data import DataLoader

# Load raw data
diabetes = load_diabetes()
X, y = diabetes.data, diabetes.target

# Split data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Scale features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create datasets
train_dataset = DiabetesDataset(X_train, y_train)
test_dataset = DiabetesDataset(X_test, y_test)

# Create dataloaders
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

# Inspect dataset
print(f"Dataset size: {len(train_dataset)}")
```

```

print(f"Features shape: {train_dataset.X.shape}")
print(f"Targets shape: {train_dataset.y.shape}")

# Get a sample
features, target = train_dataset[0]
print(f"Sample features: {features.shape}") # (10,)
print(f"Sample target: {target.shape}")    # (1,)

```

```

Dataset size: 353
Features shape: torch.Size([353, 10])
Targets shape: torch.Size([353, 1])
Sample features: torch.Size([10])
Sample target: torch.Size([1])

```

50.5. Advanced Usage

50.5.1. Custom Transforms

```

from spotoptim.data import DiabetesDataset
from sklearn.datasets import load_diabetes
import torch

# Define custom transforms
def add_noise(x):
    """Add Gaussian noise to features."""
    return x + torch.randn_like(x) * 0.01

def log_transform(y):
    """Apply log transform to target."""
    return torch.log1p(y)

# Load data
diabetes = load_diabetes()
X, y = diabetes.data, diabetes.target

# Create dataset with transforms
dataset = DiabetesDataset(
    X, y,
    transform=add_noise,
    target_transform=log_transform
)

```

```
# Transforms are applied when accessing items
features, target = dataset[0]
```

50.5.2. Different Train/Test Splits

```
from spotoptim.data import get_diabetes_dataloaders

# 70/30 split
train_loader, test_loader, scaler = get_diabetes_dataloaders(
    test_size=0.3,
    random_state=42
)
print(f"Training samples: {len(train_loader.dataset)}") # ~310
print(f"Test samples: {len(test_loader.dataset)}")      # ~132

# 90/10 split
train_loader, test_loader, scaler = get_diabetes_dataloaders(
    test_size=0.1,
    random_state=42
)
print(f"Training samples: {len(train_loader.dataset)}") # ~398
print(f"Test samples: {len(test_loader.dataset)}")      # ~44
```

```
Training samples: 309
Test samples: 133
Training samples: 397
Test samples: 45
```

50.5.3. Without Feature Scaling

```
from spotoptim.data import get_diabetes_dataloaders

# Load without scaling (useful for tree-based models)
train_loader, test_loader, scaler = get_diabetes_dataloaders(
    scale_features=False
)

print(f"Scaler: {scaler}") # None
```

```
# Data is in original scale
for batch_X, batch_y in train_loader:
    print(f"Mean: {batch_X.mean(dim=0)[:3]}") # Non-zero values
    break
```

```
Scaler: None
Mean: tensor([0.0142, 0.0030, 0.0132])
```

50.5.4. Larger Batch Sizes

```
from spotoptim.data import get_diabetes_dataloaders

# Larger batches for faster training (if memory allows)
train_loader, test_loader, scaler = get_diabetes_dataloaders(
    batch_size=128
)
print(f"Batches per epoch: {len(train_loader)}") # Fewer batches

# Smaller batches for more gradient updates
train_loader, test_loader, scaler = get_diabetes_dataloaders(
    batch_size=8
)
print(f"Batches per epoch: {len(train_loader)}") # More batches
```

```
Batches per epoch: 3
Batches per epoch: 45
```

50.5.5. GPU Training with Pin Memory

```
import torch
from spotoptim.data import get_diabetes_dataloaders

# Enable pin_memory for faster GPU transfer
train_loader, test_loader, scaler = get_diabetes_dataloaders(
    batch_size=32,
    pin_memory=True # Set to True when using GPU
)

# Move model to GPU
```

```

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = model.to(device)

# Training loop with GPU
for batch_X, batch_y in train_loader:
    # Data is already pinned, faster transfer to GPU
    batch_X = batch_X.to(device, non_blocking=True)
    batch_y = batch_y.to(device, non_blocking=True)

    # ... training code ...

```

/Users/bartz/workspace/spotoptim-cookbook/.venv/lib/python3.13/site-packages/torch/utils/data/dataloader.py:692: UserWarning:

'pin_memory' argument is set as true but not supported on MPS now, device pinned memory

50.6. Complete Training Example

Here's a complete example showing data loading, model training, and evaluation:

```

import torch
import torch.nn as nn
from spotoptim.data import get_diabetes_dataloaders
from spotoptim.nn.linear_regressor import LinearRegressor

def train_diabetes_model():
    """Train a neural network on the diabetes dataset."""

    # Load data
    train_loader, test_loader, scaler = get_diabetes_dataloaders(
        test_size=0.2,
        batch_size=32,
        scale_features=True,
        random_state=42
    )

    # Create model
    model = LinearRegressor(
        input_dim=10,
        output_dim=1,
        l1=128,
        num_hidden_layers=3,

```



```

        activation="ReLU"
    )

    # Setup training
    criterion = nn.MSELoss()
    optimizer = model.get_optimizer("Adam", lr=0.001, weight_decay=1e-5)

    # Training configuration
    num_epochs = 200
    best_test_loss = float('inf')

    print("Starting training...")
    print(f"Training samples: {len(train_loader.dataset)}")
    print(f"Test samples: {len(test_loader.dataset)}")
    print(f"Batches per epoch: {len(train_loader)}")
    print("-" * 60)

    for epoch in range(num_epochs):
        # Training phase
        model.train()
        train_loss = 0.0

        for batch_X, batch_y in train_loader:
            # Forward pass
            predictions = model(batch_X)
            loss = criterion(predictions, batch_y)

            # Backward pass
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            train_loss += loss.item()

        avg_train_loss = train_loss / len(train_loader)

        # Evaluation phase
        model.eval()
        test_loss = 0.0

        with torch.no_grad():
            for batch_X, batch_y in test_loader:
                predictions = model(batch_X)
                loss = criterion(predictions, batch_y)

```

```

        test_loss += loss.item()

    avg_test_loss = test_loss / len(test_loader)

    # Track best model
    if avg_test_loss < best_test_loss:
        best_test_loss = avg_test_loss
        # Could save model here: torch.save(model.state_dict(), 'best_model.pt')

    # Print progress
    if (epoch + 1) % 20 == 0:
        print(f"Epoch {epoch+1:3d}/{num_epochs}: "
              f"Train Loss = {avg_train_loss:.4f}, "
              f"Test Loss = {avg_test_loss:.4f}")

    print("-" * 60)
    print(f"Training complete!")
    print(f"Best test loss: {best_test_loss:.4f}")

    return model, best_test_loss

# Run training
if __name__ == "__main__":
    model, best_loss = train_diabetes_model()

```

Starting training...

Training samples: 353

Test samples: 89

Batches per epoch: 12

```

-----
Epoch 20/200: Train Loss = 28948.9683, Test Loss = 26638.0827
Epoch 40/200: Train Loss = 27534.8274, Test Loss = 26633.5352
Epoch 60/200: Train Loss = 27730.8024, Test Loss = 26629.1048
Epoch 80/200: Train Loss = 35064.4762, Test Loss = 26624.5521
Epoch 100/200: Train Loss = 29947.0649, Test Loss = 26620.0820
Epoch 120/200: Train Loss = 33625.0223, Test Loss = 26615.4421
Epoch 140/200: Train Loss = 32708.0872, Test Loss = 26610.6647
Epoch 160/200: Train Loss = 28962.4118, Test Loss = 26605.8444
Epoch 180/200: Train Loss = 28625.3312, Test Loss = 26600.7832
Epoch 200/200: Train Loss = 34682.6623, Test Loss = 26595.5202
-----

```

Training complete!

Best test loss: 26595.5202

50.7. Integration with SpotOptim

Use the diabetes dataset for hyperparameter optimization with SpotOptim:

```
import numpy as np
import torch
import torch.nn as nn
from spotoptim import SpotOptim
from spotoptim.data import get_diabetes_dataloaders
from spotoptim.nn.linear_regressor import LinearRegressor

def evaluate_model(X):
    """Objective function for SpotOptim.

    Args:
        X: Array of hyperparameters [lr, l1, num_hidden_layers]

    Returns:
        Array of validation losses
    """
    results = []

    for params in X:
        lr, l1, num_hidden_layers = params
        lr = 10 ** lr # Log scale for learning rate
        l1 = int(l1)
        num_hidden_layers = int(num_hidden_layers)

        # Load data
        train_loader, test_loader, _ = get_diabetes_dataloaders(
            test_size=0.2,
            batch_size=32,
            random_state=42
        )

        # Create model
        model = LinearRegressor(
            input_dim=10,
            output_dim=1,
            l1=l1,
            num_hidden_layers=num_hidden_layers,
            activation="ReLU"
        )
```

```

# Train briefly
criterion = nn.MSELoss()
optimizer = model.get_optimizer("Adam", lr=lr)

num_epochs = 50
for epoch in range(num_epochs):
    model.train()
    for batch_X, batch_y in train_loader:
        predictions = model(batch_X)
        loss = criterion(predictions, batch_y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

# Evaluate
model.eval()
test_loss = 0.0
with torch.no_grad():
    for batch_X, batch_y in test_loader:
        predictions = model(batch_X)
        loss = criterion(predictions, batch_y)
        test_loss += loss.item()

results.append(test_loss / len(test_loader))

return np.array(results)

# Optimize hyperparameters
optimizer = SpotOptim(
    fun=evaluate_model,
    bounds=[
        (-4, -2), # log10(lr): 0.0001 to 0.01
        (16, 128), # l1: number of neurons
        (0, 4) # num_hidden_layers
    ],
    var_type=["float", "int", "int"],
    max_iter=30,
    n_initial=10,
    seed=42,
    verbose=True
)

result = optimizer.optimize()
print(f"Best hyperparameters found:")

```

```
print(f"  Learning rate: {10**result.x[0]:.6f}")
print(f"  Hidden neurons (l1): {int(result.x[1])}")
print(f"  Hidden layers: {int(result.x[2])}")
print(f"  Best MSE: {result.fun:.4f}")
```

TensorBoard logging disabled

Initial best: $f(x) = 26560.914062$

Iteration 1: $f(x) = 26619.498047$

Iteration 2: $f(x) = 26657.300130$

Attempt 2/10: Previous point was duplicate after rounding, trying fallback

Acquisition failure: Using random space-filling design as fallback.

Iteration 3: $f(x) = 26630.593750$

Attempt 2/10: Previous point was duplicate after rounding, trying fallback

Acquisition failure: Using random space-filling design as fallback.

Iteration 4: $f(x) = 26684.070964$

Attempt 2/10: Previous point was duplicate after rounding, trying fallback

Acquisition failure: Using random space-filling design as fallback.

Iteration 5: $f(x) = 26571.974609$

Attempt 2/10: Previous point was duplicate after rounding, trying fallback

Acquisition failure: Using random space-filling design as fallback.

Iteration 6: $f(x) = 26645.708984$

Attempt 2/10: Previous point was duplicate after rounding, trying fallback

Acquisition failure: Using random space-filling design as fallback.

Iteration 7: New best $f(x) = 26518.239583$

Attempt 2/10: Previous point was duplicate after rounding, trying fallback

Acquisition failure: Using random space-filling design as fallback.

Iteration 8: $f(x) = 26651.987630$

Attempt 2/10: Previous point was duplicate after rounding, trying fallback

Acquisition failure: Using random space-filling design as fallback.

Iteration 9: $f(x) = 26621.998047$

Attempt 2/10: Previous point was duplicate after rounding, trying fallback

Acquisition failure: Using random space-filling design as fallback.

Iteration 10: $f(x) = 26546.020182$

Attempt 2/10: Previous point was duplicate after rounding, trying fallback

Acquisition failure: Using random space-filling design as fallback.

Iteration 11: $f(x) = 26642.139323$

Attempt 2/10: Previous point was duplicate after rounding, trying fallback

Acquisition failure: Using random space-filling design as fallback.

Iteration 12: $f(x) = 26545.641276$

Attempt 2/10: Previous point was duplicate after rounding, trying fallback

Acquisition failure: Using random space-filling design as fallback.

Iteration 13: $f(x) = 26573.894531$

Attempt 2/10: Previous point was duplicate after rounding, trying fallback

Acquisition failure: Using random space-filling design as fallback.

50. Diabetes Dataset Utilities

Iteration 14: $f(x) = 26657.354167$

Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.

Iteration 15: $f(x) = 26578.886068$

Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.

Iteration 16: $f(x) = 26606.752604$

Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.

Iteration 17: $f(x) = 26688.576823$

Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.

Iteration 18: $f(x) = 26639.442708$

Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.

Iteration 19: $f(x) = 26620.895833$

Attempt 2/10: Previous point was duplicate after rounding, trying fallback
Acquisition failure: Using random space-filling design as fallback.

Iteration 20: $f(x) = 26578.162760$

Best hyperparameters found:

Learning rate: 0.005901

Hidden neurons (l1): 53

Hidden layers: 2

Best MSE: 26518.2396

50.8. Best Practices

50.8.1. 1. Always Use Feature Scaling

```
# Good: Features are standardized
train_loader, test_loader, scaler = get_diabetes_data loaders(
    scale_features=True
)
```

Neural networks typically perform better with normalized inputs.

50.8.2. 2. Set Random Seeds for Reproducibility

```
# Reproducible train/test splits
train_loader, test_loader, scaler = get_diabetes_dataloaders(
    random_state=42
)

# Also set PyTorch seed
import torch
torch.manual_seed(42)
```

```
<torch._C.Generator at 0x116961e50>
```

50.8.3. 3. Don't Shuffle Test Data

```
# Good: Test data in consistent order
train_loader, test_loader, scaler = get_diabetes_dataloaders(
    shuffle_train=True,    # Shuffle training data
    shuffle_test=False     # Don't shuffle test data
)
```

This ensures consistent evaluation metrics across runs.

50.8.4. 4. Choose Appropriate Batch Size

```
# Small dataset (442 samples) - moderate batch size works well
train_loader, test_loader, scaler = get_diabetes_dataloaders(
    batch_size=32 # Good balance for this dataset
)
```

Too large: Fewer gradient updates per epoch
 Too small: Noisy gradients, slower training

50.8.5. 5. Save the Scaler for Production

```
import pickle
import numpy as np
from spotoptim.data import get_diabetes_dataloaders
```

```
# Train with scaling
train_loader, test_loader, scaler = get_diabetes_data loaders(
    scale_features=True
)

# Save scaler for production use
with open('scaler.pkl', 'wb') as f:
    pickle.dump(scaler, f)

# Later: Load and use on new data
with open('scaler.pkl', 'rb') as f:
    loaded_scaler = pickle.load(f)

# Create some example new data (same shape as diabetes features)
new_data = np.random.randn(5, 10) # 5 samples, 10 features
new_data_scaled = loaded_scaler.transform(new_data)

print(f"Original data shape: {new_data.shape}")
print(f"Scaled data shape: {new_data_scaled.shape}")
print(f"Scaled data mean: {new_data_scaled.mean(axis=0)[:3]}") # Should be close to 0
```

```
Original data shape: (5, 10)
Scaled data shape: (5, 10)
Scaled data mean: [7.45507756 0.05846817 5.04323937]
```

50.9. Troubleshooting

50.9.1. Issue: Out of Memory

Solution: Reduce batch size or disable pin_memory

```
train_loader, test_loader, scaler = get_diabetes_data loaders(
    batch_size=16, # Smaller batches
    pin_memory=False # Disable if not using GPU
)
```

50.9.2. Issue: Different Data Ranges

Symptom: Model not converging, loss is NaN

Solution: Ensure feature scaling is enabled


```
train_loader, test_loader, scaler = get_diabetes_dataloaders(
    scale_features=True # Must be True for neural networks
)
```

50.9.3. Issue: Non-Reproducible Results

Solution: Set all random seeds

```
import torch
import numpy as np

# Set all seeds
torch.manual_seed(42)
np.random.seed(42)

train_loader, test_loader, scaler = get_diabetes_dataloaders(
    random_state=42,
    shuffle_train=False # Disable shuffle for full reproducibility
)
```

50.9.4. Issue: Slow Data Loading

Solution: Use multiple workers (if not on Windows)

```
train_loader, test_loader, scaler = get_diabetes_dataloaders(
    num_workers=4, # Use 4 subprocesses
    pin_memory=True # Enable for GPU
)
```

Note: On Windows, set `num_workers=0` to avoid multiprocessing issues.

50.10. Summary

The diabetes dataset utilities in SpotOptim provide:

- **Easy data loading:** One function call gets complete data pipeline
- **PyTorch integration:** Native Dataset and DataLoader support
- **Preprocessing included:** Automatic feature scaling and train/test splitting
- **Flexible configuration:** Control batch size, splitting, scaling, and more
- **Production ready:** Save scalers and ensure reproducibility

50.11. Jupyter Notebook

Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

Part X.

Details About SPOT

51. SpotOptim Step-by-Step Optimization Process

52. Introduction

This document provides a comprehensive step-by-step explanation of the optimization process in the `SpotOptim` class. We'll use the 2-dimensional Rosenbrock function as our example and cover all methods involved in each stage of optimization.

Topics Covered:

- Standard optimization workflow
- Handling noisy functions with repeats
- Handling function evaluation failures (NaN/inf values)

53. Setup and Test Functions

Let's start by defining our test functions, including variants with noise and occasional failures. The two-dimensional Rosenbrock function is defined as:

$$f(x, y) = (a - x)^2 + b(y - x^2)^2$$

where typically ($a = 1$) and ($b = 100$). The generalized form for n dimensions is:

$$f(X) = \sum_{i=1}^{N-1} [100 \cdot (x_{i+1} - x_i^2)^2 + (1 - x_i)^2]$$

The documentation can be found here: [DOC](#)

```
import numpy as np
from spotoptim import SpotOptim
from spotoptim.function import rosenbrock
import warnings
warnings.filterwarnings('ignore')
```

Set random seed for reproducibility:

```
np.random.seed(42)
```

Based on the standard Rosenbrock function, we define two variants:

1. the noisy Rosenbrock function:

```
def rosenbrock_noisy(X, noise_std=0.1):
    """
    Rosenbrock with Gaussian noise for testing noisy optimization.
    """
    X = np.atleast_2d(X)
    base_values = rosenbrock(X)
    noise = np.random.normal(0, noise_std, size=base_values.shape)
    return base_values + noise
```

and 2. the Rosenbrock function with occasional failures:

53. Setup and Test Functions

```
def rosenbrock_with_failures(X, failure_prob=0.15):  
    """  
    Rosenbrock that occasionally returns NaN to simulate evaluation failures.  
    """  
    X = np.atleast_2d(X)  
    values = rosenbrock(X)  
  
    # Randomly inject failures  
    for i in range(len(values)):  
        if np.random.random() < failure_prob:  
            values[i] = np.nan  
  
    return values
```

54. Standard Optimization Workflow

Let's trace through a complete optimization run to understand each step.

54.1. Phase: Initialization and Setup

When you create a `SpotOptim` instance, several initialization steps occur during the `__init__()` method, see DOC.

1. Objective function (`fun`) and bounds are stored.
2. Acquisition function (`acquisition`) is stored (default is `y`).
3. Determine if noise handling is active (based on `repeats_initial` and `repeats_surrogate`)
4. For dimensions with tuple bounds (factor variables), internal integer mappings are created and bounds are replaced with $(0, n\_levels-1)$. This is handled by `_process_factor_bounds()`. So, e.g., `[('red', 'green', 'blue')]` is mapped to the integer interval $[(0, 2)]$ internally.
5. The dimension of the problem (`n_dim`) is inferred from the bounds.
6. If `var_type` is not provided, it defaults to all “float” (continuous) variables, except for factor variables which are set to “factor” via `_process_factor_bounds()`.
7. Default variable names (`var_name`) are set if not provided.
8. Default variable transformations (`var_transform`) are set if not provided.
9. Transformations are applied to bounds based on `var_transform` settings. Natural bounds are stored in `_original_lower` and `_original_upper`.
10. Dimension reduction by identifying fixed dimensions (if any) is performed via `_setup_dimension_reduction()`.
11. The surrogate is initialized (default: Gaussian Process with Matérn kernel) as follows:

```
kernel = ConstantKernel(  
    constant_value=1.0, constant_value_bounds=(1e-3, 1e3)  
) * Matern(length_scale=1.0, length_scale_bounds=(1e-2, 1e2), nu=2.5)  
self.surrogate = GaussianProcessRegressor(  
    kernel=kernel,  
    n_restarts_optimizer=10,  
    normalize_y=True,
```

```

    random_state=self.seed,
)

```

12. The Design generator is initialized (default: Latin Hypercube Sampling).
13. The storage for results ins initialized. This includes the following attributes:

- `X_`: Evaluated design points
- `y_`: Corresponding function values
- `y_mo`: For multi-objective functions, stores all objectives
- `best_x_`: Best point found so far
- `best_y_`: Best function value found so far
- `n_iter_`: Number of iterations completed
- `mean_X, mean_y, var_y`: For noisy functions, mean and variance tracking
- `min_mean_X`: Best mean point for noisy functions
- `min_mean_y`: Best mean value for noisy functions
- `min_var_y`: Variance at best mean point
- `min_X`: Best point found (deterministic)
- `min_y`: Best function value found (deterministic)
- `counter`: Total number of function evaluations
- `success_rate`: Ratio of successful evaluations
- `success_counter`: Count of successful evaluations
- `window_size`: For moving average of success rate
- `success_history`: History of success/failure for evaluations

```

# Create optimizer
opt = SpotOptim(
    fun=rosenbrock,
    bounds=[(-2, 2), (1, 2)],
    n_initial=10,
    max_iter=30,
    verbose=True,
    seed=42,
    var_trans=["id", "log"]
)

```

TensorBoard logging disabled

54.2. Phase: Initial Design Generation

54.2.1. Method: `get_initial_design()`

This method generates or processes the initial sample points. It is manually called here for demonstration (normally done inside `optimize()`).

What happens in `get_initial_design()`:

1. If `X0=None`: Generate space-filling design using Latin Hypercube Sampling (LHS)
2. If `x0` (starting point) provided: Include it as first point and transform user's points to internal scale
3. Apply dimension reduction if configured
4. Round integer/factor variables

```
X0 = opt.get_initial_design(X0=None)

print(f"Generated {len(X0)} initial design points using Latin Hypercube Sampling")
print(f"Design shape: {X0.shape}")
print(f"\nFirst 3 points (internal scale):")
print(X0[:3])
```

```
Generated 10 initial design points using Latin Hypercube Sampling
Design shape: (10, 2)
```

```
First 3 points (internal scale):
[[-1.10958242  0.45478229]
 [ 1.65656083  0.228921  ]
 [-1.63767094  0.55620747]]
```

Note, because `bounds` were set to `[(-2, 2), (-2, 2)]` and `n_initial=10`, the generated points are within this range and the design has two-dimensional shape (10, 2).

- Print initial design in original scale and in internal scale:

```
X0_original = opt._inverse_transform_X(X0)
print(f"\nFirst 3 points (original scale):")
print(X0_original[:3])
print(f"\nFirst 3 points (internal scale):")
print(X0[:3])
```

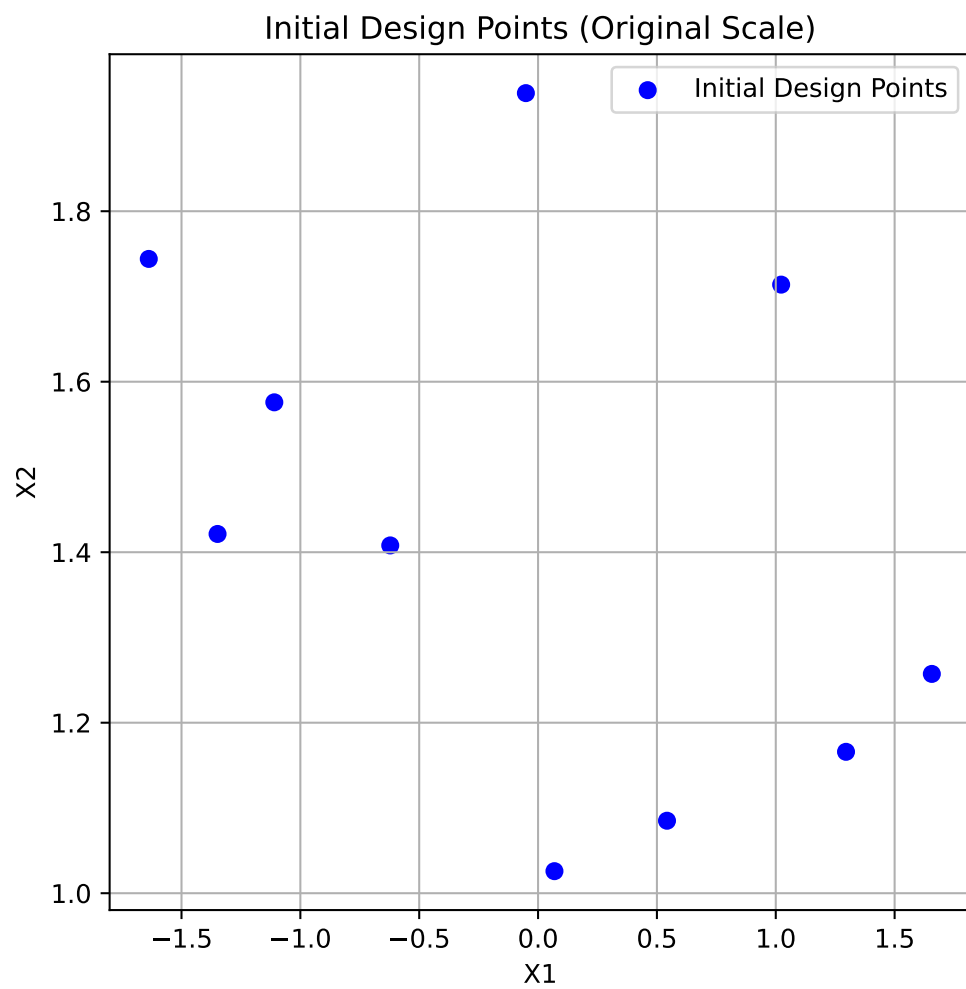
```
First 3 points (original scale):
[[-1.10958242  1.57583027]
 [ 1.65656083  1.25724272]
 [-1.63767094  1.7440456  ]]
```

```
First 3 points (internal scale):
[[-1.10958242  0.45478229]
 [ 1.65656083  0.228921  ]
 [-1.63767094  0.55620747]]
```

54. Standard Optimization Workflow

- Plot initial design in original scale:

```
import matplotlib.pyplot as plt
plt.figure(figsize=(6, 6))
plt.scatter(X0_original[:, 0], X0_original[:, 1], c='blue', label='Initial Design Points')
plt.xlabel("X1")
plt.ylabel("X2")
plt.title("Initial Design Points (Original Scale)")
plt.legend()
plt.grid(True)
plt.show()
```



54.3. Phase 3: Initial Design Curation

54.3.1. Method: `_curate_initial_design()`

This method ensures we have sufficient unique points and handles repeats.

What happens in `_curate_initial_design()`:

1. Remove duplicate points (can occur after rounding integers)
2. Generate additional points if duplicates reduced count below `n_initial`
3. Repeat each point `repeats_initial` times if > 1 (for noisy functions)

```
X0_curated = opt._curate_initial_design(X0)
print(f"Curated design shape: {X0_curated.shape}")
print(f"Unique points: {len(np.unique(X0_curated, axis=0))}")
print(f"Total points (with repeats): {len(X0_curated)}")
```

```
Curated design shape: (10, 2)
Unique points: 10
Total points (with repeats): 10
```

54.4. Phase 4: Initial Design Evaluation

54.4.1. Method: `_evaluate_function()`

Evaluates the objective function at all initial design points. The points are converted back to the original scale for evaluation.

What happens in `_evaluate_function()`:

1. Convert points from internal to original scale
2. Call objective function with batch of points
3. Convert multi-objective to single-objective if needed
4. Return array of function values

```
X0_original = opt._inverse_transform_X(X0_curated)
y0 = opt._evaluate_function(X0_curated)

print(f"Evaluated {len(y0)} points")
print(f"Function values shape: {y0.shape}")
print(f"First 5 points with function values:")
for i in range(min(5, len(y0))):
    print(f"  Point {i+1}: {X0_original[i]} → f(x)={y0[i]:.6f}")
```

54. Standard Optimization Workflow

```
print(f"\nBest initial value: {np.min(y0):.6f}")
print(f"Worst initial value: {np.max(y0):.6f}")
print(f"Mean initial value: {np.mean(y0):.6f}")
```

Evaluated 10 points

Function values shape: (10,)

First 5 points with function values:

```
Point 1: [-1.10958242  1.57583027] → f(x)=16.329192
Point 2: [1.65656083  1.25724272] → f(x)=221.533421
Point 3: [-1.63767094  1.7440456 ] → f(x)=94.926795
Point 4: [1.29554412  1.1658592 ] → f(x)=26.360696
Point 5: [-0.05124545  1.93852778] → f(x)=375.876648
```

Best initial value: 16.329192

Worst initial value: 375.876648

Mean initial value: 107.571208

54.5. Phase 5: Handling Failed Evaluations

54.5.1. Method: `_handle_NA_initial_design(X0,y0)`

Removes points that returned NaN or inf values. In contrast to later phases, no penalties are applied here; invalid points of the initial design are simply removed.

What happens in `_handle_NA_initial_design()`:

1. Identify NaN/inf values in function evaluations
2. Remove corresponding design points
3. Return cleaned arrays and original count
4. Note: No penalties applied in initial design - invalid points simply removed

```
n_before = len(y0)
X0_clean, y0_clean, n_evaluated = opt._handle_NA_initial_design(X0_curated, y0)

print(f"Points before filtering: {n_before}")
print(f"Points after filtering: {len(y0_clean)}")
print(f"Removed: {n_before - len(y0_clean)} NaN/inf values")
print(f"\nAll remaining values finite: {np.all(np.isfinite(y0_clean))}")
```

Points before filtering: 10

Points after filtering: 10

Removed: 0 NaN/inf values

All remaining values finite: True

54.6. Phase 6: Validation Check

54.6.1. Method: `_check_size_initial_design(y0, n_evaluated)`

Ensures we have enough valid points to continue. The minimum required is the smaller of:

- typical minimum for surrogate fitting (3 for multi-dimensional, 2 for 1D), or
- what the user requested (`n_initial`).

What happens in `_check_size_initial_design()`:

1. Check if at least 1 valid point exists
2. Raise error if all initial evaluations failed
3. Print warnings if many points were invalid

```
try:
    opt._check_size_initial_design(y0_clean, n_evaluated)
    print(f" Validation passed: {len(y0_clean)} valid points available")
    print(f" Minimum required: 1 point")
    print(f" Original evaluated: {n_evaluated} points")
except ValueError as e:
    print(f" Validation failed: {e}")
```

```
Validation passed: 10 valid points available
Minimum required: 1 point
Original evaluated: 10 points
```

54.7. Phase 7: Storage Initialization

54.7.1. Method: `_init_storage()`

Store evaluated points and initialize tracking variables.

What happens during storage initialization:

1. `_init_storage()`: Initialize statistics tracking variables.
 - Initialize storage (as done in `optimize()`)

54. Standard Optimization Workflow

```
# Initialize storage and statistics using the new _init_storage() method
opt._init_storage(X0_clean, y0_clean)
```

```
print(f"X_ (evaluated points): shape {opt.X_.shape}")
print(f"y_ (function values): shape {opt.y_.shape}")
print(f"n_iter_ (iterations): {opt.n_iter_}")
```

```
X_ (evaluated points): shape (10, 2)
y_ (function values): shape (10,)
n_iter_ (iterations): 0
```

54.8. Phase: Statistics Update

54.8.1. Method: `update_stats()`

Update statistics like means and variances for noisy evaluations.

*** What happens in `update_stats()`:***

1. If noise is set (i.e., `repeats_initial > 1`):
 - Compute mean and variance of function values for each unique point
 - Store in `mean_X`, `mean_y`, and `var_y`

```
print(f"\nStatistics updated:")
if opt.noise:
    print(f" - mean_X: {opt.mean_X.shape}")
    print(f" - mean_y: {opt.mean_y.shape}")
    print(f" - var_y: {opt.var_y.shape}")
else:
    print(f" - No noise tracking (repeats=1)")
```

```
Statistics updated:
- No noise tracking (repeats=1)
```

54.9. Phase: Log initial Design to TensorBoard

54.9.1. Method: `_log_initial_design_tensorboard()`

This method logs the initial design points and their evaluations to TensorBoard for visualization.

54.10. Phase: Initial Best Point

54.10.1. Method: `_get_best_xy_initial_design()`

Identify and report the best point from initial design.

What happens in `_get_best_xy_initial_design()`:

1. Find minimum value in `y_` (or `mean_y` if noise)
2. Store as `best_y_`
3. Store corresponding point as `best_x_`
4. Print progress if verbose

```
opt._get_best_xy_initial_design()

print(f"Best point found: {opt.best_x_}")
print(f"Best value found: {opt.best_y_:.6f}")
print(f"\nOptimum location: [1, 1]")
print(f"Optimum value: 0")
print(f"Current gap: {opt.best_y_ - 0:.6f}")
```

```
Initial best: f(x) = 16.329192
Best point found: [-1.10958242  1.57583027]
Best value found: 16.329192
```

```
Optimum location: [1, 1]
Optimum value: 0
Current gap: 16.329192
```

54.11. Illustration of the Initial Design Phase Results

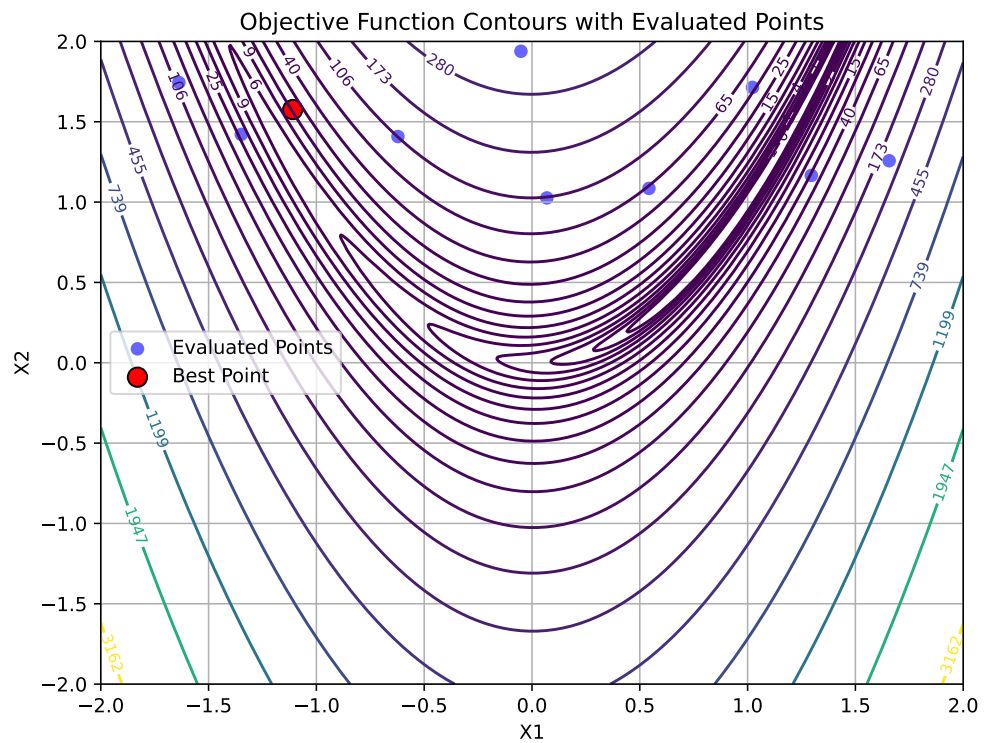
To visualize the results, we generate a contour plot with contour lines of the objective function and mark the best point. The other evaluated points are shown as well:

```
# Create a grid of points for contour plotting
x = np.linspace(-2, 2, 400)
y = np.linspace(-2, 2, 400)
X, Y = np.meshgrid(x, y)
grid_points = np.column_stack([X.ravel(), Y.ravel()])
Z = rosenbrock(grid_points).reshape(X.shape)
plt.figure(figsize=(8, 6))
# Contour plot
```

54. Standard Optimization Workflow

```
contour = plt.contour(X, Y, Z, levels=np.logspace(-0.5, 3.5, 20), cmap='viridis')
plt.clabel(contour, inline=True, fontsize=8)

# Plot evaluated points from the initial design phase:
plt.scatter(opt.X[:, 0], opt.X[:, 1], c='blue', label='Evaluated Points', alpha=0.6)
# Mark best point
plt.scatter(opt.best_x[0], opt.best_x[1], c='red', s=100, label='Best Point', edgecolor='black')
plt.xlabel('X1')
plt.ylabel('X2')
plt.title('Objective Function Contours with Evaluated Points')
plt.legend()
plt.grid(True)
plt.show()
```



55. Sequential Optimization Loop

After initialization, the main optimization loop begins. Each iteration follows these steps:

55.1. Step: Surrogate Model Fitting

55.1.1. Method: `_fit_scheduler()`

Fit surrogate model using appropriate data based on noise handling.

*** What happens in `_fit_scheduler()`:***

First, the method transforms the input data `X` to the internal scale used by the optimizer. Then, it decides which data to use for fitting the surrogate model based on whether noise handling is enabled:

1. If `noise` is set (i.e., `repeats_surrogate > 1`):
 - Fit surrogate using mean points (`mean_X`, `mean_y`)
2. Else:
 - Fit surrogate using all evaluated points (`X_`, `y_`)

`_fit_scheduler()` then calls `_fit_surrogate()` with the selected data.

55.1.2. Method: `_fit_surrogate()`

Fit a surrogate model (Gaussian Process) to current data.

What happens in `_fit_surrogate()`:

1. If `max_surrogate_points` set and exceeded: Select subset of points using the `_selection_dispatcher()` method.
2. Fit surrogate model using `surrogate.fit(X, y)`
3. Surrogate learns the function landscape and uncertainty

55. Sequential Optimization Loop

```
X_for_surrogate = opt._transform_X(opt.X_)
opt._fit_surrogate(X_for_surrogate, opt.y_)

print(f"Surrogate fitted with {len(opt.y_)} points")
print(f"Surrogate type: {type(opt.surrogate).__name__}")
print(f"Kernel: {opt.surrogate.kernel_}")
```

```
Surrogate fitted with 10 points
Surrogate type: GaussianProcessRegressor
Kernel: 1.06**2 * Matern(length_scale=0.355, nu=2.5)
```

55.1.3. Method: `_selection_dispatcher()`

What happens in `_selection_dispatcher()`:

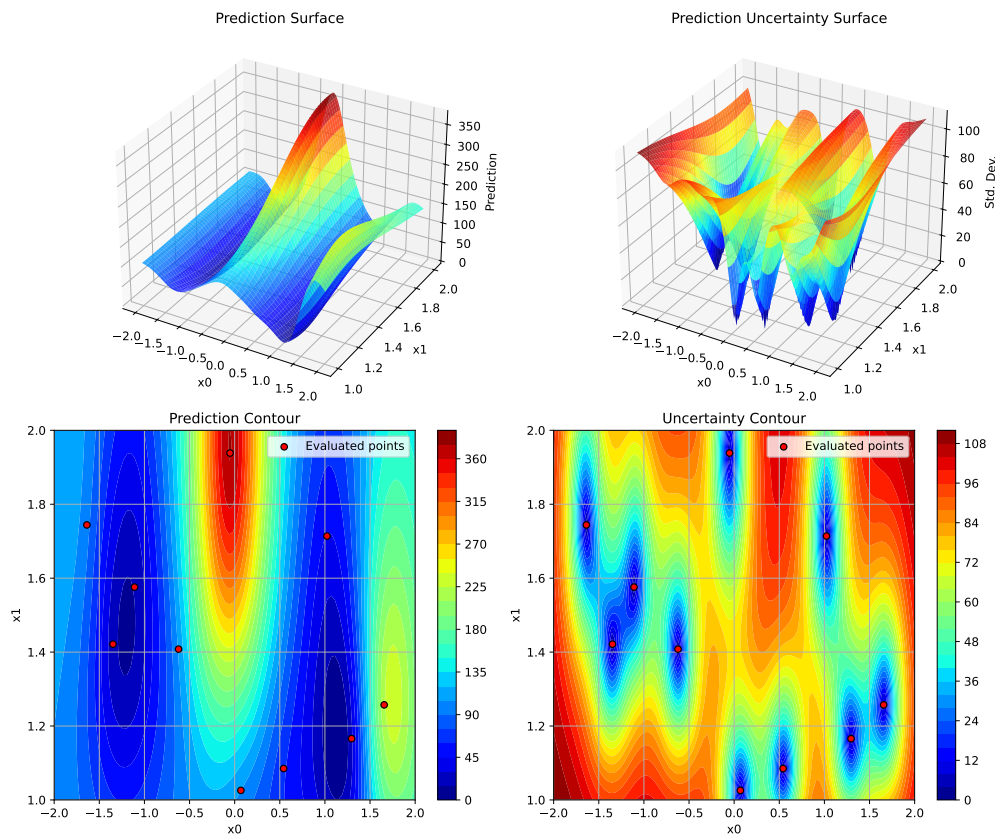
1. If `max_surrogate_points` is set and exceeded:
 - Select a subset of points from `X_` and `y_` for fitting the surrogate model.
 - Strategies may include random sampling, clustering, or other heuristics to reduce the dataset size while preserving important information.

Surrogate Model Selection:

- Default: Gaussian Process with Matérn kernel
- Provides: Mean prediction $\mu(x)$ and uncertainty $\sigma(x)$
- Can be replaced with Random Forest, Kriging, etc.
- The fitted surrogate can be plotted with the following code:

```
opt.plot_surrogate()
```

55.1. Step: Surrogate Model Fitting



55.1.4. Step: Apply OCBA

55.1.5. Method: `_apply_ocba()`

What happens in `_apply_ocba()`:

1. Compute the optimality criteria for each point in the design space.
2. Select the most promising points based on the criteria.
3. Update the surrogate model with the selected points.

```
X_ocba = opt._apply_ocba()
if X_ocba is not None:
    print(f"OCBA selected {X_ocba.shape[0]} points for re-evaluation")
    print(f"OCBA points shape: {X_ocba.shape}")
else:
    print("OCBA not applied (noise=False or ocba_delta=0)")
```

55. Sequential Optimization Loop

OCBA not applied (noise=False or ocba_delta=0)

55.2. Step: Predict with Uncertainty

55.2.1. Method: `_predict_with_uncertainty()`

Make predictions at new locations with uncertainty estimates.

What happens in `_predict_with_uncertainty()`:

1. Call `surrogate.predict(X, return_std=True)`
2. Returns mean (μ) and standard deviation (σ)
3. Used by acquisition function to balance exploitation (low σ) and exploration (high σ)

Here we test prediction at a few points:

```
X_test = np.array([[0.5, 0.5], [1.0, 1.0], [-0.5, 0.5]])
mu, sigma = opt._predict_with_uncertainty(X_test)

print(f"Test points: {X_test.shape}")
print(f"\nPredictions (  $\mu$   $\pm$   $\sigma$  ):")
for i, (x, m, s) in enumerate(zip(X_test, mu, sigma)):
    print(f"  x={x}  $\rightarrow$   $\mu$ ={m:.4f},  $\sigma$ ={s:.4f}")
```

Test points: (3, 2)

```
Predictions (  $\mu$   $\pm$   $\sigma$  ):
  x=[0.5 0.5]  $\rightarrow$   $\mu$ =135.9904,  $\sigma$ =96.1193
  x=[1. 1.]  $\rightarrow$   $\mu$ =97.6979,  $\sigma$ =104.5930
  x=[-0.5 0.5]  $\rightarrow$   $\mu$ =169.1890,  $\sigma$ =64.4389
```

55.3. Step: Next Point Suggestion

55.3.1. Method: `_suggest_next_point()`

Optimize acquisition function to find next evaluation point.

What happens in `_suggest_next_point()`:

1. Use `differential_evolution` to optimize acquisition function
2. Find point that maximizes EI (or minimizes predicted value for 'y')

3. Apply rounding for integer/factor variables
4. Check distance to existing points (avoid duplicates)
5. If duplicate or too close: Use fallback strategy
6. Return suggested point

```
x_next = opt._suggest_next_point()
print(f"Next point suggested: {x_next}")
```

Next point suggested: [1.13866324 0.15750508]

Predict at suggested point:

```
mu_next, sigma_next = opt._predict_with_uncertainty(x_next.reshape(1, -1))
print(f"\nPrediction at suggested point:")
print(f"    (x_next) = {mu_next[0]:.4f}")
print(f"    (x_next) = {sigma_next[0]:.4f}")
```

Prediction at suggested point:

```
(x_next) = 1.3255
(x_next) = 50.4602
```

Fallback Strategies (if acquisition optimization fails):

- 'best': Re-evaluate current best point
- 'mean': Select point with best predicted mean
- 'random': Random point in search space

55.4. Step: Acquisition Function Evaluation

55.4.1. Method: `_acquisition_function()`

Compute acquisition function value to guide search. The acquisition function is used as an objective function by the optimizer on the surrogate, e.g., differential evolution. It determines where to sample next.

```
# Evaluate acquisition at test points
print(f"Acquisition type: {opt.acquisition} (Expected Improvement)")
print(f"\nAcquisition values at test points:")

for x in X_test:
    acq_val = opt._acquisition_function(x)
    print(f"    x={x} → acq={acq_val:.6f}")
```

55. Sequential Optimization Loop

Acquisition type: y (Expected Improvement)

Acquisition values at test points:

x=[0.5 0.5] → acq=135.990446

x=[1. 1.] → acq=97.697886

x=[-0.5 0.5] → acq=169.189001

Acquisition function can be used to balance:

- exploitation, i.e., low predicted mean $\mu(x)$ and
- exploration, i.e., high uncertainty $\sigma(x)$

Acquisition Functions Available:

1. **Expected Improvement (EI)** - Default, best balance

$$\text{EI}(x) = (f^* - \mu(x))\Phi(Z) + \sigma(x)\phi(Z)$$

$$\text{where } Z = \frac{f^* - \mu(x)}{\sigma(x)}$$

2. **Probability of Improvement (PI)** - More exploitative

$$\text{PI}(x) = \Phi\left(\frac{f^* - \mu(x)}{\sigma(x)}\right)$$

3. **Mean ('y')** - Pure exploitation

$$\text{acq}(x) = \mu(x)$$

55.5. Step: Update Repeats for Infill Points

55.5.1. Method: `_update_repeats_infill_points()`

Repeat the infill point for noisy function evaluation if `repeats_surrogate > 1`.

What happens in `_update_repeats_infill_points()`:

1. Takes the suggested next point (`x_next`)
2. If `repeats_surrogate > 1`: Creates multiple copies for repeated evaluation
3. Otherwise: Returns point as 2D array (shape: $1 \times n_{\text{features}}$)
4. Returns array ready for function evaluation

```
x_next_repeated = opt._update_repeats_infill_points(x_next)
print(f"Shape before repeating: {x_next.shape}")
print(f"Shape after repeating: {x_next_repeated.shape}")
print(f"Number of evaluations planned: {x_next_repeated.shape[0]}")
```

Shape before repeating: (2,)
 Shape after repeating: (1, 2)
 Number of evaluations planned: 1

55.6. Append OCBA Points to Infill Points

Combines OCBA selected points with the next suggested point for evaluation.

```
if X_ocba is not None:
    x_next_repeated = np.append(X_ocba, x_next_repeated, axis=0)
```

55.7. Step: Evaluation of New Points

55.7.1. Method: `_evaluate_function()` (again)

Evaluate the objective function at the suggested points.

```
x_next_2d = x_next.reshape(1, -1)
y_next = opt._evaluate_function(x_next_2d)

print(f"Evaluated point: {x_next}")
print(f"Function value: {y_next[0]:.6f}")
print(f"Current best: {opt.best_y:.6f}")
```

```
Evaluated point: [1.13866324 0.15750508]
Function value: 1.606003
Current best: 16.329192
```

55.8. Step: Handle Failed Evaluations (Sequential)

55.8.1. Method: `_handle_NA_new_points()`

Handle NaN/inf values in new evaluations with penalty approach.

What happens in `_handle_NA_new_points()`:

1. **Apply penalty** to NaN/inf values (unlike initial design)
 - $\text{Penalty} = \max(\text{history}) + 3 \times \text{std}(\text{history}) + \text{random noise}$

55. Sequential Optimization Loop

2. **Remove** remaining invalid values after penalty
3. **Return None** if all evaluations failed → skip iteration
4. **Continue** if any valid evaluations exist

```
x_clean, y_clean = opt._handle_NA_new_points(x_next_2d, y_next)

if x_clean is not None:
    print(f" Valid evaluations: {len(y_clean)}")
    print(f" All values finite: {np.all(np.isfinite(y_clean))}")
else:
    print(f" All evaluations failed - iteration would be skipped")
```

```
Valid evaluations: 1
All values finite: True
```

Why penalties in sequential phase?

- Preserves optimization history
- Allows surrogate to learn “bad” regions
- Random noise prevents identical penalties

55.9. Step: Update Success Rate

55.9.1. Method: `_update_success_rate(y0)`

Update success rate BEFORE updating storage (initial design - all should be successes since starting from scratch)

What happens in `_update_success_rate()`:

1. Calculate success rate as ratio of valid evaluations to total evaluated

```
opt._update_success_rate(y0_clean)
print(f"\nSuccess rate updated: {opt.success_rate:.2%} (valid evaluations / total eval
```

```
Success rate updated: 0.00% (valid evaluations / total evaluations)
```

55.10. Step: Update Storage

55.10.1. Internal updates

Add new evaluations to storage.

```
opt._update_storage(x_next_repeated, y_next)
```

55.11. Update Statistics

Update statistics after new evaluations.

55.11.1. What happens in `update_stats()`:

Always updated:

- `min_y`: Best (minimum) function value
- `min_X`: Design point with best value
- `counter`: Total number of evaluations

For noisy functions only (if `noise=True`):

- `mean_X`: Unique design points
- `mean_y`: Mean values per design point
- `var_y`: Variance per design point

```
opt.update_stats()
print(f"Basic statistics:")
print(f"  min_y: {opt.min_y:.6f}")
print(f"  min_X: {opt.min_X}")
print(f"  counter: {opt.counter}")

if opt.noise:
    print(f"\nNoise statistics:")
    print(f"  mean_X shape: {opt.mean_X.shape}")
    print(f"  mean_y shape: {opt.mean_y.shape}")
    print(f"  var_y shape: {opt.var_y.shape}")
else:
    print(f"\nNoise handling: disabled (deterministic function)")
```

55. Sequential Optimization Loop

```
Basic statistics:
  min_y: 1.606003
  min_X: [1.13866324 1.1705867 ]
  counter: 11
```

Noise handling: disabled (deterministic function)

55.12. Step: Update Best Solution

55.12.1. Method: `_update_best_main_loop()`

Update the best solution if improvement found.

```
best_before = opt.best_y_
opt._update_best_main_loop(x_clean, y_clean)

print(f"Best before: {best_before:.6f}")
print(f"Best after: {opt.best_y_:.6f}")

if opt.best_y_ < best_before:
    print(f"\n New best found!")
    print(f"  Location: {opt.best_x_}")
    print(f"  Value: {opt.best_y_:.6f}")
else:
    print(f"\n Best unchanged")
```

```
Iteration 0: New best f(x) = 1.606003
Best before: 16.329192
Best after: 1.606003
```

```
New best found!
Location: [1.13866324 1.1705867 ]
Value: 1.606003
```

Loop Termination Conditions:

The optimization continues until:

1. `len(y_) >= max_iter` (reached evaluation budget), OR
2. `elapsed_time >= max_time` (reached time limit)

56. Complete Optimization Example

Now let's run a complete optimization to see all steps in action:

```
# Create fresh optimizer
opt_complete = SpotOptim(
    fun=rosenbrock,
    bounds=[(-2, 2), (-2, 2)],
    n_initial=8,
    max_iter=25,
    acquisition='ei',
    verbose=False, # Set to False for cleaner output
    seed=42
)

# Run optimization
result = opt_complete.optimize()

print(f"\nOptimization Result:")
print(f"{'='*70}")
print(f"Best point found: {result.x}")
print(f"Best value: {result.fun:.6f}")
print(f"True optimum: [1.0, 1.0]")
print(f"True minimum: 0.0")
print(f"Gap to optimum: {result.fun:.6f}")
print(f"\nFunction evaluations: {result.nfev}")
print(f"Sequential iterations: {result.nit}")
print(f"Success: {result.success}")
print(f"Message: {result.message}")
```

Optimization Result:

```
=====
Best point found: [1.08776853 1.15560806]
Best value: 0.084058
True optimum: [1.0, 1.0]
True minimum: 0.0
Gap to optimum: 0.084058
```

56. Complete Optimization Example

Function evaluations: 25
Sequential iterations: 17
Success: True
Message: Optimization terminated: maximum evaluations (25) reached

57. Noisy Functions with Repeats

When dealing with noisy objective functions, SpotOptim can evaluate each point multiple times and track statistics.

57.1. Configuration for Noisy Functions

```
NOISE_STD = 10.0
print("\n" + "="*70)
print("NOISY FUNCTION OPTIMIZATION")
print("="*70)
print(f"\nConfiguration for noisy optimization with noise std = {NOISE_STD}:")

# Wrapper to add noise
def rosenbrock_noisy_wrapper(X):
    return rosenbrock_noisy(X, noise_std=NOISE_STD)

opt_noisy = SpotOptim(
    fun=rosenbrock_noisy_wrapper,
    bounds=[(-2, 2), (-2, 2)],
    n_initial=6,
    max_iter=20,
    repeats_initial=3,      # Evaluate each initial point 3 times
    repeats_surrogate=2,    # Evaluate each sequential point 2 times
    verbose=False,
    seed=42
)

print("Configuration for noisy optimization:")
print(f"  repeats_initial: {opt_noisy.repeats_initial}")
print(f"  repeats_surrogate: {opt_noisy.repeats_surrogate}")
print(f"  noise: {opt_noisy.noise}")
```

57. Noisy Functions with Repeats

NOISY FUNCTION OPTIMIZATION

```
Configuration for noisy optimization with noise std = 10.0:
Configuration for noisy optimization:
  repeats_initial: 3
  repeats_surrogate: 2
  noise: True
```

57.2. Noisy Optimization Workflow Differences

57.2.1. Modified Initial Design

With `repeats_initial > 1`:

```
result_noisy = opt_noisy.optimize()

print(f"\nInitial design with repeats:")
print(f"  n_initial = {opt_noisy.n_initial}")
print(f"  repeats_initial = {opt_noisy.repeats_initial}")
print(f"  Total initial evaluations: {opt_noisy.n_initial * opt_noisy.repeats_initial}")

print(f"\nStatistics tracked:")
print(f"  mean_X shape: {opt_noisy.mean_X.shape} (unique points)")
print(f"  mean_y shape: {opt_noisy.mean_y.shape} (mean values)")
print(f"  var_y shape: {opt_noisy.var_y.shape} (variances)")

print(f"\nExample statistics for first point:")
idx = 0
print(f"  Point: {opt_noisy.mean_X[idx]}")
print(f"  Mean value: {opt_noisy.mean_y[idx]:.4f}")
print(f"  Variance: {opt_noisy.var_y[idx]:.4f}")
print(f"  Std dev: {np.sqrt(opt_noisy.var_y[idx]):.4f}")
```

```
Initial design with repeats:
  n_initial = 6
  repeats_initial = 3
  Total initial evaluations: 18
```

Statistics tracked:

```
mean_X shape: (7, 2) (unique points)
mean_y shape: (7,) (mean values)
var_y shape: (7,) (variances)
```

Example statistics for first point:

```
Point: [-1.90573195 -1.79824535]
Mean value: 2960.5138
Variance: 68.6148
Std dev: 8.2834
```

Key Differences with Noise:

1. **Repeated Evaluations:** Each point evaluated multiple times
2. **Statistics Tracking:**
 - `mean_X`: Unique evaluation locations
 - `mean_y`: Mean function values at each location
 - `var_y`: Variance of function values
 - `n_eval`: Number of evaluations per location
3. **Surrogate Fitting:** Uses `mean_y` instead of `y_`
4. **Best Selection:** Based on `mean_y` not individual `y_`

57.2.2. Update Statistics Method

```
print(f"After optimization:")
print(f"  Total evaluations: {len(opt_noisy.y_)}")
print(f"  Unique points: {len(opt_noisy.mean_X)}")
print(f"  Average repeats per point: {len(opt_noisy.y_) / len(opt_noisy.mean_X):.2f}")

print(f"\nVariance statistics:")
print(f"  Mean variance: {np.mean(opt_noisy.var_y):.6f}")
print(f"  Max variance: {np.max(opt_noisy.var_y):.6f}")
print(f"  Min variance: {np.min(opt_noisy.var_y):.6f}")
```

After optimization:

```
Total evaluations: 20
Unique points: 7
Average repeats per point: 2.86
```

Variance statistics:

```
Mean variance: 43.630970
Max variance: 94.990756
Min variance: 6.357449
```


58. Optimal Computing Budget Allocation (OCBA)

OCBA intelligently allocates additional evaluations to distinguish between competing solutions.

58.1. OCBA Configuration

```
print("\n" + "="*70)
print("OPTIMAL COMPUTING BUDGET ALLOCATION (OCBA)")
print("="*70)

opt_ocba = SpotOptim(
    fun=rosenbrock_noisy_wrapper,
    bounds=[(-2, 2), (-2, 2)],
    n_initial=8,
    max_iter=30,
    repeats_initial=2,
    repeats_surrogate=1,
    ocba_delta=3,          # Allocate 3 additional evaluations per iteration
    verbose=False,
    seed=42
)

print("OCBA Configuration:")
print(f"  ocba_delta: {opt_ocba.ocba_delta}")
print(f"  Purpose: Intelligently re-evaluate existing points")
print(f"  Benefit: Better distinguish between similar solutions")
```

```
=====
OPTIMAL COMPUTING BUDGET ALLOCATION (OCBA)
=====
OCBA Configuration:
```

58. Optimal Computing Budget Allocation (OCBA)

ocba_delta: 3
Purpose: Intelligently re-evaluate existing points
Benefit: Better distinguish between similar solutions

58.2. OCBA Method

58.2.1. Method: `_apply_ocba()`

```
result_ocba = opt_ocba.optimize()

print(f"\nOCBA applied during optimization")
print(f"  Final total evaluations: {result_ocba.nfev}")
print(f"  Expected without OCBA: ~{opt_ocba.n_initial * opt_ocba.repeats_initial + result_ocba.nfev}")
print(f"  Additional OCBA evaluations: ~{result_ocba.nit * opt_ocba.ocba_delta}")

print(f"\nOCBA intelligently allocated extra evaluations to:")
print(f"  - Current best candidate (confirm it's truly best)")
print(f"  - Close competitors (distinguish between similar solutions)")
print(f"  - High-variance points (reduce uncertainty)")
```

```
OCBA applied during optimization
  Final total evaluations: 30
  Expected without OCBA: ~27
  Additional OCBA evaluations: ~33
```

```
OCBA intelligently allocated extra evaluations to:
  - Current best candidate (confirm it's truly best)
  - Close competitors (distinguish between similar solutions)
  - High-variance points (reduce uncertainty)
```

OCBA Algorithm:

1. Identify best solution (lowest mean value)
2. Calculate allocation ratios based on:
 - Distance to best solution
 - Variance of each solution
3. Allocate `ocba_delta` additional evaluations
4. Returns points to re-evaluate
5. Requires: 3 points with variance > 0

58.2. OCBA Method

OCBA Activation Conditions: - `noise = True` (repeats > 1) - `ocba_delta > 0` -
At least 3 design points exist - All points have variance > 0

59. Handling Function Evaluation Failures

SpotOptim robustly handles functions that occasionally fail (return NaN/inf).

59.1. Example with Failures

```
opt_failures = SpotOptim(  
    fun=rosenbrock_with_failures,  
    bounds=[(-2, 2), (-2, 2)],  
    n_initial=12,  
    max_iter=35,  
    verbose=False,  
    seed=42  
)  
  
result_failures = opt_failures.optimize()  
  
print(f"\nOptimization with ~15% random failure rate:")  
print(f"  Function evaluations: {result_failures.nfev}")  
print(f"  Sequential iterations: {result_failures.nit}")  
print(f"  Success: {result_failures.success}")  
  
# Count how many values are non-finite in raw evaluations  
n_total = len(opt_failures.y_)  
n_finite = np.sum(np.isfinite(opt_failures.y_))  
print(f"\nEvaluation statistics:")  
print(f"  Total evaluations: {n_total}")  
print(f"  Finite values: {n_finite}")  
print(f"  Note: Failed evaluations handled transparently")
```

```
Optimization with ~15% random failure rate:  
  Function evaluations: 35
```

59. Handling Function Evaluation Failures

Sequential iterations: 27
Success: True

Evaluation statistics:
Total evaluations: 35
Finite values: 35
Note: Failed evaluations handled transparently

59.2. Failure Handling in Initial Design

59.2.1. Method: `_handle_NA_initial_design()`

```
print("Initial design failure handling:")
print("  1. Identify NaN/inf values")
print("  2. Remove invalid points entirely")
print("  3. Continue with valid points only")
print("  4. No penalties applied")
print("  5. Require at least 1 valid point")

print("\nRationale:")
print("  - Initial design should be clean")
print("  - Invalid regions identified naturally")
print("  - Surrogate trained on good data only")
```

Initial design failure handling:

1. Identify NaN/inf values
2. Remove invalid points entirely
3. Continue with valid points only
4. No penalties applied
5. Require at least 1 valid point

Rationale:

- Initial design should be clean
- Invalid regions identified naturally
- Surrogate trained on good data only

59.3. Failure Handling in Sequential Phase

59.3.1. Method: `_handle_NA_new_points()`

```
print("Sequential phase failure handling:")
print("  1. Apply penalty to NaN/inf values")
print("    - Penalty = max(history) + 3*std(history)")
print("    - Add random noise to avoid duplicates")
print("  2. Remove remaining invalid values")
print("  3. Skip iteration if all evaluations failed")
print("  4. Continue if any valid evaluations")

print("\nPenalty approach benefits:")
print("  Preserves optimization history")
print("  Surrogate learns to avoid bad regions")
print("  Better exploration-exploitation balance")
print("  More robust convergence")
```

Sequential phase failure handling:

1. Apply penalty to NaN/inf values
 - Penalty = max(history) + 3*std(history)
 - Add random noise to avoid duplicates
2. Remove remaining invalid values
3. Skip iteration if all evaluations failed
4. Continue if any valid evaluations

Penalty approach benefits:

- Preserves optimization history
- Surrogate learns to avoid bad regions
- Better exploration-exploitation balance
- More robust convergence

59.4. Penalty Application

59.4.1. Method: `_apply_penalty_NA()`

Let's demonstrate penalty calculation:

59. Handling Function Evaluation Failures

```
# Simulate historical values
y_history_sim = np.array([10.0, 15.0, 8.0, 12.0, 20.0, 9.0])
y_new_sim = np.array([7.0, np.nan, 11.0, np.inf])

print("Historical values:", y_history_sim)
print("New evaluations:", y_new_sim)

# Apply penalty
y_repaired = opt_failures._apply_penalty_NA(
    y_new_sim,
    y_history=y_history_sim,
    penalty_value=None, # Compute adaptively
    sd=0.1
)

print(f"\nAfter penalty application:", y_repaired)
print(f"All finite: {np.all(np.isfinite(y_repaired))}")

# Show penalty calculation
max_hist = np.max(y_history_sim)
std_hist = np.std(y_history_sim, ddof=1)
penalty_base = max_hist + 3 * std_hist

print(f"\nPenalty calculation:")
print(f"  max(history) = {max_hist:.2f}")
print(f"  std(history) = {std_hist:.2f}")
print(f"  Base penalty = {max_hist:.2f} + 3×{std_hist:.2f} = {penalty_base:.2f}")
print(f"  Actual penalty = {penalty_base:.2f} + noise")
```

Historical values: [10. 15. 8. 12. 20. 9.]

New evaluations: [7. nan 11. inf]

After penalty application: [7. 33.62137308 11. 33.5370057]

All finite: True

Penalty calculation:

max(history) = 20.00

std(history) = 4.50

Base penalty = 20.00 + 3×4.50 = 33.51

Actual penalty = 33.51 + noise

60. Complete Method Summary

60.1. Methods Called During `optimize()`

60.1.1. Preparation Phase

1. `get_initial_design()` - Generate/process initial sample points
2. `_curate_initial_design()` - Remove duplicates, handle repeats
3. `_evaluate_function()` - Evaluate objective function
4. `_handle_NA_initial_design()` - Remove NaN/inf from initial design
5. `_check_size_initial_design()` - Validate sufficient points
6. `_init_storage()` - Initialize storage (`X_`, `y_`, `n_iter_`)
7. `update_stats()` - Compute mean/variance for noisy functions
8. `_init_tensorboard()` - Log initial design to TensorBoard (if enabled)
9. `_get_best_xy_initial_design()` - Identify initial best

60.1.2. Sequential Optimization Loop (each iteration)

10. `_fit_scheduler()` - Select data and fit surrogate based on noise handling
 - Internally calls `_transform_X()` - Transform to internal scale
 - Internally calls `_fit_surrogate()` - Fit Gaussian Process to data
11. `_apply_ocba()` - OCBA allocation (if enabled)
12. `_suggest_next_point()` - Optimize acquisition function
 - Internally calls `_acquisition_function()`
 - Internally calls `_predict_with_uncertainty()`
13. `_update_repeats_infill_points()` - Repeat suggested point for noisy functions
14. `_evaluate_function()` - Evaluate at suggested point(s)
15. `_handle_NA_new_points()` - Handle failures with penalties
 - Internally calls `_apply_penalty_NA()`
 - Internally calls `_remove_nan()`
16. `_update_success_rate()` - Update success tracking
17. `_update_storage()` - Append new evaluations to storage
 - Internally calls `_inverse_transform_X()` - Convert back to original scale

60. Complete Method Summary

18. `update_stats()` - Update mean/variance statistics
19. `_write_tensorboard_hparams()` - Log hyperparameters (if enabled)
20. `_write_tensorboard_scalars()` - Log scalar metrics (if enabled)
21. `_update_best_main_loop()` - Update best solution

60.1.3. Finalization

22. `to_all_dim()` - Expand to full dimensions (if dimensionality reduction used)
23. `_determine_termination()` - Determine termination reason
24. `_close_tensorboard_writer()` - Close logging (if enabled)
25. `_map_to_factor_values()` - Convert factors back to strings
26. Return `OptimizeResult` object

60.2. Helper Methods Used

- `_generate_initial_design()` - LHS generation
- `_repair_non_numeric()` - Round integer/factor variables
- `_select_new()` - Check for duplicate points
- `_handle_acquisition_failure()` - Fallback strategies
- `to_red_dim()` - Dimension reduction (if enabled)
- `_selection_dispatcher()` - Subset selection for large datasets

61. Termination Conditions

61.1. Method: `_determine_termination()`

```
print("\n" + "="*70)
print("TERMINATION CONDITIONS")
print("="*70)
print("\nMethod: _determine_termination()")
print("-" * 70)

print("Optimization terminates when:")
print("  1. len(y_) >= max_iter (evaluation budget exhausted)")
print("  2. elapsed_time >= max_time (time limit reached)")
print("  3. Whichever comes first")

print(f"\nExample from previous run:")
print(f"  Message: {result_failures.message}")
print(f"  Evaluations: {result_failures.nfev}/{opt_failures.max_iter}")
print(f"  Iterations: {result_failures.nit}")
```

```
=====
TERMINATION CONDITIONS
=====

Method: _determine_termination()
-----

Optimization terminates when:
  1. len(y_) >= max_iter (evaluation budget exhausted)
  2. elapsed_time >= max_time (time limit reached)
  3. Whichever comes first

Example from previous run:
  Message: Optimization terminated: maximum evaluations (35) reached
  Evaluations: 35/35
  Iterations: 27
```


62. Performance Comparison on Noisy Functions

Let's compare standard, noisy, and noisy+OCBA optimization:

```
print("\n" + "="*70)
print("PERFORMANCE COMPARISON")
print("="*70)

# Standard optimization
opt_std = SpotOptim(
    fun=rosenbrock_noisy_wrapper,
    bounds=[(-2, 2), (-2, 2)],
    n_initial=8,
    max_iter=50,
    repeats_initial=1,
    repeats_surrogate=1,
    verbose=False,
    seed=10
)
result_std = opt_std.optimize()

# With repeats (no OCBA)
opt_rep = SpotOptim(
    fun=rosenbrock_noisy_wrapper,
    bounds=[(-2, 2), (-2, 2)],
    n_initial=8,
    max_iter=50,
    repeats_initial=2,
    repeats_surrogate=1,
    ocba_delta=0,
    verbose=False,
    seed=10
)
result_rep = opt_rep.optimize()

# With repeats + OCBA
```

62. Performance Comparison on Noisy Functions

```
opt_ocba_comp = SpotOptim(
    fun=rosenbrock_noisy_wrapper,
    bounds=[(-2, 2), (-2, 2)],
    n_initial=8,
    max_iter=50,
    repeats_initial=1,
    repeats_surrogate=1,
    ocba_delta=1,
    verbose=False,
    seed=10
)
result_ocba_comp = opt_ocba_comp.optimize()

# Evaluate the found optima with the TRUE (noise-free) Rosenbrock function
# to get the actual quality of the solutions
print(f"\n" + "="*70)
print("TRUE FUNCTION VALUE EVALUATION")
print("="*70)
print("\nRe-evaluating found optima with noise-free Rosenbrock function:")
print("-" * 70)

# Evaluate each solution with the TRUE function (no noise)
true_val_std = rosenbrock(result_std.x.reshape(1, -1))[0]
true_val_rep = rosenbrock(result_rep.x.reshape(1, -1))[0]
true_val_ocba = rosenbrock(result_ocba_comp.x.reshape(1, -1))[0]

print(f"\nStandard (no repeats):")
print(f"  Found x: {result_std.x}")
print(f"  Noisy value (from optimization): {result_std.fun:.6f}")
print(f"  TRUE value (noise-free): {true_val_std:.6f}")

print(f"\nWith repeats (no OCBA):")
print(f"  Found x: {result_rep.x}")
print(f"  Noisy value (from optimization): {result_rep.fun:.6f}")
print(f"  TRUE value (noise-free): {true_val_rep:.6f}")

print(f"\nWith repeats + OCBA:")
print(f"  Found x: {result_ocba_comp.x}")
print(f"  Noisy value (from optimization): {result_ocba_comp.fun:.6f}")
print(f"  TRUE value (noise-free): {true_val_ocba:.6f}")

print(f"\n" + "="*70)
print("PERFORMANCE COMPARISON SUMMARY")
print("="*70)
```

```

print(f"\nComparison on noisy Rosenbrock (={NOISE_STD}):")
print(f"{'Method':<30} {'Noisy Value':<15} {'TRUE Value':<15} {'Evaluations':<15}")
print(f"{'-'*75}")
print(f"{'Standard (no repeats)':<30} {result_std.fun:<15.6f} {true_val_std:<15.6f} {result_std.nfev:<15}")
print(f"{'With repeats (no OCBA)':<30} {result_rep.fun:<15.6f} {true_val_rep:<15.6f} {result_rep.nfev:<15}")
print(f"{'With repeats + OCBA':<30} {result_ocba_comp.fun:<15.6f} {true_val_ocba:<15.6f} {result_ocba_comp.nfev:<15}")
print(f"{'-'*75}")
print(f"{'True optimum':<30} {'0.0':<15} {'0.0':<15}")

print("\nKey observations:")
print(" • Noisy values (from optimization) are misleading due to noise")
print(" • TRUE values show actual solution quality")
print(" • Repeats help find better solutions despite noise")
print(" • OCBA focuses evaluations on distinguishing best solutions")
print(" • More evaluations generally lead to better TRUE performance")

```

PERFORMANCE COMPARISON

TRUE FUNCTION VALUE EVALUATION

Re-evaluating found optima with noise-free Rosenbrock function:

Standard (no repeats):

Found x: [-0.02303646 -0.12724366]
Noisy value (from optimization): -14.055814
TRUE value (noise-free): 2.679232

With repeats (no OCBA):

Found x: [-0.18461315 0.02269759]
Noisy value (from optimization): -12.282442
TRUE value (noise-free): 1.416269

With repeats + OCBA:

Found x: [-0.06715147 -0.03562076]
Noisy value (from optimization): -16.585110
TRUE value (noise-free): 1.299855

62. Performance Comparison on Noisy Functions

PERFORMANCE COMPARISON SUMMARY

=====

Comparison on noisy Rosenbrock (=10.0):

Method	Noisy Value	TRUE Value	Evaluations

Standard (no repeats)	-14.055814	2.679232	50
With repeats (no OCBA)	-12.282442	1.416269	50
With repeats + OCBA	-16.585110	1.299855	50

True optimum	0.0	0.0	

Key observations:

- Noisy values (from optimization) are misleading due to noise
- TRUE values show actual solution quality
- Repeats help find better solutions despite noise
- OCBA focuses evaluations on distinguishing best solutions
- More evaluations generally lead to better TRUE performance

63. Summary

63.1. Complete Workflow Diagram

SPOTOPTIM WORKFLOW

INITIALIZATION PHASE

```
get_initial_design()
    Generate LHS or process user design

_curate_initial_design()
    Remove duplicates, add repeats

_evaluate_function()
    Evaluate objective function

_handle_NA_initial_design()
    Remove NaN/inf points

_check_size_initial_design()
    Validate sufficient points

_init_storage()
    Initialize X_, y_, n_iter_

update_stats()
    Compute mean/variance (if noise)

_init_tensorboard() [if enabled]
    Log initial design to TensorBoard

_get_best_xy_initial_design()
    Identify initial best
```

63. Summary

SEQUENTIAL OPTIMIZATION LOOP (until max_iter or max_time)

```
_fit_scheduler()
    _transform_X() - Transform to internal scale
    _fit_surrogate() - Fit GP to current data

_apply_ocba() [if enabled]
    Allocate additional evaluations

_suggest_next_point()
    _acquisition_function()
    _predict_with_uncertainty()
    Optimize to find next point

_update_repeats_infill_points()
    Repeat point if repeats_surrogate > 1

_evaluate_function()
    Evaluate at suggested point(s)

_handle_NA_new_points()
    _apply_penalty_NA()
    _remove_nan()

_update_success_rate()
    Track evaluation success

_update_storage()
    Append new evaluations (with scale conversion)

update_stats()
    Update mean/variance

_write_tensorboard_hparams() [if enabled]
    Log hyperparameters

_write_tensorboard_scalars() [if enabled]
    Log scalar metrics

_update_best_main_loop()
    Update best if improved
```

FINALIZATION

```

to_all_dim() [if dimensionality reduction used]
    Expand to full dimensions

_determine_termination()
    Set termination message

_close_tensorboard_writer() [if enabled]
    Close TensorBoard logging

_map_to_factor_values() [if factors used]
    Convert factors back to strings

Return OptimizeResult

```

63.2. Key Concepts

63.2.1. 1. Initial Design

- **Latin Hypercube Sampling:** Space-filling design for efficient exploration
- **Curation:** Remove duplicates from integer/factor rounding
- **Failure Handling:** Remove invalid points, no penalties

63.2.2. 2. Surrogate Model

- **Default:** Gaussian Process with Matérn kernel
- **Provides:** Mean $\hat{y}(x)$ and uncertainty $\hat{\sigma}(x)$ predictions
- **Purpose:** Learn function landscape with limited evaluations

63.2.3. 3. Acquisition Function

- **EI:** Expected Improvement (default) - best balance
- **PI:** Probability of Improvement - more exploitative
- **Mean:** Pure exploitation of surrogate predictions

63.2.4. 4. Noise Handling

- **Repeats:** Evaluate each point multiple times
- **Statistics:** Track mean and variance
- **OCBA:** Intelligently allocate additional evaluations

63.2.5. 5. Failure Handling

- **Initial Phase:** Remove invalid points
- **Sequential Phase:** Apply adaptive penalties with noise
- **Robustness:** Continue optimization despite failures

63.3. Best Practices

1. **For deterministic functions:**
 - Use default settings (no repeats)
 - Acquisition = 'ei'
 - Focus on `n_initial` and `max_iter`
2. **For noisy functions:**
 - Set `repeats_initial` = 2
 - Set `repeats_surrogate` = 1
 - Consider OCBA with `ocba_delta` = 2
3. **For unreliable functions:**
 - SpotOptim handles failures automatically
 - No special configuration needed
 - Penalties guide search away from bad regions
4. **For expensive functions:**
 - Increase `n_initial` (better initial model)
 - Use 'ei' acquisition (best sample efficiency)
 - Consider `max_time` limit

63.4. Conclusion

SpotOptim provides a robust and flexible framework for surrogate-based optimization with:

Efficient space-filling initial designs (LHS)
Powerful Gaussian Process surrogate models
Smart acquisition functions (EI, PI, Mean)
Automatic noise handling with statistics
Intelligent budget allocation (OCBA)
Robust failure handling
Comprehensive progress tracking

The modular design allows easy customization while maintaining robust defaults for most use cases.

63.5. Jupyter Notebook

i Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

Part XI.

Optimization

64. Aircraft Wing Weight Example

i Note

- This section is based on chapter 1.3 “A ten-variable weight function” in Forrester, Sóbester, and Keane (2008).
- The following Python packages are imported:

```
import math
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.axes_grid1 import ImageGrid
```

64.1. AWWWE Equation

- Example from Forrester, Sóbester, and Keane (2008)
- Understand the **weight** of an unpainted light aircraft wing as a function of nine design and operational parameters:

$$W = 0.036 S_W^{0.758} \times W_{fw}^{0.0035} \left(\frac{A}{\cos^2 \Lambda} \right)^{0.6} \times q^{0.006} \times \lambda^{0.04} \\ \times \left(\frac{100 R_{tc}}{\cos \Lambda} \right)^{-0.3} \times (N_z W_{dg})^{0.49}$$

64.2. AWWWE Parameters and Equations (Part 1)

Table 64.1.: Aircraft Wing Weight Parameters

Symbol	Parameter	Baseline	Mini- mum	Maxi- mum
S_W	Wing area (ft^2)	174	150	200
W_{fw}	Weight of fuel in wing (lb)	252	220	300

64. Aircraft Wing Weight Example

Symbol	Parameter	Baseline	Minimum	Maximum
A	Aspect ratio	7.52	6	10
Λ	Quarter-chord sweep (deg)	0	-10	10
q	Dynamic pressure at cruise (lb/ft^2)	34	16	45
λ	Taper ratio	0.672	0.5	1
R_{tc}	Aerofoil thickness to chord ratio	0.12	0.08	0.18
N_z	Ultimate load factor	3.8	2.5	6
W_{dg}	Flight design gross weight (lb)	2000	1700	2500
W_p	paint weight (lb/ft^2)	0.064	0.025	0.08

The study begins with a baseline Cessna C172 Skyhawk Aircraft as its reference point. It aims to investigate the impact of wing area and fuel weight on the overall weight of the aircraft. Two crucial parameters in this analysis are the aspect ratio (A), defined as the ratio of the wing's length to the average chord (thickness of the airfoil), and the taper ratio (λ), which represents the ratio of the maximum to the minimum thickness of the airfoil or the maximum to minimum chord.

It's important to note that the equation used in this context is not a computer simulation but will be treated as one for the purpose of illustration. This approach involves employing a true mathematical equation, even if it's considered unknown, as a useful tool for generating realistic settings to test the methodology. The functional form of this equation was derived by "calibrating" known physical relationships to curves obtained from existing aircraft data, as referenced in Raymer (2006). Essentially, it acts as a surrogate for actual measurements of aircraft weight.

Examining the mathematical properties of the AWWE (Aircraft Weight With Wing Area and Fuel Weight Equation), it is evident that the response is highly nonlinear concerning its inputs. While it's common to apply the logarithm to simplify equations with complex exponents, even when modeling the logarithm, which transforms powers into slope coefficients and products into sums, the response remains nonlinear due to the presence of trigonometric terms. Given the combination of nonlinearity and high input dimension, simple linear and quadratic response surface approximations are likely to be inadequate for this analysis.

64.3. Goals: Understanding and Optimization

The primary goals of this study revolve around understanding and optimization:

1. **Understanding:** One of the straightforward objectives is to gain a deep understanding of the input-output relationships in this context. Given the global perspective implied by this setting, it becomes evident that a more sophisticated

64.3. Goals: Understanding and Optimization

model is almost necessary. At this stage, let's focus on this specific scenario to establish a clear understanding.

2. **Optimization:** Another application of this analysis could be optimization. There may be an interest in minimizing the weight of the aircraft, but it's likely that there will be constraints in place. For example, the presence of wings with a nonzero area is essential for the aircraft to be capable of flying. In situations involving (constrained) optimization, a global perspective and, consequently, the use of flexible modeling are vital.

The provided Python code serves as a genuine computer implementation that “solves” a mathematical model. It accepts arguments encoded in the unit cube, with defaults used to represent baseline settings, as detailed in the table labeled as Table 64.1. To map values from the interval $[a, b]$ to the interval $[0, 1]$, the following formula can be employed:

$$y = f(x) = \frac{x - a}{b - a}. \quad (64.1)$$

To reverse this mapping and obtain the original values, the formula

$$g(y) = a + (b - a)y \quad (64.2)$$

can be used. The function `wingwt()` expects inputs from the unit cube, which are then transformed back to their original scales using Equation 64.2. The function is defined as follows:

```
def wingwt(Sw=0.48, Wfw=0.4, A=0.38, L=0.5, q=0.62, l=0.344, Rtc=0.4, Nz=0.37, Wdg=0.38):
    # put coded inputs back on natural scale
    Sw = Sw * (200 - 150) + 150
    Wfw = Wfw * (300 - 220) + 220
    A = A * (10 - 6) + 6
    L = (L * (10 - (-10)) - 10) * np.pi/180
    q = q * (45 - 16) + 16
    l = l * (1 - 0.5) + 0.5
    Rtc = Rtc * (0.18 - 0.08) + 0.08
    Nz = Nz * (6 - 2.5) + 2.5
    Wdg = Wdg*(2500 - 1700) + 1700
    # calculation on natural scale
    W = 0.036 * Sw**0.758 * Wfw**0.0035 * (A/np.cos(L)**2)**0.6 * q**0.006
    W = W * l**0.04 * (100*Rtc/np.cos(L))**(-0.3) * (Nz*Wdg)**(0.49)
    return(W)
```

64.4. Properties of the Python “Solver”

The compute time required by the “wingwt” solver is extremely short and can be considered trivial in terms of computational resources. The approximation error is exceptionally small, effectively approaching machine precision, which indicates the high accuracy of the solver’s results.

To simulate time-consuming evaluations, a deliberate delay is introduced by incorporating a `sleep(3600)` command, which effectively synthesizes a one-hour execution time for a particular evaluation.

Moving on to the AWWE visualization, plotting in two dimensions is considerably simpler than dealing with nine dimensions. To aid in creating visual representations, the code provided below establishes a grid within the unit square to facilitate the generation of sliced visuals. This involves generating a “meshgrid” as outlined in the code.

```
x = np.linspace(0, 1, 3)
y = np.linspace(0, 1, 3)
X, Y = np.meshgrid(x, y)
zp = zip(np.ravel(X), np.ravel(Y))
list(zp)
```

```
[(np.float64(0.0), np.float64(0.0)),
 (np.float64(0.5), np.float64(0.0)),
 (np.float64(1.0), np.float64(0.0)),
 (np.float64(0.0), np.float64(0.5)),
 (np.float64(0.5), np.float64(0.5)),
 (np.float64(1.0), np.float64(0.5)),
 (np.float64(0.0), np.float64(1.0)),
 (np.float64(0.5), np.float64(1.0)),
 (np.float64(1.0), np.float64(1.0))]
```

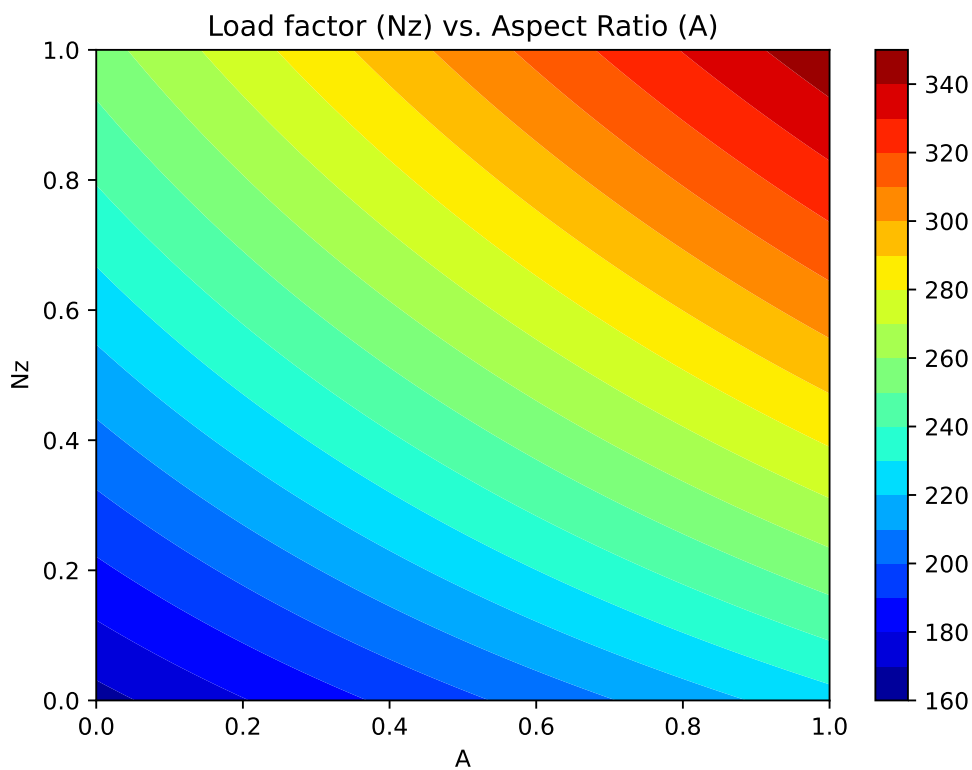
The coding used to transform inputs from natural units is largely a matter of taste, so long as it’s easy to undo for reporting back on original scales

```
%matplotlib inline
# plt.style.use('seaborn-white')
x = np.linspace(0, 1, 100)
y = np.linspace(0, 1, 100)
X, Y = np.meshgrid(x, y)
```


64.5. Plot 1: Load Factor (N_z) and Aspect Ratio (A)

We will vary N_z and A , with other inputs fixed at their baseline values.

```
z = wingwt(A = X, Nz = Y)
fig = plt.figure(figsize=(7., 5.))
plt.contourf(X, Y, z, 20, cmap='jet')
plt.xlabel("A")
plt.ylabel("Nz")
plt.title("Load factor (Nz) vs. Aspect Ratio (A)")
plt.colorbar()
plt.show()
```



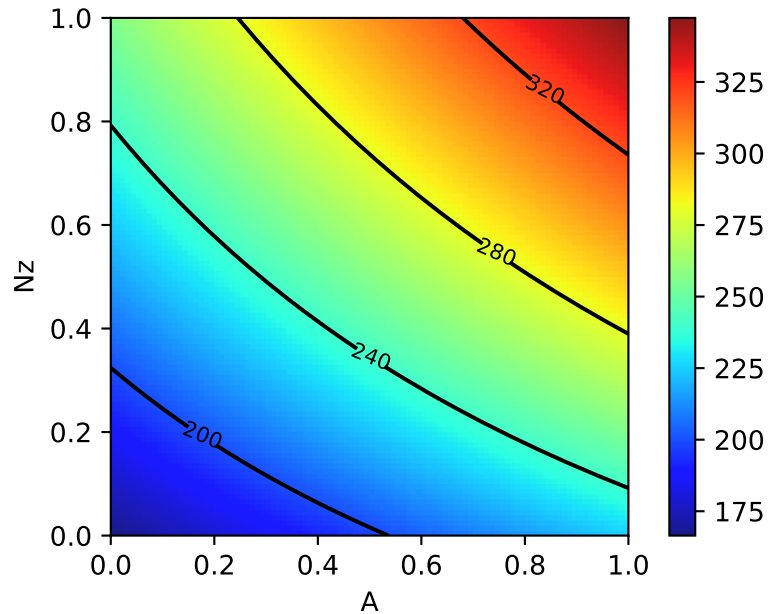
Contour plots can be refined, e.g., by adding explicit contour lines as shown in the following figure.

```
contours = plt.contour(X, Y, z, 4, colors='black')
plt.clabel(contours, inline=True, fontsize=8)
```

64. Aircraft Wing Weight Example

```
plt.xlabel("A")
plt.ylabel("Nz")

plt.imshow(z, extent=[0, 1, 0, 1], origin='lower',
           cmap='jet', alpha=0.9)
plt.colorbar()
```



The interpretation of the AWWE plot can be summarized as follows:

- The figure displays the weight response as a function of two variables, N_z and A , using an image-contour plot.
- The slight curvature observed in the contours suggests an interaction between these two variables.
- Notably, the range of outputs depicted in the figure, spanning from approximately 160 to 320, nearly encompasses the entire range of outputs observed from various input settings within the full 9-dimensional input space.
- The plot indicates that aircraft wings tend to be heavier when the aspect ratios (A) are high.
- This observation aligns with the idea that wings are designed to withstand and accommodate high gravitational forces (g -forces, large N_z), and there may be a compounding effect where larger values of N_z contribute to increased wing weight.

64.6. Plot 2: Taper Ratio and Fuel Weight

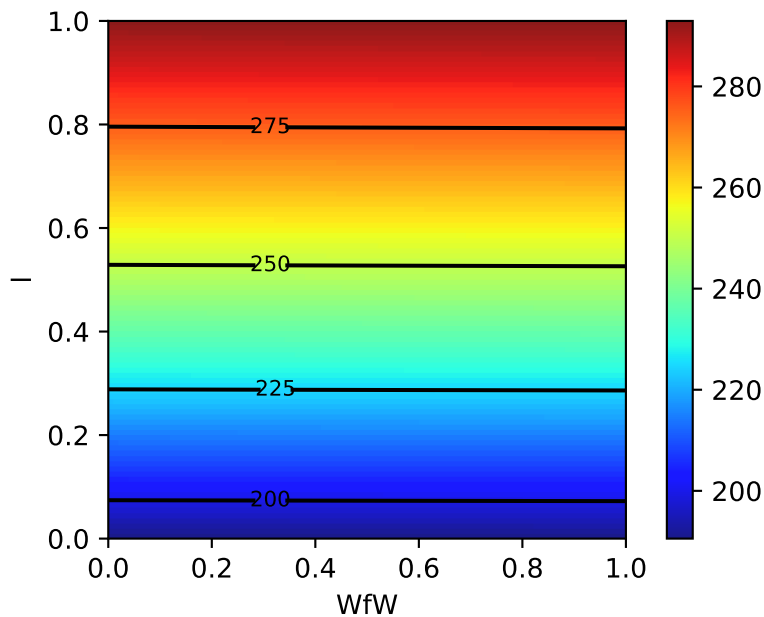
- It's plausible that this phenomenon is related to the design considerations of fighter jets, which cannot have the efficient and lightweight glider-like wings typically found in other types of aircraft.

64.6. Plot 2: Taper Ratio and Fuel Weight

- The same experiment for two other inputs, e.g., taper ratio λ and fuel weight W_{fw}

```
z = wingwt(Wfw = X, Nz = Y)
contours = plt.contour(X, Y, z, 4, colors='black')
plt.clabel(contours, inline=True, fontsize=8)
plt.xlabel("WfW")
plt.ylabel("l")

plt.imshow(z, extent=[0, 1, 0, 1], origin='lower',
           cmap='jet', alpha=0.9)
plt.colorbar();
```



- Interpretation of Taper Ratio (l) and Fuel Weight (W_{fw})
 - Apparently, neither input has much effect on wing weight:

64. Aircraft Wing Weight Example

- * with λ having a marginally greater effect, covering less than 4 percent of the span of weights observed in the $A \times N_z$ plane
- There's no interaction evident in $\lambda \times W_{fw}$

64.7. The Big Picture: Combining all Variables

```
p1 = ["Sw", "Wfw", "A", "L", "q", "l", "Rtc", "Nz", "Wdg"]
```

```
Z = []
Zlab = []
l = len(p1)
# lc = math.comb(l,2)
for i in range(l):
    for j in range(i+1, l):
        # for j in range(l):
            # print(p1[i], p1[j])
            d = {p1[i]: X, p1[j]: Y}
            Z.append(wingwt(**d))
            Zlab.append([p1[i], p1[j]])
```

Now we can generate all 36 combinations, e.g., our first example is combination $p = 19$.

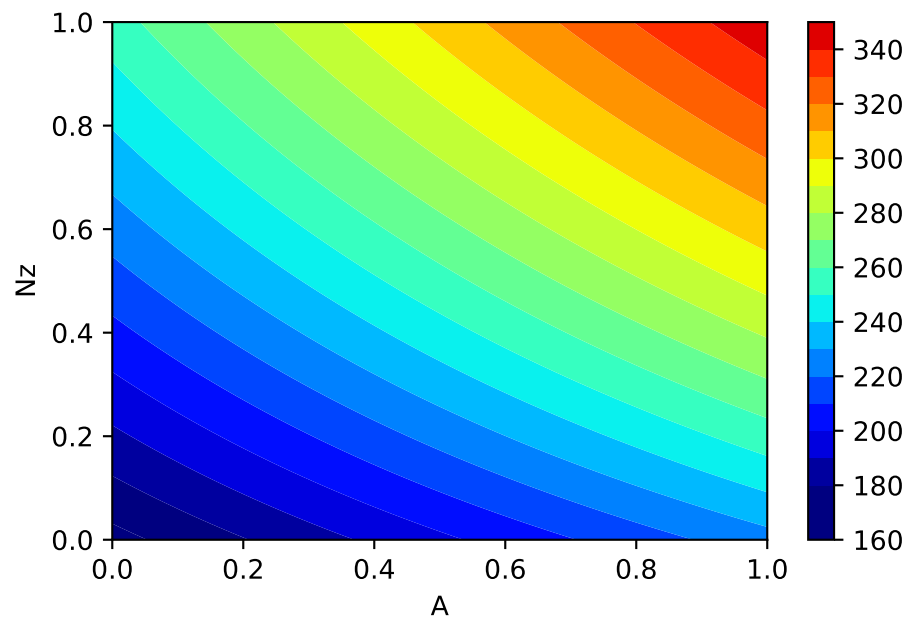
```
p = 19
Zlab[p]
```

```
['A', 'Nz']
```

To help interpret outputs from experiments such as this one—to level the playing field when comparing outputs from other pairs of inputs—code below sets up a color palette that can be re-used from one experiment to the next. We use the arguments `vmin=180` and `vmax=360` to implement comparability

```
plt.contourf(X, Y, Z[p], 20, cmap='jet', vmin=180, vmax=360)
plt.xlabel(Zlab[p][0])
plt.ylabel(Zlab[p][1])
plt.colorbar()
```

64.7. The Big Picture: Combining all Variables



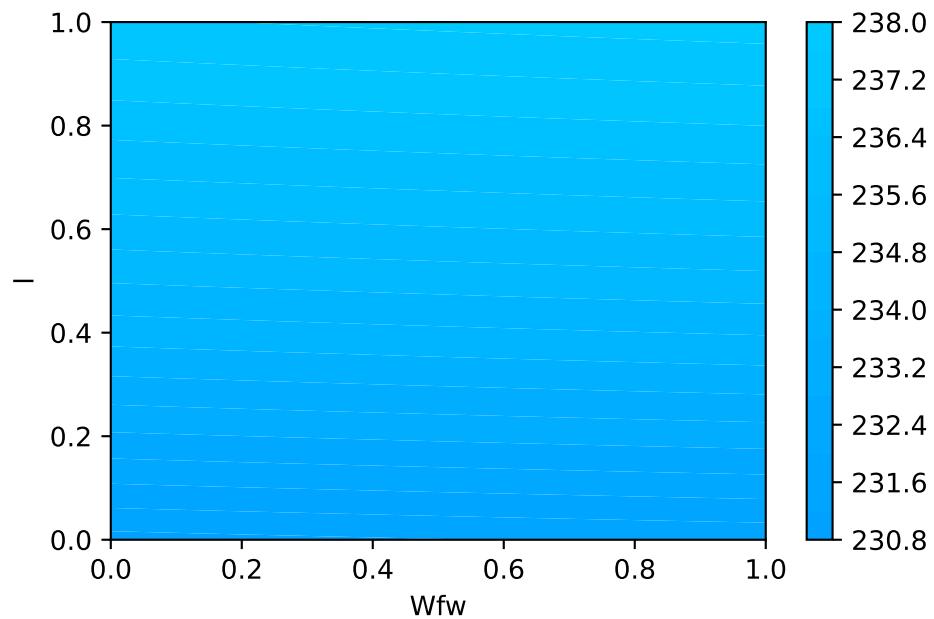
- Let's plot the second example, taper ratio λ and fuel weight W_{fw}
- This is combination 11:

```
p = 11
Zlab[p]
```

```
['Wfw', '1']
```

```
plt.contourf(X, Y, Z[p], 20, cmap='jet', vmin=180, vmax=360)
plt.xlabel(Zlab[p][0])
plt.ylabel(Zlab[p][1])
plt.colorbar()
```

64. Aircraft Wing Weight Example

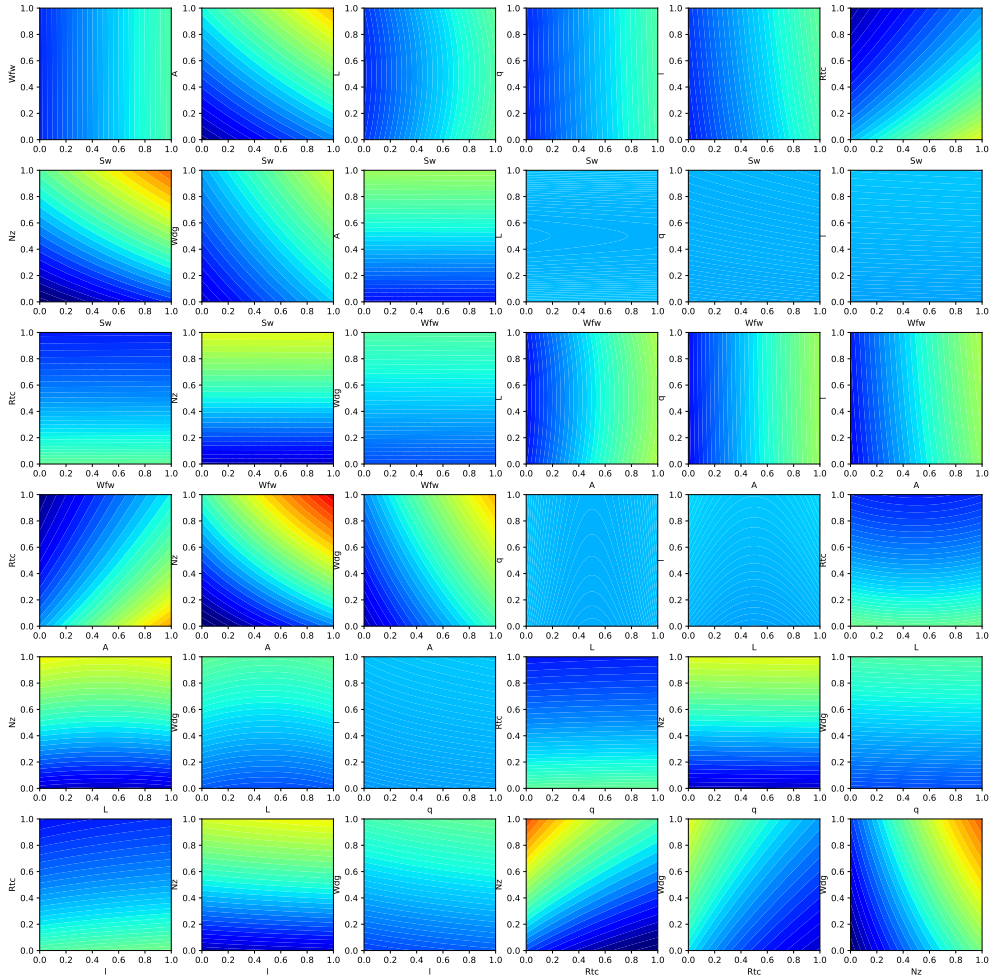


- Using a global colormap indicates that these variables have minor effects on the wing weight.
- Important factors can be detected by visual inspection
- Plotting the Big Picture: we can plot all 36 combinations in one figure.

```
fig = plt.figure(figsize=(20., 20.))
grid = ImageGrid(fig, 111, # similar to subplot(111)
                  nrows_ncols=(6,6), # creates 2x2 grid of axes
                  axes_pad=0.5, # pad between axes in inch.
                  share_all=True,
                  label_mode="all",
                  )

i = 0
for ax, im in zip(grid, Z):
    # Iterating over the grid returns the Axes.
    ax.set_xlabel(Zlab[i][0])
    ax.set_ylabel(Zlab[i][1])
    # ax.set_title(Zlab[i][1] + " vs. " + Zlab[i][0])
    ax.contourf(X, Y, im, 30, cmap = "jet", vmin = 180, vmax = 360)
    i = i + 1

plt.show()
```



64.8. AWE Landscape

- Our Observations

1. The load factor N_z , which determines the magnitude of the maximum aerodynamic load on the wing, is very active and involved in interactions with other variables.
 - Classic example: the interaction of N_z with the aspect ratio A indicates a heavy wing for high aspect ratios and large g -forces
 - This is the reason why highly manoeuvrable fighter jets cannot have very efficient, glider wings)

64. Aircraft Wing Weight Example

2. Aspect ratio A and airfoil thickness to chord ratio R_{tc} have nonlinear interactions.
 3. Most important variables:
 - Ultimate load factor N_z , wing area S_w , and flight design gross weight W_{dg} .
 4. Little impact: dynamic pressure q , taper ratio l , and quarter-chord sweep L .
- Expert Knowledge
 - Aircraft designers know that the overall weight of the aircraft and the wing area must be kept to a minimum
 - the latter usually dictated by constraints such as required stall speed, landing distance, turn rate, etc.

64.9. Summary of the First Experiments

- First, we considered two pairs of inputs, out of 36 total pairs
- Then, the “Big Picture”:
 - For each pair we evaluated `wingwt` 10,000 times
- Doing the same for all pairs would require 360K evaluations:
 - not a reasonable number with a real computer simulation that takes any non-trivial amount of time to evaluate
 - Only 1s per evaluation: > 100 hours
- Many solvers take minutes/hours/days to execute a single run
- And: three-way interactions?
- Consequence: a different strategy is needed

64.10. Exercise

64.10.1. Adding Paint Weight

- Paint weight is not considered.
- Add Paint Weight W_p to formula (the updated formula is shown below) and update the functions and plots in the notebook.

$$W = 0.036 S_w^{0.758} \times W_{fw}^{0.0035} \times \left(\frac{A}{\cos^2 \Lambda} \right)^{0.6} \times q^{0.006} \times \lambda^{0.04} \\ \times \left(\frac{100 R_{tc}}{\cos \Lambda} \right)^{-0.3} \times (N_z W_{dg})^{0.49} + S_w W_p$$

64.11. Jupyter Notebook

i Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

Part XII.

Numerical Methods

65. Simulation and Surrogate Modeling

- We will consider the interplay between
 - mathematical models,
 - numerical approximation,
 - simulation,
 - computer experiments, and
 - field data
- Experimental design will play a key role in our developments, but not in the classical regression and response surface methodology sense
- Challenging real-data/real-simulation examples benefiting from modern surrogate modeling methodology
- We will consider the classical, response surface methodology (RSM) approach, and then move on to more modern approaches
- All approaches are based on surrogates

65.1. Surrogates

- Gathering data is **expensive**, and sometimes getting exactly the data you want is impossible or unethical
- **Surrogate**: substitute for the real thing
- In statistics, draws from predictive equations derived from a fitted model can act as a surrogate for the data-generating mechanism
- Benefits of the surrogate approach:
 - Surrogate could represent a cheaper way to explore relationships, and entertain “what ifs?”
 - Surrogates favor faithful yet pragmatic reproduction of dynamics:
 - * interpretation,
 - * establishing causality, or
 - * identification
 - Many numerical simulators are **deterministic**, whereas field observations are noisy or have measurement error

65.1.1. Costs of Simulation

- Computer simulations are generally cheaper (but not always!) than physical observation
- Some computer simulations can be just as expensive as field experimentation, but computer modeling is regarded as easier because:
 - the experimental apparatus is better understood
 - more aspects may be controlled.

65.1.2. Mathematical Models and Meta-Models

- Use of mathematical models leveraging numerical solvers has been commonplace for some time
- Mathematical models became more complex, requiring more resources to simulate/solve numerically
- Practitioners increasingly relied on **meta-models** built off of limited simulation campaigns

65.1.3. Surrogates = Trained Meta-models

- Data collected via expensive computer evaluations tuned flexible functional forms that could be used in lieu of further simulation to
 - save money or computational resources;
 - cope with an inability to perform future runs (expired licenses, off-line or over-impacted supercomputers)
- Trained meta-models became known as **surrogates**

65.1.4. Computer Experiments

- **Computer experiment**: design, running, and fitting meta-models.
 - Like an ordinary statistical experiment, except the data are generated by computer codes rather than physical or field observations, or surveys
- **Surrogate modeling** is statistical modeling of computer experiments

65.1.5. Limits of Mathematical Modeling

- Mathematical biologists, economists and others had reached the limit of equilibrium-based mathematical modeling with cute closed-form solutions
- **Stochastic simulations replace deterministic solvers** based on FEM, Navier–Stokes or Euler methods
- Agent-based simulation models are used to explore predator-prey (Lotka–Volterra) dynamics, spread of disease, management of inventory or patients in health insurance markets
- Consequence: the distinction between surrogate and statistical model is all but gone

65.1.6. Why Computer Simulations are Necessary

- You can't seed a real community with Ebola and watch what happens
- If there's (real) field data, say on a historical epidemic, further experimentation may be almost entirely limited to the mathematical and computer modeling side
- Classical statistical methods offer little guidance

65.1.7. Simulation Requirements

- Simulation should
 - enable rich **diagnostics** to help criticize that models
 - **understanding** its sensitivity to inputs and other configurations
 - providing the ability to **optimize** and
 - refine both **automatically** and with expert intervention
- And it has to do all that while remaining **computationally tractable**
- One perspective is so-called **response surface methods** (RSMs):
- a poster child from industrial statistics' heyday, well before information technology became a dominant industry

65.2. Applications of Surrogate Models

The four most common usages of surrogate models are:

- **Augmenting Expensive Simulations:** Surrogate models act as a 'curve fit' to approximate the results of expensive simulation codes, enabling predictions without rerunning the primary source. This provides significant speed improvements while maintaining useful accuracy.

- **Calibration of Predictive Codes:** Surrogates bridge the gap between simpler, faster but less accurate models and more accurate, slower models. This multi-fidelity approach allows for improved accuracy without the full computational expense.
- **Handling Noisy or Missing Data:** Surrogates smooth out random or systematic errors in experimental or computational data, filling gaps and revealing overall trends while filtering out extraneous details.
- **Data Mining and Insight Generation:** Surrogates help identify functional relationships between variables and their impact on results. They enable engineers to focus on critical variables and visualize data trends effectively.

65.3. DACE and RSM

Mathematical models implemented in computer codes are used to circumvent the need for expensive field data collection. These models are particularly useful when dealing with highly nonlinear response surfaces, high signal-to-noise ratios (which often involve deterministic evaluations), and a global scope. As a result, a new approach is required in comparison to Response Surface Methodology (RSM), which is discussed in Section 68.1.

With the improvement in computing power and simulation fidelity, researchers gain higher confidence and a better understanding of the dynamics in physical, biological, and social systems. However, the expansion of configuration spaces and increasing input dimensions necessitates more extensive designs. High-performance computing (HPC) allows for thousands of runs, whereas previously only tens were possible. This shift towards larger models and training data presents new computational challenges.

Research questions for DACE (Design and Analysis of Computer Experiments) include how to design computer experiments that make efficient use of computation and how to meta-model computer codes to save on simulation effort. The choice of surrogate model for computer codes significantly impacts the optimal experiment design, and the preferred model-design pairs can vary depending on the specific goal.

The combination of computer simulation, design, and modeling with field data from similar real-world experiments introduces a new category of computer model tuning problems. The ultimate goal is to automate these processes to the greatest extent possible, allowing for the deployment of HPC with minimal human intervention.

One of the remaining differences between RSM and DACE lies in how they handle noise. DACE employs replication, a technique that would not be used in a deterministic setting, to separate signal from noise. Traditional RSM is best suited for situations where a substantial proportion of the variability in the data is due to noise, and where the acquisition of data values can be severely limited. Consequently, RSM is better suited for a different class of problems, aligning with its intended purposes.

Two very good texts on computer experiments and surrogate modeling are Santner, Williams, and Notz (2003) and Forrester, Sóbester, and Keane (2008). The former is the canonical reference in the statistics literature and the latter is perhaps more popular in engineering.

Example 65.1 (Example: DACE and RSM). Imagine you are a chemical engineer tasked with optimizing a chemical process to maximize yield. You can control temperature and pressure, but repeated experiments show variability in yield due to inconsistencies in raw materials.

- Using RSM: You would use RSM to design a series of experiments varying temperature and pressure. You would then fit a response surface (a mathematical model) to the data, helping you understand how changes in temperature and pressure affect yield. Using this model, you can identify optimal conditions for maximizing yield despite the noise.
- Using DACE: If instead you use a computational model to simulate the chemical process and want to account for numerical noise or uncertainty in model parameters, you might use DACE. You would run simulations at different conditions, possibly repeating them to assess variability and build a surrogate model that accurately predicts yields, which can be optimized to find the best conditions.

65.3.1. Noise Handling in RSM and DACE

Noise in RSM: In experimental settings, noise often arises due to variability in experimental conditions, measurement errors, or other uncontrollable factors. This noise can significantly affect the response variable, Y . Replication is a standard procedure for handling noise in RSM. In the context of computer experiments, noise might not be present in the traditional sense since simulations can be deterministic. However, variability can arise from uncertainty in input parameters or model inaccuracies. DACE predominantly utilizes advanced interpolation to construct accurate models of deterministic data, sometimes considering statistical noise modeling if needed.

65.4. Updating a Surrogate Model

A surrogate model is updated by incorporating new data points, known as infill points, into the model to improve its accuracy and predictive capabilities. This process is iterative and involves the following steps:

- Identify Regions of Interest: The surrogate model is analyzed to determine areas where it is inaccurate or where further exploration is needed. This could be regions with high uncertainty or areas where the model predicts promising results (e.g., potential optima).

65. Simulation and Surrogate Modeling

- **Select Infill Points:** Infill points are new data points chosen based on specific criteria, such as:
- **Exploitation:** Sampling near predicted optima to refine the solution. **Exploration:** Sampling in regions of high uncertainty to improve the model globally. **Balanced Approach:** Combining exploitation and exploration to ensure both local and global improvements.
- **Evaluate the True Function:** The true function (e.g., a simulation or experiment) is evaluated at the selected infill points to obtain their corresponding outputs.
- **Update the Surrogate Model:** The surrogate model is retrained or updated using the new data, including the infill points, to improve its accuracy.
- **Repeat:** The process is repeated until the model meets predefined accuracy criteria or the computational budget is exhausted.

Definition 65.1 (Infill Points). Infill points are strategically chosen new data points added to the surrogate model. They are selected to:

- Reduce uncertainty in the model.
- Improve predictions in regions of interest.
- Enhance the model's ability to identify optima or trends.

The selection of infill points is often guided by infill criteria, such as:

- **Expected Improvement (EI):** Maximizing the expected improvement over the current best solution.
- **Uncertainty Reduction:** Sampling where the model's predictions have high variance.
- **Probability of Improvement (PI):** Sampling where the probability of improving the current best solution is highest.

The iterative infill-points updating process ensures that the surrogate model becomes increasingly accurate and useful for optimization or decision-making tasks.

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

66. Sampling Plans

i Note

- This section is based on chapter 1 in Forrester, Sóbester, and Keane (2008).
- The following Python packages are imported:

```
import numpy as np
import pandas as pd
import numpy as np
from typing import Tuple, Optional
import matplotlib.pyplot as plt
from spotpython.utils.sampling import rlh
from spotpython.utils.effects import screening_plot, screeningplan
from spotpython.fun.objectivefunctions import Analytical
from spotpython.utils.sampling import (fullfactorial, bestlh,
    jd, mm, mmphi, mmsort, perturb, mmlhs, phisort, mmphi_intensive)
from spotpython.design.poor import Poor
from spotpython.design.clustered import Clustered
from spotpython.design.sobol import Sobol
from spotpython.design.spacefilling import SpaceFilling
from spotpython.design.random import Random
from spotpython.design.grid import Grid
```

66.1. Ideas and Concepts

Definition 66.1 (Sampling Plan). In the context of computer experiments, the term sampling plan refers to the set of input values, say X , at which the computer code is evaluated.

The goal of a sampling plan is to efficiently explore the input space to understand the behavior of the computer code and build a surrogate model that accurately represents the code's behavior. Traditionally, Response Surface Methodology (RSM) has been used to design sampling plans for computer experiments. These sampling plans are

based on procedures that generate points by means of a rectangular grid or a factorial design.

However, more recently, Design and Analysis of Computer Experiments (DACE) has emerged as a more flexible and powerful approach for designing sampling plans.

Engineering design often requires the construction of a surrogate model $\hat{f}$ to approximate the expensive response of a black-box function f . The function $f(x)$ represents a continuous metric (e.g., quality, cost, or performance) defined over a design space $D \subset \mathbb{R}^k$, where x is a k -dimensional vector of design variables. Since evaluating f is costly, only a sparse set of samples is used to construct $\hat{f}$, which can then provide inexpensive predictions for any $x \in D$.

The process involves:

- Sampling discrete observations:
- Using these samples to construct an approximation $\hat{f}$.
- Ensuring the surrogate model is well-posed, meaning it is mathematically valid and can generalize predictions effectively.

A sampling plan

$$X = \{x^{(i)} \in D | i = 1, \dots, n\}$$

determines the spatial arrangement of observations. While some models require a minimum number of data points n , once this threshold is met, a surrogate model can be constructed to approximate f efficiently.

A well-posed model does not always perform well because its ability to generalize depends heavily on the sampling plan used to collect data. If the sampling plan is poorly designed, the model may fail to capture critical behaviors in the design space. For example:

- Extreme Sampling: Measuring performance only at the extreme values of parameters may miss important behaviors in the center of the design space, leading to incomplete understanding.
- Uneven Sampling: Concentrating samples in certain regions while neglecting others forces the model to extrapolate over unsampled areas, potentially resulting in inaccurate or misleading predictions. Additionally, in some cases, the data may come from external sources or be limited in scope, leaving little control over the sampling plan. This can further restrict the model's ability to generalize effectively.

66.1.1. The ‘Curse of Dimensionality’ and How to Avoid It

The “curse of dimensionality” refers to the exponential increase in computational complexity and data requirements as the number of dimensions (variables) in a problem grows. For a one-dimensional space, sampling n locations may suffice for accurate predictions. In high-dimensional spaces, the amount of data needed to maintain the same level of accuracy or coverage increases dramatically. For example, if a one-dimensional space requires n samples for a certain accuracy, a k -dimensional space would require n^k samples. This makes tasks like optimization, sampling, and modeling computationally expensive and often impractical in high-dimensional settings.

Example 66.1 (Example: Curse of Dimensionality). Consider a simple example where we want to model the cost of a car tire based on its wheel diameter. If we have one variable (wheel diameter), we might need 10 simulations to get a good estimate of the cost. Now, if we add 8 more variables (e.g., tread pattern, rubber type, etc.), the number of simulations required increases to 10^8 (10 million). This is because the number of combinations of design variables grows exponentially with the number of dimensions. This means that the computational budget required to evaluate all combinations of design variables becomes infeasible. In this case, it would take 11,416 years to complete the simulations, making it impractical to explore the design space fully.

66.1.2. Physical versus Computational Experiments

Physical experiments are prone to experimental errors from three main sources:

- Human error: Mistakes made by the experimenter.
- Random error: Measurement inaccuracies that vary unpredictably.
- Systematic error: Consistent bias due to flaws in the experimental setup.

The key distinction is repeatability: systematic errors remain constant across repetitions, while random errors vary.

Computational experiments, on the other hand, are deterministic and free from random errors. However, they are still affected by:

- Human error: Bugs in code or incorrect boundary conditions.
- Systematic error: Biases from model simplifications (e.g., inviscid flow approximations) or finite resolution (e.g., insufficient mesh resolution).

The term “noise” is used differently in physical and computational contexts. In physical experiments, it refers to random errors, while in computational experiments, it often refers to systematic errors.

Understanding these differences is crucial for designing experiments and applying techniques like Gaussian process-based approximations. For physical experiments, replication mitigates random errors, but this is unnecessary for deterministic computational experiments.

66.1.3. Designing Preliminary Experiments (Screening)

Minimizing the number of design variables $x_1, x_2, \dots, x_k$ is crucial before modeling the objective function f . This process, called screening, aims to reduce dimensionality without compromising the analysis. If f is at least once differentiable over the design domain D , the partial derivative $\frac{\partial f}{\partial x_i}$ can be used to classify variables:

- Negligible Variables: If $\frac{\partial f}{\partial x_i} = 0, \forall x \in D$, the variable x_i can be safely neglected.
- Linear Additive Variables: If $\frac{\partial f}{\partial x_i} = \text{constant} \neq 0, \forall x \in D$, the effect of x_i is linear and additive.
- Nonlinear Variables: If $\frac{\partial f}{\partial x_i} = g(x_i), \forall x \in D$, where $g(x_i)$ is a non-constant function, f is nonlinear in x_i .
- Interactive Nonlinear Variables: If $\frac{\partial f}{\partial x_i} = g(x_i, x_j, \dots), \forall x \in D$, where $g(x_i, x_j, \dots)$ is a function involving interactions with other variables, f is nonlinear in x_i and interacts with x_j .

Measuring $\frac{\partial f}{\partial x_i}$ across the entire design space is often infeasible due to limited budgets. The percentage of time allocated to screening depends on the problem: If many variables are expected to be inactive, thorough screening can significantly improve model accuracy by reducing dimensionality. If most variables are believed to impact the objective, focus should shift to modeling instead. Screening is a trade-off between computational cost and model accuracy, and its effectiveness depends on the specific problem context.

66.1.3.1. Estimating the Distribution of Elementary Effects

In order to simplify the presentation of what follows, we make, without loss of generality, the assumption that the design space $D = [0, 1]^k$; that is, we normalize all variables into the unit cube. We shall adhere to this convention for the rest of the book and strongly urge the reader to do likewise when implementing any algorithms described here, as this step not only yields clearer mathematics in some cases but also safeguards against scaling issues.

Before proceeding with the description of the Morris algorithm, we need to define an important statistical concept. Let us restrict our design space D to a k -dimensional, p -level full factorial grid, that is,

$$x_i \in \{0, \frac{1}{p-1}, \frac{2}{p-1}, \dots, 1\}, \quad \text{for } i = 1, \dots, k.$$

Definition 66.2 (Elementary Effect). For a given baseline value $x \in D$, let $d_i(x)$ denote the elementary effect of x_i , where:

$$d_i(x) = \frac{f(x_1, \dots, x_i + \Delta, \dots, x_k) - f(x_1, \dots, x_i - \Delta, \dots, x_k)}{2\Delta}, \quad i = 1, \dots, k, \quad (66.1)$$

where Δ is the step size, which is defined as the distance between two adjacent levels in the grid. In other words, we have:

with

$$\Delta = \frac{\xi}{p-1}, \quad \xi \in \mathbb{N}^*, \quad \text{and} \quad x \in D, \quad \text{such that its components } x_i \leq 1 - \Delta.$$

Δ is the step size. The elementary effect $d_i(x)$ measures the sensitivity of the function f to changes in the variable x_i at the point x .

Morris's method aims to estimate the parameters of the distribution of elementary effects associated with each variable. A large measure of central tendency indicates that a variable has a significant influence on the objective function across the design space, while a large measure of spread suggests that the variable is involved in interactions or contributes to the nonlinearity of f . In practice, the sample mean and standard deviation of a set of $d_i(x)$ values, calculated in different parts of the design space, are used for this estimation.

To ensure efficiency, the preliminary sampling plan X should be designed so that each evaluation of the objective function f contributes to the calculation of two elementary effects, rather than just one (as would occur with a naive random spread of baseline x values and adding Δ to one variable). Additionally, the sampling plan should provide a specified number (e.g., r) of elementary effects for each variable, independently drawn with replacement. For a detailed discussion on constructing such a sampling plan, readers are encouraged to consult Morris's original paper (Morris, 1991). Here, we focus on describing the process itself.

The random orientation of the sampling plan B can be constructed as follows:

- Let B be a $(k+1) \times k$ matrix of 0s and 1s, where for each column i , two rows differ only in their i -th entries.
- Compute a random orientation of B , denoted B^* :

$$B^* = (1_{k+1,k} x^* + (\Delta/2) [(2B - 1_{k+1,k}) D^* + 1_{k+1,k}]) P^*,$$

where:

- D^* is a k -dimensional diagonal matrix with diagonal elements ± 1 (equal probability),

66. Sampling Plans

- $\mathbf{1}$ is a matrix of 1s,
- x^* is a randomly chosen point in the p -level design space (limited by Δ),
- P^* is a $k \times k$ random permutation matrix with one 1 per column and row.

`spotpython` provides a Python implementation to compute B^* , see <https://github.com/sequential-parameter-optimization/spotPython/blob/main/src/spotpython/utils/effects.py>.

Here is the corresponding code:

```
def randorient(k, p, xi, seed=None):
    # Initialize random number generator with the provided seed
    if seed is not None:
        rng = np.random.default_rng(seed)
    else:
        rng = np.random.default_rng()

    # Step length
    Delta = xi / (p - 1)

    m = k + 1

    # A truncated p-level grid in one dimension
    xs = np.arange(0, 1 - Delta, 1 / (p - 1))
    xsl = len(xs)
    if xsl < 1:
        print(f"xi = {xi}.")
        print(f"p = {p}.")
        print(f"Delta = {Delta}.")
        print(f"p - 1 = {p - 1}.")
        raise ValueError(f"The number of levels xsl is {xsl}, but it must be greater t

    # Basic sampling matrix
    B = np.vstack((np.zeros((1, k)), np.tril(np.ones((k, k)))))

    # Randomization

    # Matrix with +1s and -1s on the diagonal with equal probability
    Dstar = np.diag(2 * rng.integers(0, 2, size=k) - 1)

    # Random base value
    xstar = xs[rng.integers(0, xsl, size=k)]

    # Permutation matrix
    Pstar = np.zeros((k, k))
```



```

rp = rng.permutation(k)
for i in range(k):
    Pstar[i, rp[i]] = 1

# A random orientation of the sampling matrix
Bstar = (np.ones((m, 1)) @ xstar.reshape(1, -1) +
         (Delta / 2) * ((2 * B - np.ones((m, k))) @ Dstar +
                        np.ones((m, k)))) @ Pstar

return Bstar

```

The code following snippet generates a random orientation of a sampling matrix **Bstar** using the **randorient()** function. The input parameters are:

- **k** = 3: The number of design variables (dimensions).
- **p** = 3: The number of levels in the grid for each variable.
- **xi** = 1: A parameter used to calculate the step size Delta.

Step-size calculation is performed as follows: $\Delta = \text{xi} / (p - 1) = 1 / (3 - 1) = 0.5$, which determines the spacing between levels in the grid.

Next, random sampling matrix construction is computed:

- A truncated grid is created with levels $[0, 0.5]$ (based on Delta).
- A basic sampling matrix **B** is constructed, which is a lower triangular matrix with 0s and 1s.

Then, randomization is applied:

- **Dstar**: A diagonal matrix with random entries of +1 or -1.
- **xstar**: A random starting point from the grid.
- **Pstar**: A random permutation matrix.

Random orientation is applied to the basic sampling matrix **B** to create **Bstar**. This involves scaling, shifting, and permuting the rows and columns of **B**.

The final output is the matrix **Bstar**, which represents a random orientation of the sampling plan. Each row corresponds to a sampled point in the design space, and each column corresponds to a design variable.

Example 66.2 (Random Orientation of the Sampling Matrix in 2-D).

```

k = 2
p = 3
xi = 1
Bstar = randorient(k, p, xi, seed=123)
print(f"Random orientation of the sampling matrix:\n{Bstar}")

```

66. Sampling Plans

Random orientation of the sampling matrix:

```
[[0.5 0. ]  
 [0.  0. ]  
 [0.  0.5]]
```

We can visualize the random orientation of the sampling matrix in 2-D as shown in Figure 66.1.

```
plt.figure(figsize=(6, 6))  
plt.scatter(Bstar[:, 0], Bstar[:, 1], color='blue', s=50, label='Hypercube Points')  
for i in range(Bstar.shape[0]):  
    plt.text(Bstar[i, 0] + 0.01, Bstar[i, 1] + 0.01, str(i), fontsize=9)  
plt.xlim(-0.1, 1.1)  
plt.ylim(-0.1, 1.1)  
plt.xlabel('x1')  
plt.ylabel('x2')  
plt.grid()
```

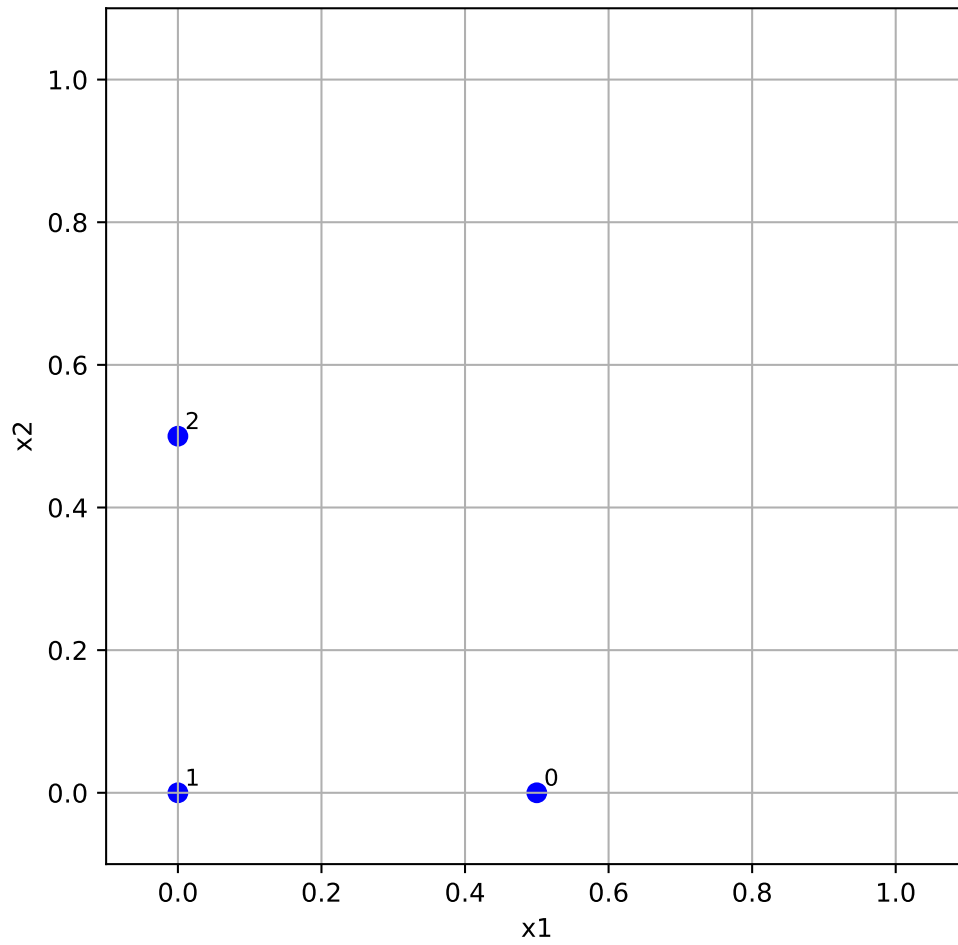


Figure 66.1.: Random orientation of the sampling matrix in 2-D. The labels indicate the row index of the points.

Example 66.3 (Random Orientation of the Sampling Matrix).

```
k = 3
p = 3
xi = 1
Bstar = randorient(k, p, xi)
print(f"Random orientation of the sampling matrix:\n{Bstar}")
```

Random orientation of the sampling matrix:

66. Sampling Plans

```
[[0.5 0.  0.5]
 [0.5 0.5 0.5]
 [0.  0.5 0.5]
 [0.  0.5 0. ]]
```

To obtain r elementary effects for each variable, the screening plan is built from r random orientations:

$$X = \begin{pmatrix} B_1^* \\ B_2^* \\ \vdots \\ B_r^* \end{pmatrix}$$

The function `screeningplan()` generates a screening plan by calling the `randorient()` function r times. It creates a list of random orientations and then concatenates them into a single array, which represents the screening plan. The screening plan implementation in Python is as follows (see <https://github.com/sequential-parameter-optimization/spotPython/blob/main/src/spotpython/utils/effects.py>):

```
def screeningplan(k, p, xi, r):
    # Empty list to accumulate screening plan rows
    X = []
    for i in range(r):
        X.append(randorient(k, p, xi))
    # Concatenate list of arrays into a single array
    X = np.vstack(X)
    return X
```

It works like follows:

- The value of the objective function f is computed for each row of the screening plan matrix X . These values are stored in a column vector t of size $(r * (k + 1)) \times 1$, where:
 - r is the number of random orientations.
 - k is the number of design variables.

The elementary effects are calculated using the following formula:

- For each random orientation, adjacent rows of the screening plan matrix X and their corresponding function values from t are used.
- These values are inserted into Equation 66.1 to compute elementary effects for each variable. An elementary effect measures the sensitivity of the objective function to changes in a specific variable.

Results can be used for a statistical analysis. After collecting a sample of r elementary effects for each variable:

- The sample mean (central tendency) is computed to indicate the overall influence of the variable.
- The sample standard deviation (spread) is computed to capture variability, which may indicate interactions or nonlinearity.

The results (sample means and standard deviations) are plotted on a chart for comparison. This helps identify which variables have the most significant impact on the objective function and whether their effects are linear or involve interactions. This is implemented in the function `screening_plot()` in `Python`, which uses the helper function `_screening()` to calculate the elementary effects and their statistics.

```
def _screening(X, fun, xi, p, labels, bounds=None) -> tuple:
    """Helper function to calculate elementary effects for a screening design.

    Args:
        X (np.ndarray): The screening plan matrix, typically structured
            within a  $[0,1]^k$  box.
        fun (object): The objective function to evaluate at each
            design point in the screening plan.
        xi (float): The elementary effect step length factor.
        p (int): Number of discrete levels along each dimension.
        labels (list of str): A list of variable names corresponding to
            the design variables.
        bounds (np.ndarray): A 2xk matrix where the first row contains
            lower bounds and the second row contains upper bounds for
            each variable.

    Returns:
        tuple: A tuple containing two arrays:
            - sm: The mean of the elementary effects for each variable.
            - ssd: The standard deviation of the elementary effects for
                each variable.
    """
    k = X.shape[1]
    r = X.shape[0] // (k + 1)

    # Scale each design point
    t = np.zeros(X.shape[0])
    for i in range(X.shape[0]):
        if bounds is not None:
            X[i, :] = bounds[0, :] + X[i, :] * (bounds[1, :] - bounds[0, :])
        t[i] = fun(X[i, :])
```

```

# Elementary effects
F = np.zeros((k, r))
for i in range(r):
    for j in range(i * (k + 1), i * (k + 1) + k):
        idx = np.where(X[j, :] - X[j + 1, :] != 0)[0][0]
        F[idx, i] = (t[j + 1] - t[j]) / (xi / (p - 1))

# Statistical measures (divide by n)
ssd = np.std(F, axis=1, ddof=0)
sm = np.mean(F, axis=1)
return sm, ssd

def screening_plot(X, fun, xi, p, labels, bounds=None, show=True) -> None:
    """Generates a plot with elementary effect screening metrics.

    This function calculates the mean and standard deviation of the
    elementary effects for a given set of design variables and plots
    the results.

    Args:
        X (np.ndarray):
            The screening plan matrix, typically structured within a  $[0,1]^k$  box.
        fun (object):
            The objective function to evaluate at each design point in the screening p
        xi (float):
            The elementary effect step length factor.
        p (int):
            Number of discrete levels along each dimension.
        labels (list of str):
            A list of variable names corresponding to the design variables.
        bounds (np.ndarray):
            A 2xk matrix where the first row contains lower bounds and
            the second row contains upper bounds for each variable.
        show (bool):
            If True, the plot is displayed. Defaults to True.

    Returns:
        None: The function generates a plot of the results.
    """
    k = X.shape[1]
    sm, ssd = _screening(X=X, fun=fun, xi=xi, p=p, labels=labels, bounds=bounds)
    plt.figure()
    for i in range(k):

```

```
plt.text(sm[i], ssd[i], labels[i], fontsize=10)
plt.axis([min(sm), 1.1 * max(sm), min(ssd), 1.1 * max(ssd)])
plt.xlabel("Sample means")
plt.ylabel("Sample standard deviations")
plt.gca().tick_params(labelsize=10)
plt.grid(True)
if show:
    plt.show()
```

66.1.4. Special Considerations When Deploying Screening Algorithms

When implementing the screening algorithm described above, two specific scenarios require special attention:

- **Duplicate Design Points:** If the dimensionality k of the space is relatively low and you can afford a large number of elementary effects r , we should be aware of the increased probability of duplicate design points appearing in the sampling plan X . *Since the responses at sample points are deterministic, there's no value in evaluating the same point multiple times. Fortunately, this issue is relatively uncommon in practice, as screening high-dimensional spaces typically requires large numbers of elementary effects, which naturally reduces the likelihood of duplicates.
- **Failed Simulations:** Numerical simulation codes occasionally fail to return valid results due to meshing errors, non-convergence of partial differential equation solvers, numerical instabilities, or parameter combinations outside the stable operating range.

From a screening perspective, this is particularly problematic because an entire random orientation B^* becomes compromised if even a single point within it fails to evaluate properly. Implementing error handling strategies or fallback methods to manage such cases should be considered.

For robust screening studies, monitoring simulation success rates and having contingency plans for failed evaluations are important aspects of the experimental design process.

66.2. Analyzing Variable Importance in Aircraft Wing Weight

Let us consider the following analytical expression used as a conceptual level estimate of the weight of a light aircraft wing as discussed in Chapter 64.

```

fun = Analytical()
k = 10
p = 10
xi = 1
r = 25
X = screeningplan(k=k, p=p, xi=xi, r=r) # shape (r x (k+1), k)
value_range = np.array([
    [150, 220, 6, -10, 16, 0.5, 0.08, 2.5, 1700, 0.025],
    [200, 300, 10, 10, 45, 1.0, 0.18, 6.0, 2500, 0.08 ],
])
labels = [
    "S_W", "W_fw", "A", "Lambda",
    "q", "lambda", "tc", "N_z",
    "W_dg", "W_p"
]
screening_plot(
    X=X,
    fun=fun.fun_wingwt,
    bounds=value_range,
    xi=xi,
    p=p,
    labels=labels,
)

```

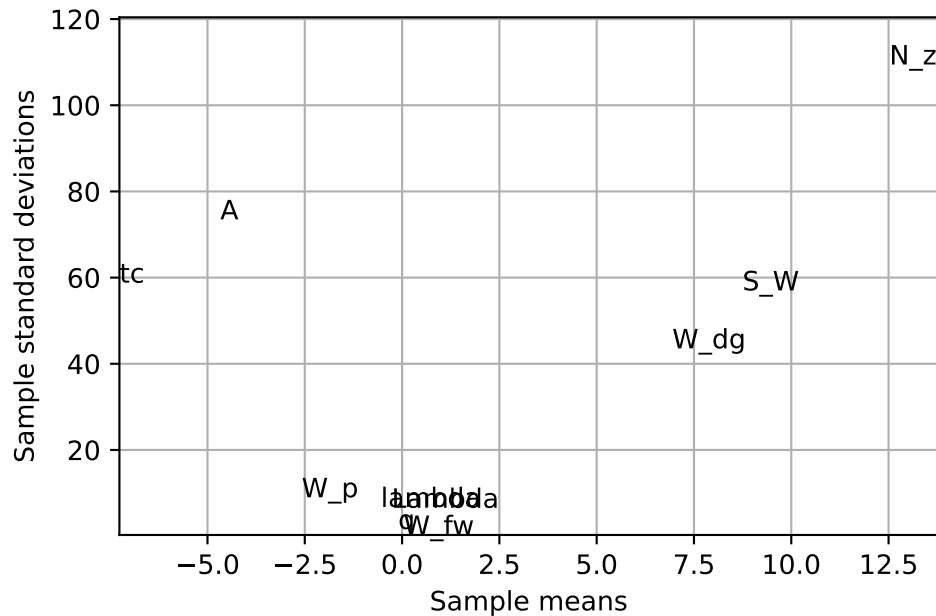



Figure 66.2.: Estimated means and standard deviations of the elementary effects for the 10 design variables of the wing weight function. Example based on Forrester, Sóbester, and Keane (2008).

i Nondeterministic Results

The code will generate a slightly different screening plan each time, as it uses random orientations of the sampling matrix B .

Figure 66.2 provides valuable insights into variable activity without requiring domain expertise. The screening study with $r = 25$ elementary effects reveals distinct patterns in how variables affect wing weight:

- Variables with Minimal Impact: A clearly defined group of variables clusters around the origin - indicating their minimal impact on wing weight:
 - Paint weight (W_p) - as expected, contributes little to overall wing weight
 - Dynamic pressure (q) - within our chosen range, this has limited effect (essentially representing different cruise altitudes at the same speed)
 - Taper ratio (λ) and quarter-chord sweep (Λ) - these geometric parameters have minor influence within the narrow range (-10° to 10°) typical of light aircraft
- Variables with Linear Effects:

66. Sampling Plans

- While still close to the origin, fuel weight (W_{fw}) shows a slightly larger central tendency with very low standard deviation. This indicates moderate importance but minimal involvement in interactions with other variables.
- Variables with Nonlinear/Interactive Effects:
 - Aspect ratio (A) and airfoil thickness ratio (R_{tc}) show similar importance levels, but their high standard deviations suggest significant nonlinear behavior and interactions with other variables.
- Dominant Variables: The most significant impacts come from:
 - Flight design gross weight (W_{dg})
 - Wing area (S_W)
 - Ultimate load factor (N_z)

These variables show both large central tendency values and high standard deviations, indicating strong direct effects and complex interactions. The interaction between aspect ratio and load factor is particularly important - high values of both create extremely heavy wings, explaining why highly maneuverable fighter jets cannot use glider-like wing designs.

What makes this screening approach valuable is its ability to identify critical variables without requiring engineering knowledge or expensive modeling. In real-world applications, we rarely have the luxury of creating comprehensive parameter space visualizations, which is precisely why surrogate modeling is needed. After identifying the active variables through screening, we can design a focused sampling plan for these key variables. This forms the foundation for building an accurate surrogate model of the objective function.

When the objective function is particularly expensive to evaluate, we might recycle the runs performed during screening for the actual model fitting step. This is most effective when some variables prove to have no impact at all. However, since completely inactive variables are rare in practice, engineers must carefully balance the trade-off between reusing expensive simulation runs and introducing potential noise into the model.

66.3. Designing a Sampling Plan

66.3.1. Stratification

A feature shared by all of the approximation models discussed in Forrester, Sóbester, and Keane (2008) is that they are more accurate in the vicinity of the points where we have evaluated the objective function. In later chapters we will delve into the laws that quantify our decaying trust in the model as we move away from a known, sampled point, but for the purposes of the present discussion we shall merely draw the intuitive conclusion that a uniform level of model accuracy throughout the design

space requires a uniform spread of points. A sampling plan possessing this feature is said to be space-filling.

The most straightforward way of sampling a design space in a uniform fashion is by means of a rectangular grid of points. This is the full factorial sampling technique.

Here is the simplified version of a **Python** function that will sample the unit hypercube at all levels in all dimensions, with the k -vector q containing the number of points required along each dimension, see <https://github.com/sequential-parameter-optimization/spotPython/blob/main/src/spotpython/utils/sampling.py>.

The variable **Edges** specifies whether we want the points to be equally spaced from edge to edge (**Edges=1**) or we want them to be in the centres of $n = q_1 \times q_2 \times \dots \times q_k$ bins filling the unit hypercube (for any other value of **Edges**).

```
def fullfactorial(q_param, Edges=1) -> np.ndarray:
    """Generates a full factorial sampling plan in the unit cube.

    Args:
        q (list or np.ndarray):
            A list or array containing the number of points along each dimension (k-vector).
        Edges (int, optional):
            Determines spacing of points. If `Edges=1`, points are equally spaced from edge to edge.
            Otherwise, points will be in the centers of  $n = q[0] \times q[1] \times \dots \times q[k-1]$  bins filling the unit cube.

    Returns:
        (np.ndarray): Full factorial sampling plan as an array of shape (n, k), where n is the total number of points.

    Raises:
        ValueError: If any dimension in `q` is less than 2.
    """
    q_levels = np.array(q_param) # Use a distinct variable for original levels
    if np.min(q_levels) < 2:
        raise ValueError("You must have at least two points per dimension.")

    n = np.prod(q_levels)
    k = len(q_levels)
    X = np.zeros((n, k))

    # q_for_prod_calc is used for calculating repetitions, includes the phantom element.
    # This matches the logic of the user-provided snippet where 'q' was modified.
    q_for_prod_calc = np.append(q_levels, 1)

    for j in range(k): # k is the original number of dimensions
        # current_dim_levels is the number of levels for the current dimension j
        # In the user's snippet, q[j] correctly refers to the original level count
```

```

# as j ranges from 0 to k-1, and q_for_prod_calc[j] = q_levels[j] for this ran
current_dim_levels = q_for_prod_calc[j]

if Edges == 1:
    one_d_slice = np.linspace(0, 1, int(current_dim_levels))
else:
    # Corrected calculation for bin centers
    if current_dim_levels == 1: # Should not be hit if np.min(q_levels) >= 2
        one_d_slice = np.array([0.5])
    else:
        one_d_slice = np.linspace(1 / (2 * current_dim_levels),
                                   1 - 1 / (2 * current_dim_levels),
                                   int(current_dim_levels))

column = np.array([])
# The product q_for_prod_calc[j + 1 : k] correctly calculates
# the product of remaining original dimensions' levels.
num_consecutive_repeats = np.prod(q_for_prod_calc[j + 1 : k])

# This loop structure replicates the logic from the user's snippet
while len(column) < n:
    for ll_idx in range(int(current_dim_levels)): # Iterate through levels of
        val_to_repeat = one_d_slice[ll_idx]
        column = np.append(column, np.ones(int(num_consecutive_repeats)) * va
    X[:, j] = column
return X

q = [3, 2]
X = fullfactorial(q, Edges=0)
print(X)

```

```

[[0.16666667 0.25      ]
 [0.16666667 0.75      ]
 [0.5        0.25      ]
 [0.5        0.75      ]
 [0.83333333 0.25      ]
 [0.83333333 0.75      ]]

```

Figure 66.3 shows the points in the unit hypercube for the case of 3x2 points.

```

X = fullfactorial(q, Edges=1)
print(X)

```

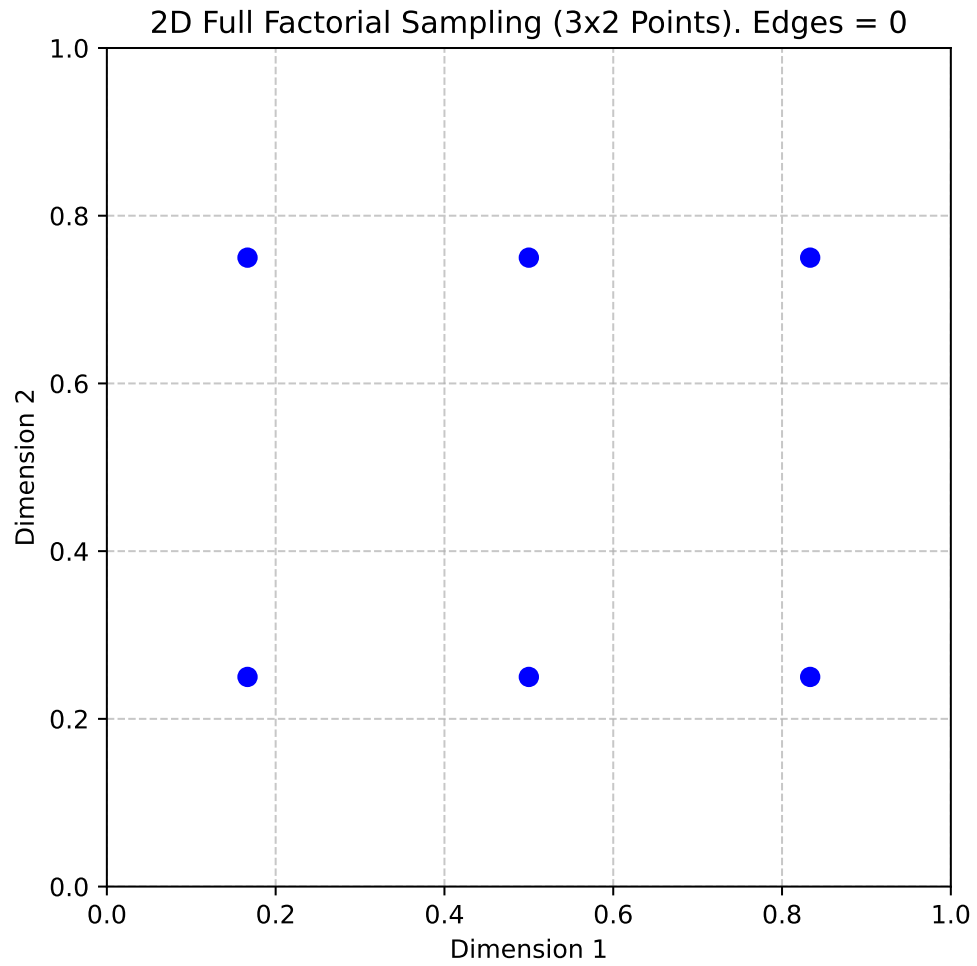


Figure 66.3.: 2D Full Factorial Sampling (3x2 Points). Edges = 0

66. Sampling Plans

```
[[0.  0. ]  
 [0.  1. ]  
 [0.5 0. ]  
 [0.5 1. ]  
 [1.  0. ]  
 [1.  1. ]]
```

Figure 66.4 shows the points in the unit hypercube for the case of 3x2 points with edges.

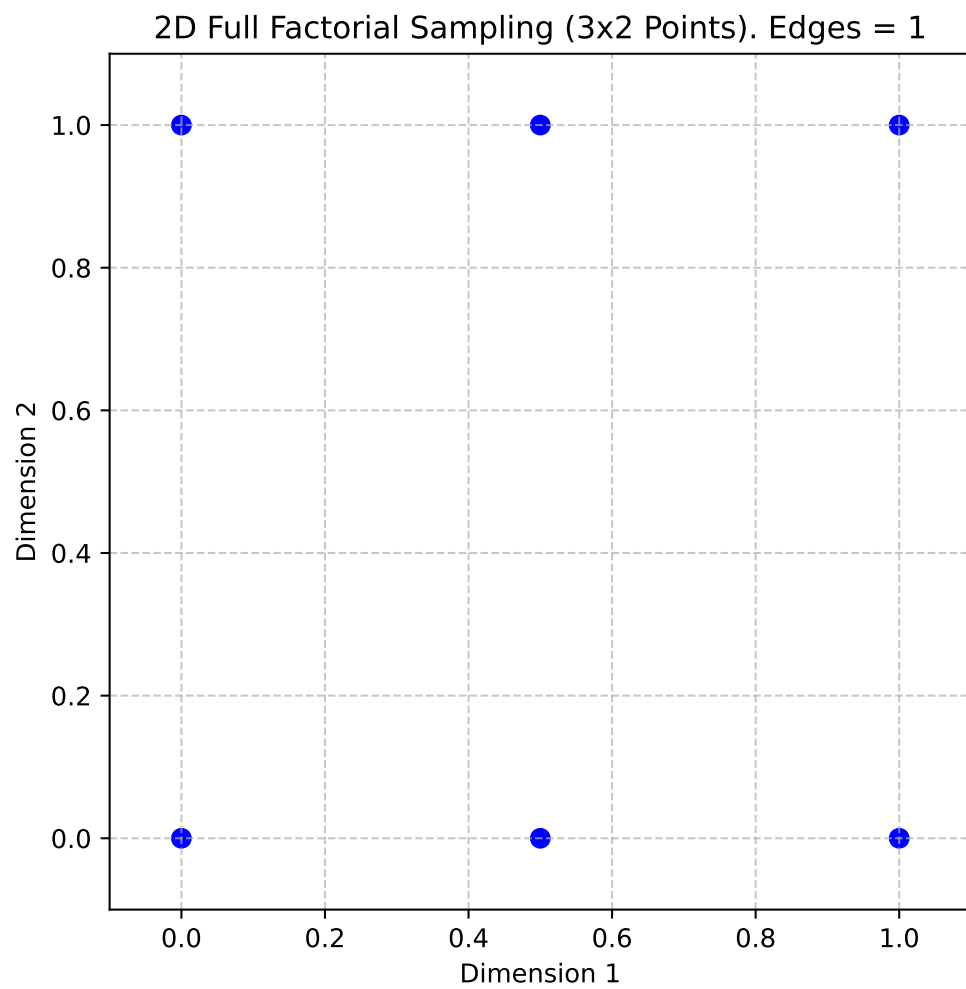


Figure 66.4.: 2D Full Factorial Sampling (3x2 Points). Edges = 1

The full factorial sampling plan method generates a uniform sampling design by creating a grid of points across all dimensions. For example, calling `fullfactorial([3, 4, 5], 1)` produces a three-dimensional sampling plan with 3, 4, and 5 levels along each dimension, respectively. While this approach satisfies the uniformity criterion, it has two significant limitations:

- **Restricted Design Sizes:** The method only works for designs where the total number of points n can be expressed as the product of the number of levels in each dimension, i.e., $n = q_1 \times q_2 \times \dots \times q_k$.
- **Overlapping Projections:** When the sampling points are projected onto individual axes, sets of points may overlap, reducing the effectiveness of the sampling plan. This can lead to non-uniform coverage in the projections, which may not fully represent the design space.

66.3.2. Latin Squares and Random Latin Hypercubes

To improve the uniformity of projections for any individual variable, the range of that variable can be divided into a large number of equal-sized bins, and random subsamples of equal size can be generated within these bins. This method is called stratified random sampling. Extending this idea to all dimensions results in a stratified sampling plan, commonly implemented using Latin hypercube sampling.

Definition 66.3 (Latin Squares and Hypercubes). In the context of statistical sampling, a square grid containing sample positions is a Latin square if (and only if) there is only one sample in each row and each column. A Latin hypercube is the generalisation of this concept to an arbitrary number of dimensions, whereby each sample is the only one in each axis-aligned hyperplane containing it

For two-dimensional discrete variables, a Latin square ensures uniform projections. An $(n \times n)$ Latin square is constructed by filling each row and column with a permutation of $\{1, 2, \dots, n\}$, ensuring each number appears only once per row and column.

Example 66.4 (Latin Square). For $n = 4$, a Latin square might look like this:

2	1	3	4
3	2	4	1
1	4	2	3
4	3	1	2

Latin Hypercubes are the multidimensional extension of Latin squares. The design space is divided into equal-sized hypercubes (bins), and one point is placed in each bin. The placement ensures that moving along any axis from an occupied bin does not encounter another occupied bin. This guarantees uniform projections across all dimensions. To construct a Latin hypercube, the following steps are taken:

66. Sampling Plans

- Represent the sampling plan as an $n \times k$ matrix X , where n is the number of points and k is the number of dimensions.
- Fill each column of X with random permutations of $\{1, 2, \dots, n\}$.
- Normalize the plan into the unit hypercube $[0, 1]^k$.

This approach ensures multidimensional stratification and uniformity in projections. Here is the code:

```
def rlh(n: int, k: int, edges: int = 0) -> np.ndarray:
    # Initialize array
    X = np.zeros((n, k), dtype=float)

    # Fill with random permutations
    for i in range(k):
        X[:, i] = np.random.permutation(n)

    # Adjust normalization based on the edges flag
    if edges == 1:
        # [X=0..n-1] -> [0..1]
        X = X / (n - 1)
    else:
        # Points at true midpoints
        # [X=0..n-1] -> [0.5/n..(n-0.5)/n]
        X = (X + 0.5) / n

    return X
```

Example 66.5 (Random Latin Hypercube). The following code can be used to generate a 2D Latin hypercube with 5 points and edges=0:

```
X = rlh(n=5, k=2, edges=0)
print(X)
```

```
[[0.3 0.1]
 [0.5 0.7]
 [0.9 0.5]
 [0.7 0.3]
 [0.1 0.9]]
```

Figure 66.5 shows the points in the unit hypercube for the case of 5 points with edges=0.

Example 66.6 (Random Latin Hypercube with Edges). The following code can be used to generate a 2D Latin hypercube with 5 points and edges=1:

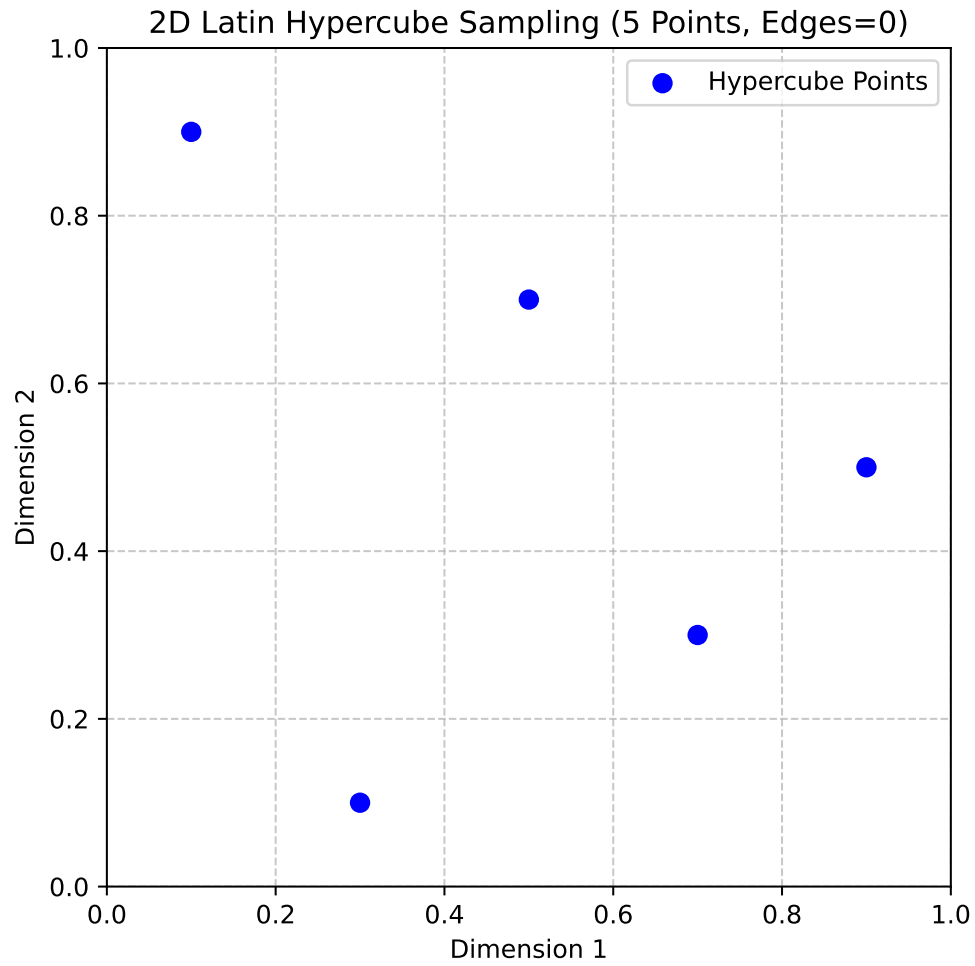


Figure 66.5.: 2D Latin Hypercube Sampling (5 Points, Edges=0)

```
X = r1h(n=5, k=2, edges=1)
print(X)
```

```
[[0.5  0.  ]
 [1.   0.75]
 [0.75 0.25]
 [0.   1.  ]
 [0.25 0.5  ]]
```

Figure 66.6 shows the points in the unit hypercube for the case of 5 points with `edges=1`.

66.3.3. Space-filling Designs: Maximin Plans

A widely adopted measure for assessing the uniformity, or ‘space-fillingness’, of a sampling plan is the maximin metric, initially proposed by Johnson, Moore, and Ylvisaker (1990). This criterion can be formally defined as follows.

Consider a sampling plan X . Let $d_1, d_2, \dots, d_m$ represent the unique distances between all possible pairs of points within X , arranged in ascending order. Furthermore, let $J_1, J_2, \dots, J_m$ be defined such that J_j denotes the count of point pairs in X separated by the distance d_j .

Definition 66.4 (Maximin plan). A sampling plan X is considered a maximin plan if, among all candidate plans, it maximizes the smallest inter-point distance d_1 . Among plans that satisfy this condition, it further minimizes J_1 , the number of pairs separated by this minimum distance.

While this definition is broadly applicable to any collection of sampling plans, our focus is narrowed to Latin hypercube designs to preserve their desirable stratification properties. However, even within this restricted class, Definition 66.4 may identify multiple equivalent maximin designs. To address this, a more comprehensive ‘tie-breaker’ definition, as proposed by Morris and Mitchell (1995), is employed:

Definition 66.5 (Maximin plan with tie-breaker). A sampling plan X is designated as the maximin plan if it sequentially optimizes the following conditions: it maximizes d_1 ; among those, it minimizes J_1 ; among those, it maximizes d_2 ; among those, it minimizes J_2 ; and so forth, concluding with minimizing J_m .

Johnson, Moore, and Ylvisaker (1990) established that the maximin criterion (Definition 66.4) is equivalent to the D-optimality criterion used in linear regression. However, the extended maximin criterion incorporating a tie-breaker (Definition 66.5) is often

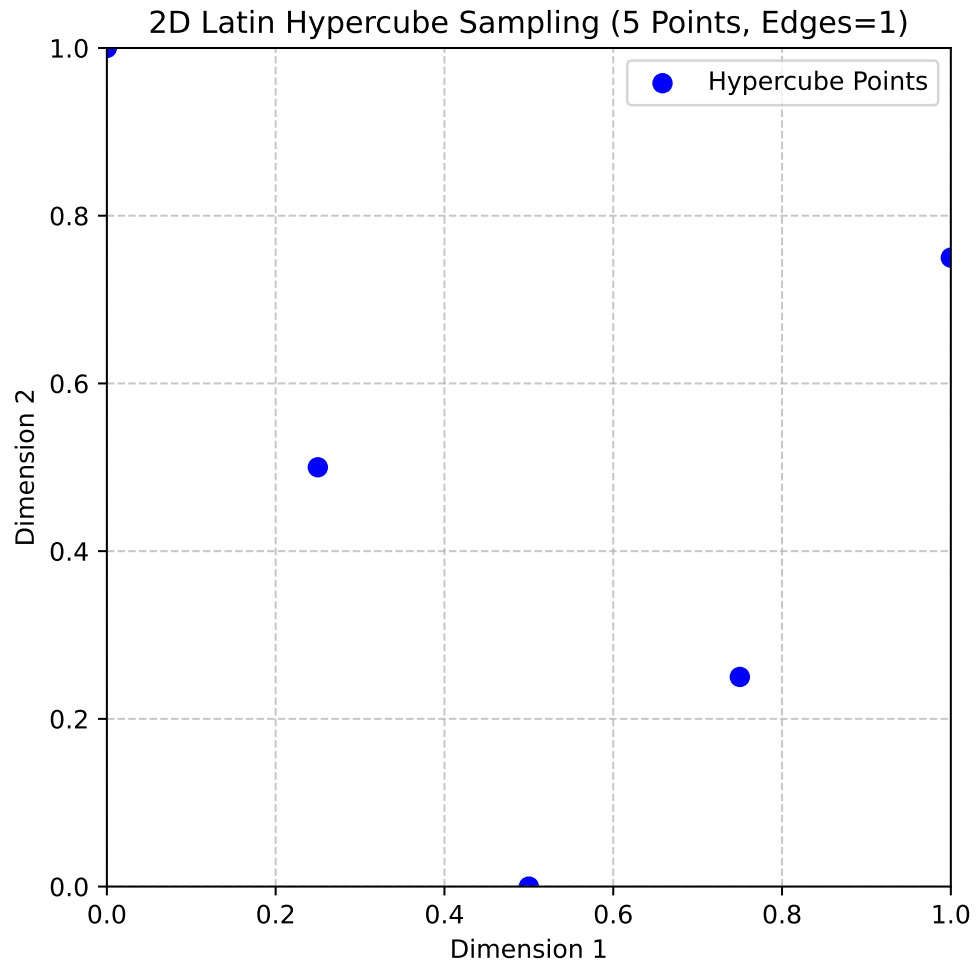


Figure 66.6.: 2D Latin Hypercube Sampling (5 Points, Edges=1)

preferred due to its intuitive nature and practical utility. Given that the sampling plans under consideration make no assumptions about model structure, the latter criterion (Definition 66.5) will be employed.

To proceed, a precise definition of ‘distance’ within these contexts is necessary. The p-norm is the most widely adopted metric for this purpose:

Definition 66.6 (p-norm). The p-norm of a vector $\vec{x} = (x_1, x_2, \dots, x_k)$ is defined as:

$$d_p(\vec{x}^{(i_1)}, \vec{x}^{(i_2)}) = \left(\sum_{j=1}^k |x_j^{(i_1)} - x_j^{(i_2)}|^p \right)^{1/p}. \quad (66.2)$$

When $p = 1$, Equation 66.2 defines the rectangular distance, occasionally referred to as the Manhattan norm (an allusion to a grid-like city layout). Setting $p = 2$ yields the Euclidean norm. The existing literature offers limited evidence to suggest the superiority of one norm over the other for evaluating sampling plans when no model structure assumptions are made. It is important to note, however, that the rectangular distance is considerably less computationally demanding. This advantage can be quite significant, particularly when evaluating large sampling plans.

For the computational implementation of Definition 66.5, the initial step involves constructing the vectors $d_1, d_2, \dots, d_m$ and $J_1, J_2, \dots, J_m$. The `jd` function facilitates this task.

66.3.3.1. The Function `jd`

The function `jd` computes the distinct p-norm distances between all pairs of points in a given set and counts their occurrences. It returns two arrays: one for the distinct distances and another for their multiplicities.

```
def jd(X: np.ndarray, p: float = 1.0) -> Tuple[np.ndarray, np.ndarray]:
    """
    Args:
        X (np.ndarray):
            A 2D array of shape (n, d) representing n points
            in d-dimensional space.
        p (float, optional):
            The distance norm to use.
            p=1 uses the Manhattan (L1) norm, while p=2 uses the
            Euclidean (L2) norm. Defaults to 1.0 (Manhattan norm).

    Returns:
        (np.ndarray, np.ndarray):
            A tuple (J, distinct_d), where:
```

```

- distinct_d is a 1D float array of unique,
  sorted distances between points.
- J is a 1D integer array that provides
  the multiplicity (occurrence count)
  of each distance in distinct_d.
"""
n = X.shape[0]

# Allocate enough space for all pairwise distances
# (n*(n-1))/2 pairs for an n-point set
pair_count = n * (n - 1) // 2
d = np.zeros(pair_count, dtype=float)

# Fill the distance array
idx = 0
for i in range(n - 1):
    for j in range(i + 1, n):
        # Compute the p-norm distance
        d[idx] = np.linalg.norm(X[i] - X[j], ord=p)
        idx += 1

# Find unique distances and their multiplicities
distinct_d = np.unique(d)
J = np.zeros_like(distinct_d, dtype=int)
for i, val in enumerate(distinct_d):
    J[i] = np.sum(d == val)
return J, distinct_d

```

Example 66.7 (The Function `jd`). Consider a small 3-point set in 2D space, with points located at (0,0), (1,1), and (2,2) as shown in Figure 66.7. The distinct distances and their occurrences can be computed using the `jd` function, as shown in the following code:

```

J, distinct_d = jd(X, p=2.0)
print("Distinct distances (d_i):", distinct_d)
print("Occurrences (J_i):", J)

```

```

Distinct distances (d_i): [1.41421356 2.82842712]
Occurrences (J_i): [2 1]

```

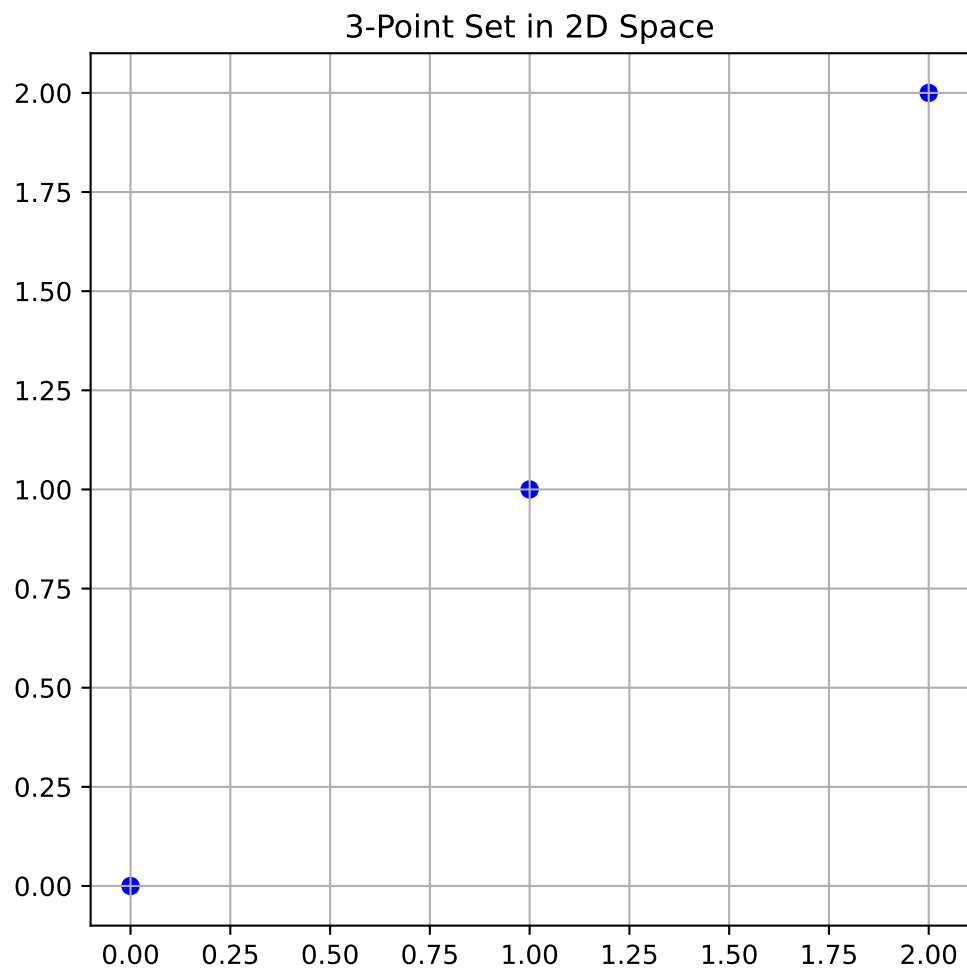


Figure 66.7.: 3-Point Set in 2D Space

66.3.4. Memory Management

A computationally intensive part of the calculation performed with the `jd`-function is the creation of the vector $\vec{d}$ containing all pairwise distances. This is particularly true for large sampling plans; for instance, a 1000-point plan requires nearly half a million distance calculations.

Definition 66.7 (Pre-allocation of Memory). Pre-allocation of memory is a programming technique where a fixed amount of memory is reserved for a data structure (like an array or vector) before it is actually filled with data. This is done to avoid the computational overhead associated with dynamic memory allocation, which involves repeatedly requesting and resizing memory as new elements are added.

Consequently, pre-allocating memory for the distance vector $\vec{d}$ is essential. This necessitates a slightly less direct method for computing the indices of $\vec{d}$, rather than appending each new element, which would involve costly dynamic memory allocation.

The implementation of Definition 66.5 is now required. Finding the most space-filling design involves pairwise comparisons. This problem can be approached using a ‘divide and conquer’ strategy, simplifying it to the task of selecting the better of two sampling plans. The function `mm(X1,X2,p)` is designed for this purpose. It returns an index indicating which of the two designs is more space-filling, or 0 if they are equally space-filling, based on the p -norm for distance computation.

66.3.4.1. The Function `mm`

The function `mm` compares two sampling plans based on the Morris-Mitchell criterion. It uses the `jd` function to compute the distances and multiplicities, constructs vectors for comparison, and determines which plan is more space-filling.

```
def mm(X1: np.ndarray, X2: np.ndarray, p: Optional[float] = 1.0) -> int:
    """
    Args:
        X1 (np.ndarray): A 2D array representing the first sampling plan.
        X2 (np.ndarray): A 2D array representing the second sampling plan.
        p (float, optional): The distance metric. p=1 uses Manhattan (L1) distance,
            while p=2 uses Euclidean (L2). Defaults to 1.0.

    Returns:
        int:
            - 0 if both plans are identical or equally space-filling
            - 1 if X1 is more space-filling
            - 2 if X2 is more space-filling
    """
```

```

X1_sorted = X1[np.lexsort(np.rot90(X1))]
X2_sorted = X2[np.lexsort(np.rot90(X2))]
if np.array_equal(X1_sorted, X2_sorted):
    return 0 # Identical sampling plans

# Compute distance multiplicities for each plan
J1, d1 = jd(X1, p)
J2, d2 = jd(X2, p)
m1, m2 = len(d1), len(d2)

# Construct V1 and V2: alternate distance and negative multiplicity
V1 = np.zeros(2 * m1)
V1[0::2] = d1
V1[1::2] = -J1

V2 = np.zeros(2 * m2)
V2[0::2] = d2
V2[1::2] = -J2

# Trim the longer vector to match the size of the shorter
m = min(m1, m2)
V1 = V1[:m]
V2 = V2[:m]

# Compare element-by-element:
# c[i] = 1 if V1[i] > V2[i], 2 if V1[i] < V2[i], 0 otherwise.
c = (V1 > V2).astype(int) + 2 * (V1 < V2).astype(int)

if np.sum(c) == 0:
    # Equally space-filling
    return 0
else:
    # The first non-zero entry indicates which plan is better
    idx = np.argmax(c != 0)
    return c[idx]

```

Example 66.8 (The Function `mm`). We can use the `mm` function to compare two sampling plans. The following code creates two 3-point sampling plans in 2D (shown in Figure 66.8) and compares them using the Morris-Mitchell criterion:

```

X1 = np.array([[0.0, 0.0], [0.5, 0.5], [0.0, 1.0], [1.0, 1.0]])
X2 = np.array([[0.1, 0.1], [0.4, 0.6], [0.1, 0.9], [0.9, 0.9]])

```

We can compare which plan has better space-filling (Morris-Mitchell). The output is

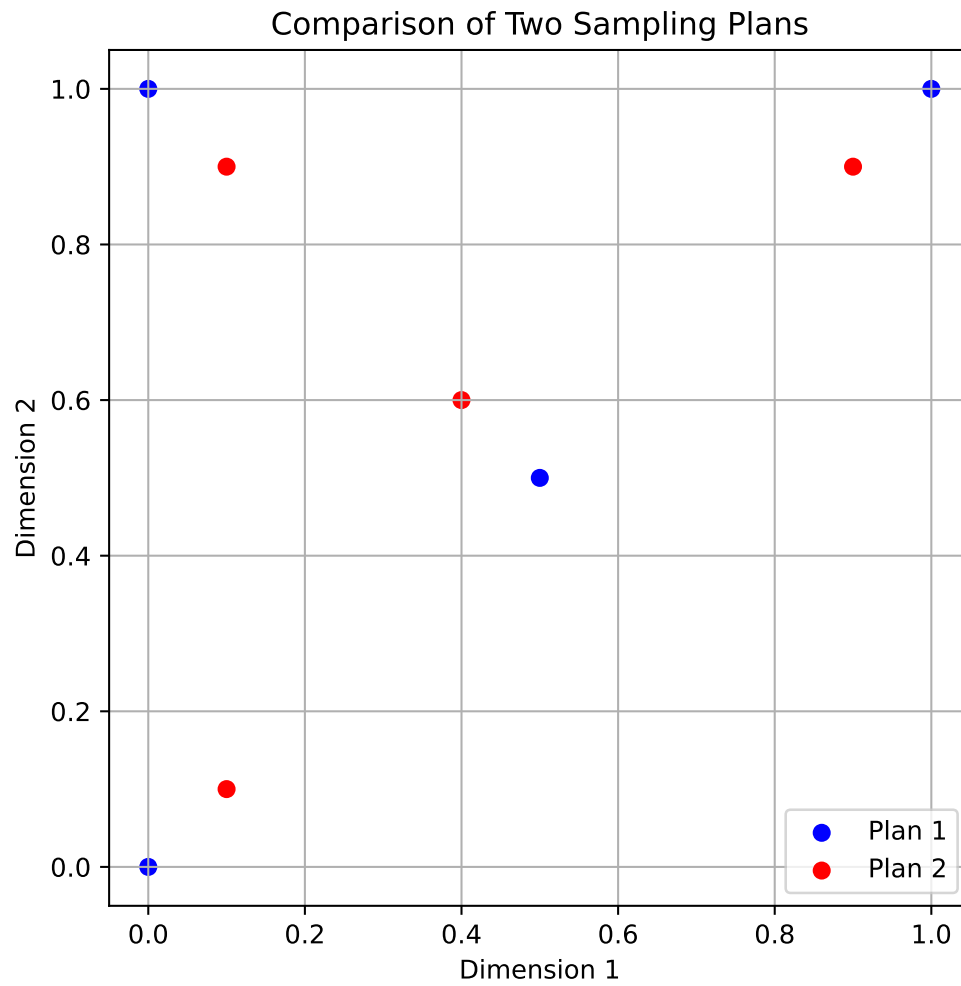


Figure 66.8.: Comparison of Two Sampling Plans

66. Sampling Plans

either 0, 1, or 2 depending on which plan is more space-filling.

```
better = mm(X1, X2, p=2.0)
print(f"Plan {better} is more space-filling.")
```

Plan 1 is more space-filling.

66.3.4.2. The Function `mmphi`

Searching across a space of potential sampling plans can be accomplished by pairwise comparisons. An optimization algorithm could, in theory, be written with `mm` as the comparative objective. However, experimental evidence (Morris and Mitchell 1995) suggests that the resulting optimization landscape can be quite deceptive, making it difficult to search reliably. This difficulty arises because the comparison process terminates upon finding the first non-zero element in the comparison array `c`. Consequently, the remaining values in the distance $(d_1, d_2, \dots, d_m)$ and multiplicity $(J_1, J_2, \dots, J_m)$ arrays are disregarded. These disregarded values, however, might contain potentially useful ‘slope’ information about the global landscape for the optimization process.

To address this, Morris and Mitchell (1995) defined the following scalar-valued criterion function, which is used to rank competing sampling plans. This function, while based on the logic of Definition 66.5, incorporates the complete vectors $d_1, d_2, \dots, d_m$ and $J_1, J_2, \dots, J_m$.

Definition 66.8 (Morris-Mitchell Criterion). The Morris-Mitchell criterion is defined as:

$$\Phi_q(X) = \left(\sum_{j=1}^m J_j d_j^{-q} \right)^{1/q}, \quad (66.3)$$

where X is the sampling plan, d_j is the distance between points, J_j is the multiplicity of that distance, and q is a user-defined exponent. The parameter q can be adjusted to control the influence of smaller distances on the overall metric.

The smaller the value of Φ_q , the better the space-filling properties of X will be.

The function `mmphi` computes the Morris-Mitchell sampling plan quality criterion for a given sampling plan. It takes a 2D array of points and calculates the space-fillingness metric based on the distances between points. This can be implemented in Python as follows:

```

def mmphi(X: np.ndarray,
          q: Optional[float] = 2.0,
          p: Optional[float] = 1.0) -> float:
    """
    Args:
        X (np.ndarray):
            A 2D array representing the sampling plan,
            where each row is a point in
            d-dimensional space (shape: (n, d)).
        q (float, optional):
            Exponent used in the computation of the metric.
            Defaults to 2.0.
        p (float, optional):
            The distance norm to use.
            For example, p=1 is Manhattan (L1),
            p=2 is Euclidean (L2). Defaults to 1.0.

    Returns:
        float:
            The space-fillingness metric  $\Phi_q$ . Larger values typically indicate a more
            space-filling plan according to the Morris-Mitchell criterion.
    """
    # Compute the distance multiplicities: J, and unique distances: d
    J, d = jd(X, p)
    # Summation of  $J[i] * d[i]^{-q}$ , then raised to  $1/q$ 
    # This follows the Morris-Mitchell definition.
     $\Phi_q$  = np.sum(J * (d ** (-q))) ** (1.0 / q)
    return  $\Phi_q$ 

```

Example 66.9 (The Function `mmphi`). We can use the `mmphi` function to evaluate the space-filling quality of the two sampling plans from Example 66.8. The following code uses these two 3-point sampling plans in 2D and computes their quality using the Morris-Mitchell criterion:

```

# Two simple sampling plans from above
quality1 = mmphi(X1, q=2, p=2)
quality2 = mmphi(X2, q=2, p=2)
print(f"Quality of sampling plan X1: {quality1}")
print(f"Quality of sampling plan X2: {quality2}")

```

```

Quality of sampling plan X1: 2.91547594742265
Quality of sampling plan X2: 3.917162046269215

```

This equation provides a more compact representation of the maximin criterion, but the selection of the q value is an important consideration. Larger values of q ensure that terms in the sum corresponding to smaller inter-point distances (the d_j values, which are sorted in ascending order) have a dominant influence. As a result, Φ_q will rank sampling plans in a way that closely emulates the original maximin definition (Definition 66.5). This implies that the optimization landscape might retain the challenging characteristics that the Φ_q metric, especially with smaller q values, is intended to alleviate. Conversely, smaller q values tend to produce a Φ_q landscape that, while not perfectly aligning with the original definition, is generally more conducive to optimization.

To illustrate the relationship between Equation 66.3 and the maximin criterion of Definition 66.5, sets of 50 random Latin hypercubes of varying sizes and dimensionalities were considered by Forrester, Sóbester, and Keane (2008). The correlation plots from this analysis suggest that as the sampling plan size increases, a smaller q value is needed for the Φ_q -based ranking to closely match the ranking derived from Definition 66.5.

Rankings based on both the direct maximin comparison (`mm`) and the Φ_q metric (`mmphi`), determined using a simple bubble sort algorithm, are implemented in the Python function `mmsort`.

66.3.4.3. The Function `mmsort`

The function `mmsort` is designed to rank multiple sampling plans based on their space-filling properties using the Morris-Mitchell criterion. It takes a 3D array of sampling plans and returns the indices of the plans sorted in ascending order of their space-filling quality.

```
def mmsort(X3D: np.ndarray, p: Optional[float] = 1.0) -> np.ndarray:
    """
    Args:
        X3D (np.ndarray):
            A 3D NumPy array of shape (n, d, m), where m is the number of
            sampling plans, and each plan is an (n, d) matrix of points.
        p (float, optional):
            The distance metric to use. p=1 for Manhattan (L1), p=2 for
            Euclidean (L2). Defaults to 1.0.

    Returns:
        np.ndarray:
            A 1D integer array of length m that holds the plan indices in
            ascending order of space-filling quality. The first index in the
            returned array corresponds to the most space-filling plan.
    """
    # Number of plans (m)
    m = X3D.shape[2]
```

```

# Create index array (1-based to match original MATLAB convention)
Index = np.arange(1, m + 1)

swap_flag = True
while swap_flag:
    swap_flag = False
    i = 0
    while i < m - 1:
        # Compare plan at Index[i] vs. Index[i+1] using mm()
        # Note: subtract 1 from each index to convert to 0-based array indexing
        if mm(X3D[:, :, Index[i] - 1], X3D[:, :, Index[i + 1] - 1], p) == 2:
            # Swap indices if the second plan is more space-filling
            Index[i], Index[i + 1] = Index[i + 1], Index[i]
            swap_flag = True
        i += 1

return Index

```

Example 66.10 (The Function `mmsort`). The `mmsort` function can be used to rank multiple sampling plans based on their space-filling properties. The following code demonstrates how to use `mmsort` to compare two 3-point sampling plans in 3D space:

Suppose we have two 3-point sampling plans X_1 and X_2 from above. They are sorted using the Morris-Mitchell criterion with $p = 2.0$. For example, the output `[1, 2]` indicates that X_1 is more space-filling than X_2 :

```

X3D = np.stack([X1, X2], axis=2)
ranking = mmsort(X3D, p=2.0)
print(ranking)

```

```
[1 2]
```

To determine the optimal Latin hypercube for a specific application, a recommended approach by Morris and Mitchell (1995) involves minimizing Φ_q for a set of q values (1, 2, 5, 10, 20, 50, and 100). Subsequently, the best plan from these results is selected based on the actual maximin definition. The `mmsort` function can be utilized for this purpose: a 3D matrix, X_{3D} , can be constructed where each 2D slice represents the best sampling plan found for each Φ_q . Applying `mmsort(X3D, 1)` then ranks these plans according to Definition 66.5, using the rectangular distance metric. The subsequent discussion will address the methods for finding these optimized Φ_q designs.

66.3.4.4. The Function phisort

`phisort` only differs from `mmsort` in having `q` as an additional argument, as well as the comparison line being:

```
if mmphi(X3D[:, :, Index[i] - 1], q=q, p=p) >
    mmphi(X3D[:, :, Index[i + 1] - 1], q=q, p=p):
```

```
def phisort(X3D: np.ndarray,
            q: Optional[float] = 2.0,
            p: Optional[float] = 1.0) -> np.ndarray:
    """
    Args:
        X3D (np.ndarray):
            A 3D array of shape (n, d, m),
            where m is the number of sampling plans.
        q (float, optional):
            Exponent for the mmphi metric. Defaults to 2.0.
        p (float, optional):
            Distance norm for mmphi.
            p=1 is Manhattan; p=2 is Euclidean.
            Defaults to 1.0.

    Returns:
        np.ndarray:
            A 1D integer array of length m, giving the plan indices in ascending
            order of mmphi. The first index in the returned array corresponds
            to the numerically lowest mmphi value.
    """
    # Number of 2D sampling plans
    m = X3D.shape[2]
    # Create a 1-based index array
    Index = np.arange(1, m + 1)
    # Bubble-sort: plan with lower mmphi() climbs toward the front
    swap_flag = True
    while swap_flag:
        swap_flag = False
        for i in range(m - 1):
            # Retrieve mmphi values for consecutive plans
            val_i = mmphi(X3D[:, :, Index[i] - 1], q=q, p=p)
            val_j = mmphi(X3D[:, :, Index[i + 1] - 1], q=q, p=p)

            # Swap if the left plan's mmphi is larger (i.e. 'worse')
            if val_i > val_j:
```

```

        Index[i], Index[i + 1] = Index[i + 1], Index[i]
        swap_flag = True
    return Index

```

Example 66.11 (The Function `phisort`). The `phisort` function can be used to rank multiple sampling plans based on the Morris-Mitchell criterion. The following code demonstrates how to use `phisort` to compare two 3-point sampling plans in 3D space:

```

X1 = bestlh(n=5, k=2, population=5, iterations=10)
X2 = bestlh(n=5, k=2, population=15, iterations=20)
X3 = bestlh(n=5, k=2, population=25, iterations=30)
# Map X1 and X2 so that X3D has the two sampling plans
# in X3D[:, :, 0] and X3D[:, :, 1]
X3D = np.array([X1, X2])
print(phisort(X3D))
X3D = np.array([X3, X2])
print(phisort(X3D))

```

```

[1 2]
[1 2]

```

66.3.5. Optimizing the Morris-Mitchell Criterion Φ_q

Once a criterion for assessing the quality of a Latin hypercube sampling plan has been established, a systematic method for optimizing this metric across the space of Latin hypercubes is required. This task is non-trivial; as the reader may recall from the earlier discussion on Latin squares, this search space is vast. In fact, its vastness means that for many practical applications, locating the globally optimal solution is often infeasible. Therefore, the objective becomes finding the best possible sampling plan achievable within a specific computational time budget.

This budget is influenced by the computational cost associated with obtaining each objective function value. Determining the optimal allocation of total computational effort—between generating the sampling plan and actually evaluating the objective function at the selected points—remains an open research question. However, it is typical for no more than approximately 5% of the total available time to be allocated to the task of generating the sampling plan itself.

Forrester, Sóbester, and Keane (2008) draw an analogy to the process of devising a revision timetable before an exam. While a well-structured timetable enhances the effectiveness of revision, an excessive amount of the revision time itself should not be consumed by the planning phase.

A significant challenge in devising a sampling plan optimizer is ensuring that the search process remains confined to the space of valid Latin hypercubes. As previously discussed, the defining characteristic of a Latin hypercube X is that each of its columns represents a permutation of the possible levels for the corresponding variable. Consequently, the smallest modification that can be applied to a Latin hypercube—without compromising its crucial multidimensional stratification property—involves swapping two elements within any single column of X . A Python implementation for ‘mutating’ a Latin hypercube through such an operation, generalized to accommodate random changes applied to multiple sites, is provided below:

66.3.5.1. The Function `perturb()`

The function `perturb` randomly swaps elements in a Latin hypercube sampling plan. It takes a 2D array representing the sampling plan and performs a specified number of random element swaps, ensuring that the result remains a valid Latin hypercube.

```
def perturb(X: np.ndarray,
            PertNum: Optional[int] = 1) -> np.ndarray:
    """
    Args:
        X (np.ndarray):
            A 2D array (sampling plan) of shape (n, k),
            where each row is a point
            and each column is a dimension.
        PertNum (int, optional):
            The number of element swaps (perturbations)
            to perform. Defaults to 1.

    Returns:
        np.ndarray:
            The perturbed sampling plan,
            identical in shape to the input, with
            one or more random column swaps executed.
    """
    # Get dimensions of the plan
    n, k = X.shape
    if n < 2 or k < 2:
        raise ValueError("Latin hypercubes require at least 2 points and 2 dimensions")
    for _ in range(PertNum):
        # Pick a random column
        col = int(np.floor(np.random.rand() * k))
        # Pick two distinct row indices
        el1, el2 = 0, 0
        while el1 == el2:
```



```

    e11 = int(np.floor(np.random.rand() * n))
    e12 = int(np.floor(np.random.rand() * n))
    # Swap the two selected elements in the chosen column
    X[e11, col], X[e12, col] = X[e12, col], X[e11, col]
return X

```

Example 66.12 (The Function `perturb()`). The `perturb` function can be used to randomly swap elements in a Latin hypercube sampling plan. The following code demonstrates how to use `perturb` to create a perturbed version of a 4x2 sampling plan:

```

X_original = np.array([[1, 3],[2, 4],[3, 1],[4, 2]])
print("Original Sampling Plan:")
print(X_original)
print("Perturbed Sampling Plan:")
X_perturbed = perturb(X_original, PertNum=1)
print(X_perturbed)

```

Original Sampling Plan:

```

[[1 3]
 [2 4]
 [3 1]
 [4 2]]

```

Perturbed Sampling Plan:

```

[[1 3]
 [2 4]
 [4 1]
 [3 2]]

```

Forrester, Sóbester, and Keane (2008) uses the term ‘mutation’, because this problem lends itself to nature-inspired computation. Morris and Mitchell (1995) use a simulated annealing algorithm, the detailed pseudocode of which can be found in their paper. As an alternative, a method based on evolutionary operation (EVOP) is offered by Forrester, Sóbester, and Keane (2008).

66.3.6. Evolutionary Operation

As introduced by Box (1957), evolutionary operation was designed to optimize chemical processes. The current parameters of the reaction would be recorded in a box at the centre of a board, with a series of ‘offspring’ boxes along the edges containing values of the parameters slightly altered with respect to the central, ‘parent’ values. Once the reaction was completed for all of these sets of variable values and the corresponding yields recorded, the contents of the central box would be replaced with that of the

66. Sampling Plans

setup with the highest yield and this would then become the parent of a new set of peripheral boxes.

This is generally viewed as a local search procedure, though this depends on the mutation step sizes, that is on the differences between the parent box and its offspring. The longer these steps, the more global is the scope of the search.

For the purposes of the Latin hypercube search, a variable scope strategy is applied. The process starts with a long step length (that is a relatively large number of swaps within the columns) and, as the search progresses, the current best basin of attraction is gradually approached by reducing the step length to a single change.

In each generation the parent is mutated (randomly, using the `perturb` function) a pertnum number of times. The sampling plan that yields the smallest Φ_q value (as per the Morris-Mitchell criterion, calculated using `mmphi`) among all offspring and the parent is then selected; in evolutionary computation parlance this selection philosophy is referred to as elitism.

The EVOP based search for space-filling Latin hypercubes is thus a truly evolutionary process: the optimized sampling plan results from the nonrandom survival of random variations.

66.3.7. Putting it all Together

All the pieces of the optimum Latin hypercube sampling process puzzle are now in place: the random hypercube generator as a starting point for the optimization process, the ‘spacefillingness’ metric that needs to be optimized, the optimization engine that performs this task and the comparison function that selects the best of the optima found for the various q ’s. These pieces just need to be put into a sequence. Here is the Python embodiment of the completed puzzle. It results in a function `bestlh` that uses the function `mmlhs` to find the best Latin hypercube sampling plan for a given set of parameters.

66.3.7.1. The Function `mmlhs`

Performs an evolutionary search (using perturbations) to find a Morris-Mitchell optimal Latin hypercube, starting from an initial plan `X_start`.

This function does the following:

1. Initializes a “best” Latin hypercube (`X_best`) from the provided `X_start`.
2. Iteratively perturbs `X_best` to create offspring.
3. Evaluates the space-fillingness of each offspring via the Morris-Mitchell metric (using `mmphi`).
4. Updates the best plan whenever a better offspring is found.

```

def mmlhs(X_start: np.ndarray,
          population: int,
          iterations: int,
          q: Optional[float] = 2.0,
          plot=False) -> np.ndarray:
    """
    Args:
        X_start (np.ndarray):
            A 2D array of shape (n, k) providing the initial Latin hypercube
            (n points in k dimensions).
        population (int):
            Number of offspring to create in each generation.
        iterations (int):
            Total number of generations to run the evolutionary search.
        q (float, optional):
            The exponent used by the Morris-Mitchell space-filling criterion.
            Defaults to 2.0.
        plot (bool, optional):
            If True, a simple scatter plot of the first two dimensions will be
            displayed at each iteration. Only if k >= 2. Defaults to False.

    Returns:
        np.ndarray:
            A 2D array representing the most space-filling Latin hypercube found
            after all iterations, of the same shape as X_start.
    """
    n = X_start.shape[0]
    if n < 2:
        raise ValueError("Latin hypercubes require at least 2 points")
    k = X_start.shape[1]
    if k < 2:
        raise ValueError("Latin hypercubes are not defined for dim k < 2")
    # Initialize best plan and its metric
    X_best = X_start.copy()
    Phi_best = mmphi(X_best, q=q)
    # After 85% of iterations, reduce the mutation rate to 1
    leveloff = int(np.floor(0.85 * iterations))
    for it in range(1, iterations + 1):
        # Decrease number of mutations over time
        if it < leveloff:
            mutations = int(round(1 + (0.5 * n - 1) * (leveloff - it) / (leveloff - 1)))
        else:
            mutations = 1
        X_improved = X_best.copy()

```

```

Phi_improved = Phi_best
# Create offspring, evaluate, and keep the best
for _ in range(population):
    X_try = perturb(X_best.copy(), mutations)
    Phi_try = mmphi(X_try, q=q)

    if Phi_try < Phi_improved:
        X_improved = X_try
        Phi_improved = Phi_try
# Update the global best if we found a better plan
if Phi_improved < Phi_best:
    X_best = X_improved
    Phi_best = Phi_improved
# Simple visualization of the first two dimensions
if plot and (X_best.shape[1] >= 2):
    plt.clf()
    plt.scatter(X_best[:, 0], X_best[:, 1], marker="o")
    plt.grid(True)
    plt.title(f"Iteration {it} - Current Best Plan")
    plt.pause(0.01)
return X_best

```

Example 66.13 (The Function `mmlhs`). The `mmlhs` function can be used to optimize a Latin hypercube sampling plan. The following code demonstrates how to use `mmlhs` to optimize a 4x2 Latin hypercube starting from an initial plan:

```

# Suppose we have an initial 4x2 plan
X_start = np.array([[0.1, 0.3], [.1, .4], [.2, .9], [.9, .2]])
print("Initial plan:")
print(X_start)
# Search for a more space-filling plan
X_opt = mmlhs(X_start, population=10, iterations=100, q=2)
print("Optimized plan:")
print(X_opt)

```

```

Initial plan:
[[0.1 0.3]
 [0.1 0.4]
 [0.2 0.9]
 [0.9 0.2]]
Optimized plan:
[[0.9 0.3]
 [0.1 0.9]

```

```
[0.2 0.4]
[0.1 0.2]]
```

Figure 66.9 shows the initial and optimized plans in 2D. The blue points represent the initial plan, while the red points represent the optimized plan.

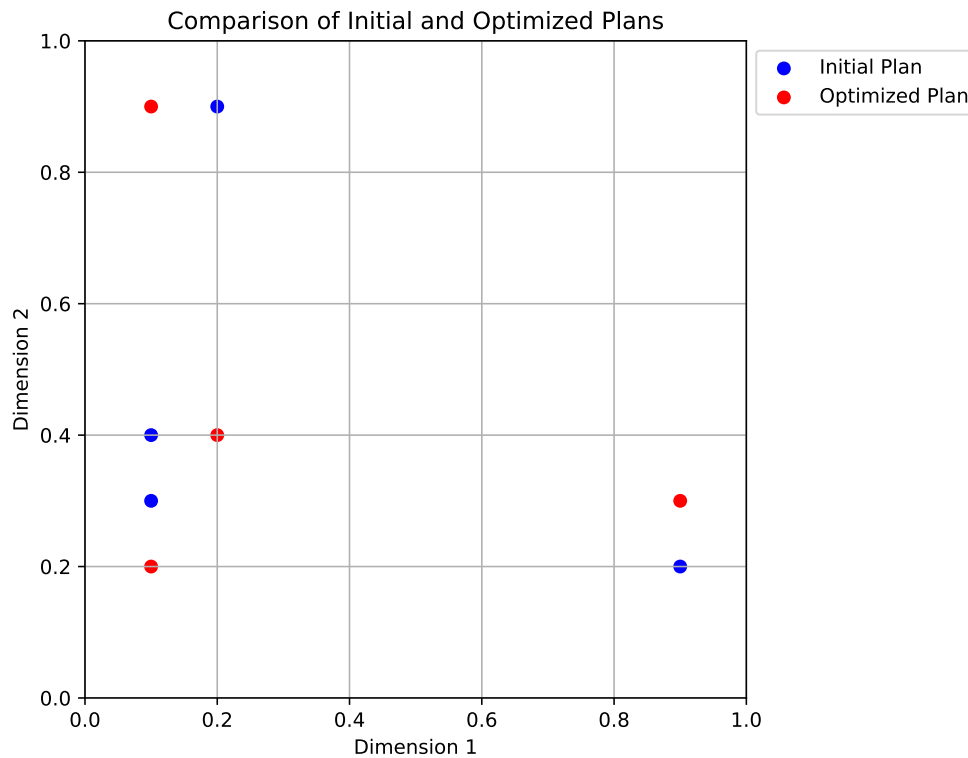


Figure 66.9.: Comparison of the initial and optimized plans in 2D.

66.3.7.2. The Function `bestlh`

Generates an optimized Latin hypercube by evolving the Morris-Mitchell criterion across multiple exponents (q values) and selecting the best plan.

```
def bestlh(n: int,
           k: int,
           population: int,
           iterations: int,
           p=1,
```

```

        plot=False,
        verbosity=0,
        edges=0,
        q_list=[1, 2, 5, 10, 20, 50, 100]) -> np.ndarray:
"""
Args:
    n (int):
        Number of points required in the Latin hypercube.
    k (int):
        Number of design variables (dimensions).
    population (int):
        Number of offspring in each generation of the evolutionary search.
    iterations (int):
        Number of generations for the evolutionary search.
    p (int, optional):
        The distance norm to use. p=1 for Manhattan (L1), p=2 for Euclidean (L2).
        Defaults to 1 (faster than 2).
    plot (bool, optional):
        If True, a scatter plot of the optimized plan in the first two dimensions
        will be displayed. Only if k>=2. Defaults to False.
    verbosity (int, optional):
        Verbosity level. 0 is silent, 1 prints the best q value found. Defaults to 0.
    edges (int, optional):
        If 1, places centers of the extreme bins at the domain edges ([0,1]).
        Otherwise, bins are fully contained within the domain, i.e. midpoints.
        Defaults to 0.
    q_list (list, optional):
        A list of q values to optimize. Defaults to [1, 2, 5, 10, 20, 50, 100].
        These values are used to evaluate the space-fillingness of the Latin
        hypercube. The best plan is selected based on the lowest mmphi value.

Returns:
    np.ndarray:
        A 2D array of shape (n, k) representing an optimized Latin hypercube.
"""
if n < 2:
    raise ValueError("Latin hypercubes require at least 2 points")
if k < 2:
    raise ValueError("Latin hypercubes are not defined for dim k < 2")

# A list of exponents (q) to optimize

# Start with a random Latin hypercube
X_start = rlh(n, k, edges=edges)

```

```

# Allocate a 3D array to store the results for each q
# (shape: (n, k, number_of_q_values))
X3D = np.zeros((n, k, len(q_list)))

# Evolve the plan for each q in q_list
for i, q_val in enumerate(q_list):
    if verbosity > 0:
        print(f"Now optimizing for q={q_val}...")
    X3D[:, :, i] = mmlhs(X_start, population, iterations, q_val)

# Sort the set of evolved plans according to the Morris-Mitchell criterion
index_order = mmsort(X3D, p=p)

# index_order is a 1-based array of plan indices; the first element is the best
best_idx = index_order[0] - 1
if verbosity > 0:
    print(f"Best lh found using q={q_list[best_idx]}...")

# The best plan in 3D array order
X = X3D[:, :, best_idx]

# Plot the first two dimensions
if plot and (k >= 2):
    plt.scatter(X[:, 0], X[:, 1], c="r", marker="o")
    plt.title(f"Morris-Mitchell optimum plan found using q={q_list[best_idx]}")
    plt.xlabel("x_1")
    plt.ylabel("x_2")
    plt.grid(True)
    plt.show()

return X

```

Example 66.14 (The Function `bestlh`). The `bestlh` function can be used to generate an optimized Latin hypercube sampling plan. The following code demonstrates how to use `bestlh` to create a 5x2 Latin hypercube with a population of 5 and 10 iterations:

```
Xbestlh= bestlh(n=5, k=2, population=5, iterations=10)
```

Figure 66.10 shows the best Latin hypercube sampling in 2D. The red points represent the optimized plan.

Sorting all candidate plans in ascending order is not strictly necessary - after all, only the best one is truly of interest. Nonetheless, the added computational complexity is

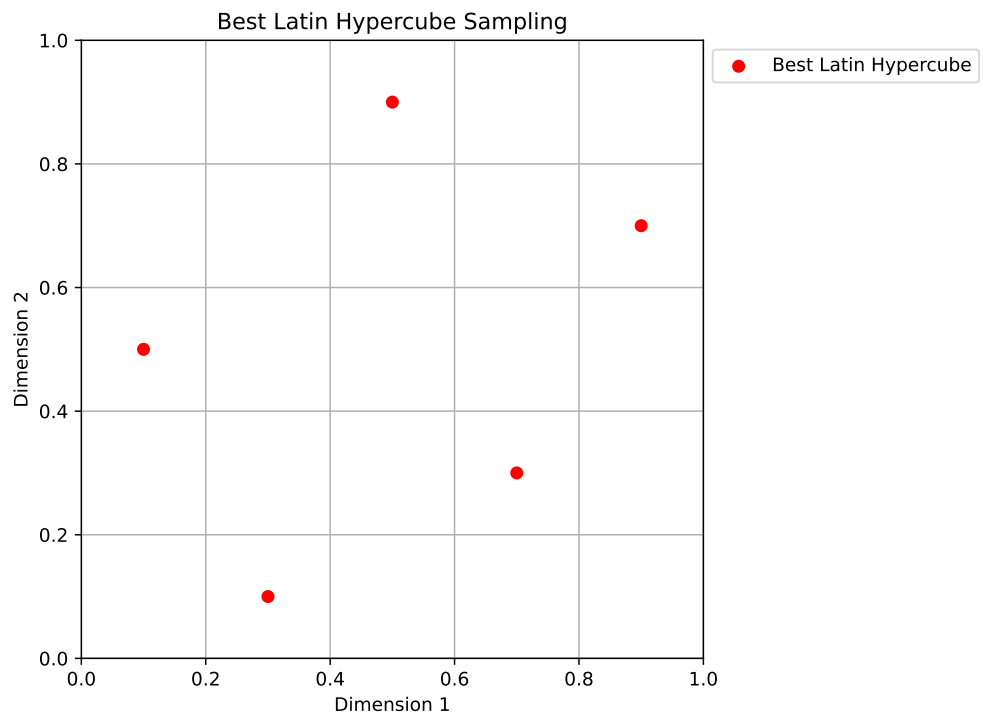


Figure 66.10.: Best Latin Hypercube Sampling

minimal (the vector will only ever contain as many elements as there are candidate q values, and only an index array is sorted, not the actual repository of plans). This sorting gives the reader the opportunity to compare, if desired, how different choices of q influence the resulting plans.

66.4. Experimental Analysis of the Morris-Mitchell Criterion

Morris-Mitchell Criterion Experimental Analysis

- Number of points: 16, Dimensions: 2
- mmphi parameters: q (exponent) = 2.0, p (distance norm) = 2.0 (1=Manhattan, 2=Euclidean)

```
N_POINTS = 16
N_DIM = 2
RANDOM_SEED = 42
q = 2.0
p = 2.0
```

66.4.1. Evaluation of Sampling Designs

We generate various sampling designs and evaluate their space-filling properties using the Morris-Mitchell criterion.

```
designs = {}
if int(np.sqrt(N_POINTS))**2 == N_POINTS:
    grid_design = Grid(k=N_DIM)
    designs["Grid (4x4)"] = grid_design.generate_grid_design(points_per_dim=int(np.sqrt(N_POINTS)))
else:
    print(f"Skipping grid design as N_POINTS={N_POINTS} is not a perfect square for a simple 2D grid")

lhs_design = SpaceFilling(k=N_DIM, seed=42)
designs["LHS"] = lhs_design.generate_qms_lhs_design(n_points=N_POINTS)

sobol_design = Sobol(k=N_DIM, seed=42)
designs["Sobol"] = sobol_design.generate_sobol_design(n_points=N_POINTS)

random_design = Random(k=N_DIM)
designs["Random"] = random_design.uniform(n_points=N_POINTS)
```

66. Sampling Plans

```

poor_design = Poor(k=N_DIM)
designs["Collinear"] = poor_design.generate_collinear_design(n_points=N_POINTS)

clustered_design = Clustered(k=N_DIM)
designs["Clustered (3 clusters)"] = clustered_design.generate_clustered_design(n_points=N_POINTS)

results = {}

print("Calculating Morris-Mitchell metric (smaller is better):")
for name, X_design in designs.items():
    metric_val = mmpm(X_design, q=q, p=p)
    results[name] = metric_val
    print(f" {name}: {metric_val:.4f}")

```

```

Calculating Morris-Mitchell metric (smaller is better):
Grid (4x4): 20.2617
LHS: 28.1868
Sobol: 28.9710
Random: 60.3385
Collinear: 87.8829
Clustered (3 clusters): 90.3702

```

```

if N_DIM == 2:
    num_designs = len(designs)
    cols = 2
    rows = int(np.ceil(num_designs / cols))
    fig, axes = plt.subplots(rows, cols, figsize=(5 * cols, 5 * rows))
    axes = axes.ravel() # Flatten axes array for easy iteration

    for i, (name, X_design) in enumerate(designs.items()):
        ax = axes[i]
        ax.scatter(X_design[:, 0], X_design[:, 1], s=50, edgecolors='k', alpha=0.7)
        ax.set_title(f"{name} \nmmpm = {results[name]:.3f}", fontsize=10)
        ax.set_xlabel("X1")
        ax.set_ylabel("X2")
        ax.set_xlim(-0.05, 1.05)
        ax.set_ylim(-0.05, 1.05)
        ax.set_aspect('equal', adjustable='box')
        ax.grid(True, linestyle='--', alpha=0.6)

    # Hide any unused subplots
    for j in range(i + 1, len(axes)):
        fig.delaxes(axes[j])

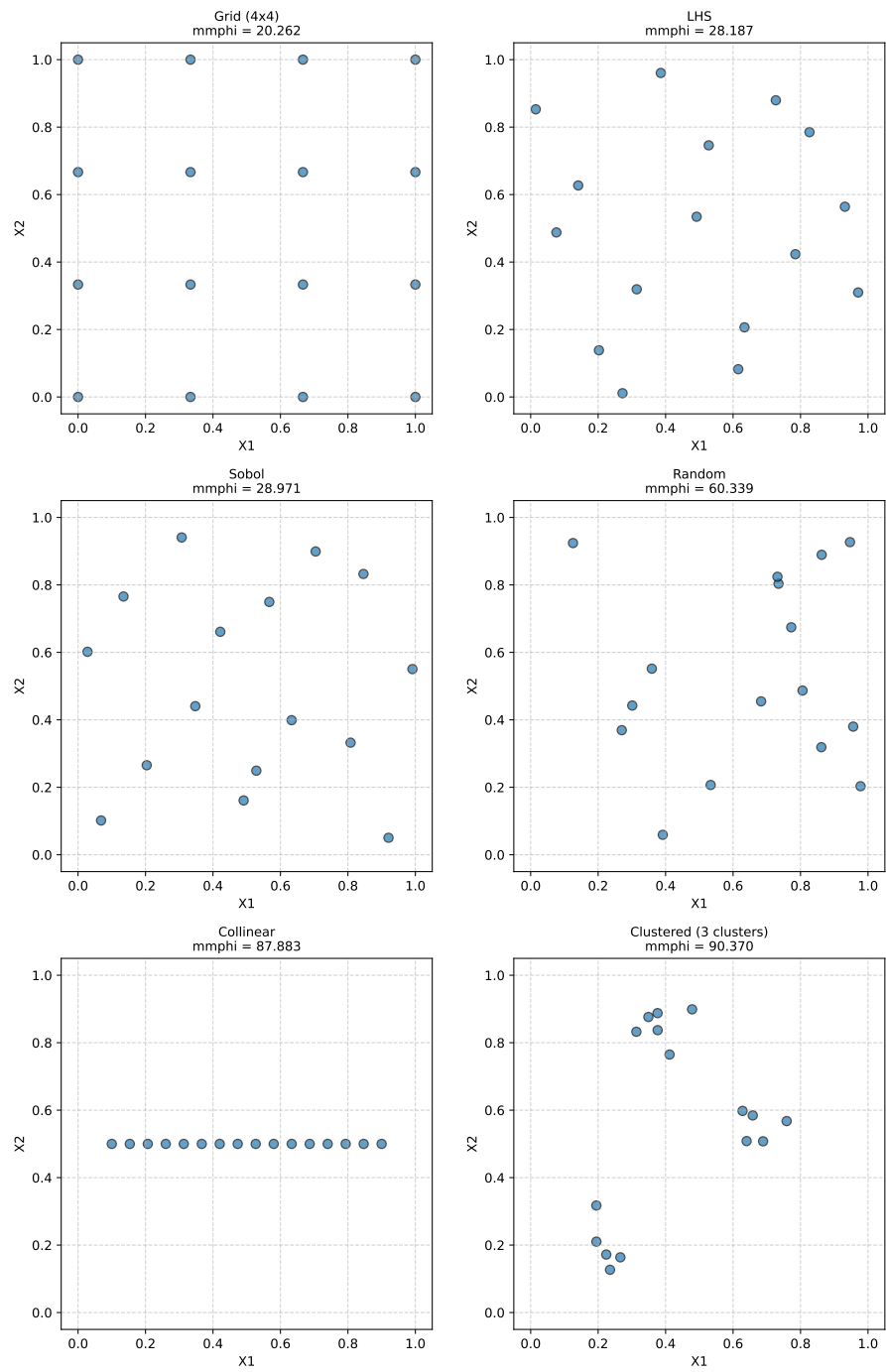
```

66.4. *Experimental Analysis of the Morris-Mitchell Criterion*

```
plt.tight_layout()
plt.suptitle(f"Comparison of 2D Sampling Designs ({N_POINTS} points each)", fontsize=14, y=1.02)
plt.show()
```

66. Sampling Plans

Comparison of 2D Sampling Designs (16 points each)



66.4.2. Demonstrate the Impact of mmphi Parameters

Demonstrating Impact of mmphi Parameters on 'LHS' Design

```
X_lhs = designs["LHS"]

# 1. Default parameters (already calculated)
print(f"  LHS (q={q}, p={p} Euclidean): {results['LHS']:.4f}")

# 2. Change q (main exponent, literature's p or k)
q_high = 15.0
metric_lhs_q_high = mmphi(X_lhs, q=q_high, p=p)
print(f"  LHS (q={q_high}, p={p} Euclidean): {metric_lhs_q_high:.4f} (Higher q penalizes small distances more)")

# 3. Change p (distance norm, literature's q or m)
p_manhattan = 1.0
metric_lhs_p_manhattan = mmphi(X_lhs, q=q, p=p_manhattan)
print(f"  LHS (q={q}, p={p_manhattan} Manhattan): {metric_lhs_p_manhattan:.4f} (Using L1 distance)")
```

```
LHS (q=2.0, p=2.0 Euclidean): 28.1868
LHS (q=15.0, p=2.0 Euclidean): 8.1573 (Higher q penalizes small distances more)
LHS (q=2.0, p=1.0 Manhattan): 22.0336 (Using L1 distance)
```

66.4.3. Morris-Mitchell Criterion: Impact of Adding Points

Impact of adding a point to a 2x2 grid design

```
# Initial 2x2 Grid Design
X_initial = np.array([[0.0, 0.0], [1.0, 0.0], [0.0, 1.0], [1.0, 1.0]])
mmphi_initial = mmphi(X_initial, q=q, p=p)

print(f"Parameters: q (exponent) = {q}, p (distance) = {p} (Euclidean)\n")
print(f"Initial 2x2 Grid Design (4 points):")
print(f"  Points:\n{X_initial}")
print(f"  Morris-Mitchell Criterion (Phi_q): {mmphi_initial:.4f}\n")
```

```
Parameters: q (exponent) = 2.0, p (distance) = 2.0 (Euclidean)
```

```
Initial 2x2 Grid Design (4 points):
```

```
  Points:
[[0. 0.]
 [1. 0.]
```

66. Sampling Plans

```
[0. 1.]
[1. 1.]]
Morris-Mitchell Criterion (Phi_q): 2.2361
```

Scenarios for adding a 5th point:

```
scenarios = {
    "Scenario 1: Add to Center": {
        "new_point": np.array([[0.5, 0.5]]),
        "description": "Adding a point in the center of the grid."
    },
    "Scenario 2: Add Close to Existing (Cluster)": {
        "new_point": np.array([[0.1, 0.1]]),
        "description": "Adding a point very close to an existing point (0,0)."
    },
    "Scenario 3: Add on Edge": {
        "new_point": np.array([[0.5, 0.0]]),
        "description": "Adding a point on an edge between (0,0) and (1,0)."
    }
}

results_summary = []
augmented_designs_for_plotting = {"Initial Design": X_initial}

for name, scenario_details in scenarios.items():
    new_point = scenario_details["new_point"]
    X_augmented = np.vstack((X_initial, new_point))
    augmented_designs_for_plotting[name] = X_augmented

    mmphi_augmented = mmphi(X_augmented, q=q, p=p)
    change = mmphi_augmented - mmphi_initial

    print(f"{name}:")
    print(f"  Description: {scenario_details['description']}")
    print(f"  New Point Added: {new_point}")
    # print(f"  Augmented Design (5 points):\n{X_augmented}") # Optional: print full n
    print(f"  Morris-Mitchell Criterion (Phi_q): {mmphi_augmented:.4f}")
    print(f"  Change from Initial Phi_q: {change:+.4f}\n")

    results_summary.append({
        "Scenario": name,
        "Initial Phi_q": mmphi_initial,
        "Augmented Phi_q": mmphi_augmented,
        "Change": change
    })
```

Scenario 1: Add to Center:

Description: Adding a point in the center of the grid.

New Point Added: $[[0.5 \ 0.5]]$

Morris-Mitchell Criterion (Φ_q): 3.6056

Change from Initial Φ_q : +1.3695

Scenario 2: Add Close to Existing (Cluster):

Description: Adding a point very close to an existing point (0,0).

New Point Added: $[[0.1 \ 0.1]]$

Morris-Mitchell Criterion (Φ_q): 7.6195

Change from Initial Φ_q : +5.3834

Scenario 3: Add on Edge:

Description: Adding a point on an edge between (0,0) and (1,0).

New Point Added: $[[0.5 \ 0. \]]$

Morris-Mitchell Criterion (Φ_q): 3.8210

Change from Initial Φ_q : +1.5849

```
num_designs = len(augmented_designs_for_plotting)
cols = 2
rows = int(np.ceil(num_designs / cols))

fig, axes = plt.subplots(rows, cols, figsize=(6 * cols, 5 * rows))
axes = axes.ravel()

plot_idx = 0
# Plot initial design first
ax = axes[plot_idx]
ax.scatter(X_initial[:, 0], X_initial[:, 1], s=100, edgecolors='k', alpha=0.7, label="Original Point")
ax.set_title(f"Initial Design\nPhi_q = {mmphi_initial:.3f}", fontsize=10)
ax.set_xlabel("X1")
ax.set_ylabel("X2")
ax.set_xlim(-0.1, 1.1)
ax.set_ylim(-0.1, 1.1)
ax.set_aspect('equal', adjustable='box')
ax.grid(True, linestyle='--', alpha=0.6)
ax.legend(fontsize='small')
plot_idx += 1

# Plot augmented designs
for name, X_design in augmented_designs_for_plotting.items():
    if name == "Initial Design":
```

```

        continue # Already plotted

    ax = axes[plot_idx]
    # Highlight original vs new point
    original_points = X_design[:-1, :]
    new_point = X_design[-1, :].reshape(1,2)

    ax.scatter(original_points[:, 0], original_points[:, 1], s=100, edgecolors='k', a
    ax.scatter(new_point[:, 0], new_point[:, 1], s=150, color='red', edgecolors='k', r

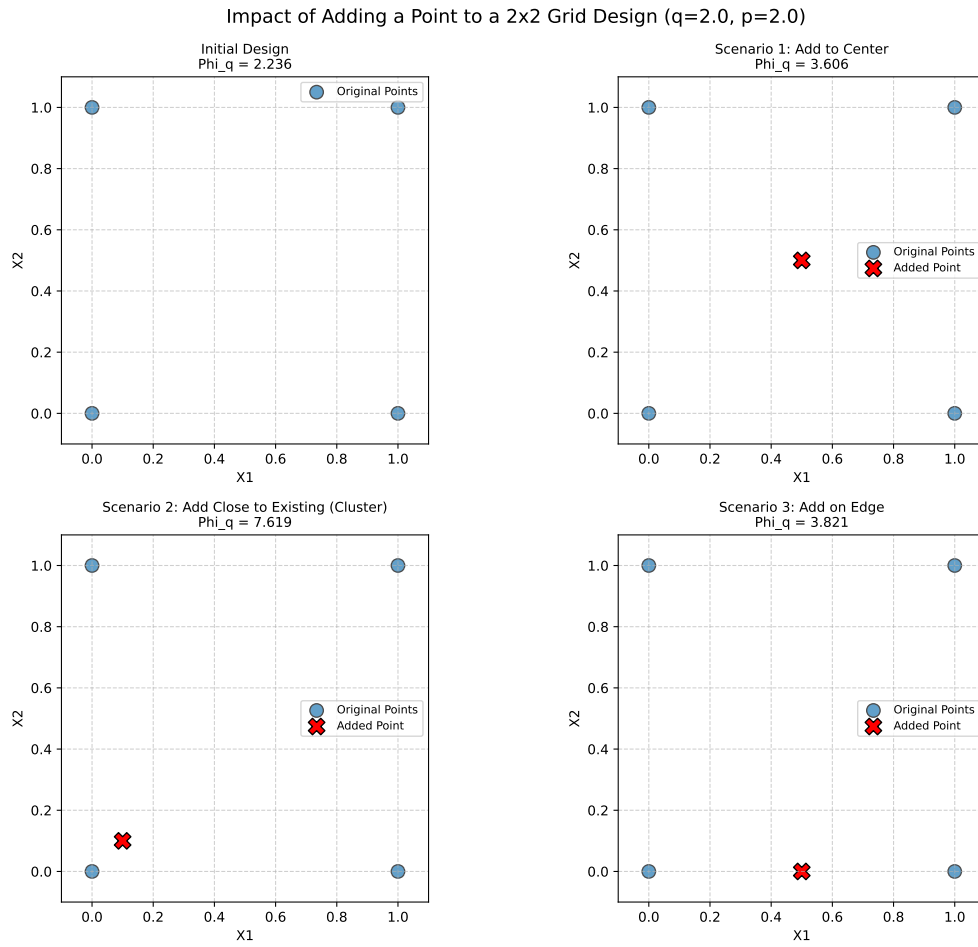
    current_phi_q = next(item['Augmented Phi_q'] for item in results_summary if item[
    ax.set_title(f"{name}\nPhi_q = {current_phi_q:.3f}", fontsize=10)
    ax.set_xlabel("X1")
    ax.set_ylabel("X2")
    ax.set_xlim(-0.1, 1.1)
    ax.set_ylim(-0.1, 1.1)
    ax.set_aspect('equal', adjustable='box')
    ax.grid(True, linestyle='--', alpha=0.6)
    ax.legend(fontsize='small')
    plot_idx +=1

# Hide any unused subplots
for j in range(plot_idx, len(axes)):
    fig.delaxes(axes[j])

plt.tight_layout(rect=[0, 0, 1, 0.96]) # Adjust layout to make space for supitle
plt.suptitle(f"Impact of Adding a Point to a 2x2 Grid Design (q={q}, p={p})", fontsize
plt.show()

```


66.4. Experimental Analysis of the Morris-Mitchell Criterion



Summary Table (Conceptual):

Scenario	Initial Phi_q	Augmented Phi_q	Change
Baseline (2x2 Grid)	2.236	—	—
Scenario 1: Add to Center	2.236	3.606	+1.369
Scenario 2: Add Close to Existing (Cluster)	2.236	7.619	+5.383
Scenario 3: Add on Edge	2.236	3.821	+1.585

66.5. A Sample-Size Invariant Version of the Morris-Mitchell Criterion

66.5.1. Comparison of `mmphi()` and `mmphi_intensive()`

The Morris-Mitchell criterion is a widely used metric for evaluating the space-filling properties of Latin hypercube sampling designs. However, it is sensitive to the number of points in the design, which can lead to misleading comparisons between designs with different sample sizes. To address this issue, a sample-size invariant version of the Morris-Mitchell criterion has been proposed. It is available in the `spotpython` package as `mmphi_intensive()`, see [SOURCE].

The functions `mmphi()` and `mmphi_intensive()` both calculate a Morris-Mitchell criterion, but they differ in their normalization, which makes `mmphi_intensive()` invariant to the sample size.

Let X be a sampling plan with n points $\{x_1, x_2, \dots, x_n\}$ in a k -dimensional space. Let $d_{ij} = \|x_i - x_j\|_p$ be the p -norm distance between points x_i and x_j . Let J_l be the multiplicity of the l -th unique distance d_l among all pairs of points in X . Let m be the total number of unique distances.

1. `mmphi()` (Morris-Mitchell Criterion Φ_q)

The `mmphi()` function, as defined in the context and implemented in `sampling.py`, calculates the Morris-Mitchell criterion Φ_q as:

$$\Phi_q(X) = \left(\sum_{l=1}^m J_l d_l^{-q} \right)^{1/q},$$

where:

- J_l is the number of pairs of points separated by the unique distance d_l .
- d_l are the unique pairwise distances.
- q is a user-defined exponent (typically $q > 0$).

This formulation is directly based on the sum of inverse powers of distances. The value of Φ_q is generally dependent on the number of points n in the design X , as the sum $\sum J_l d_l^{-q}$ will typically increase with more points (and thus more pairs).

2. `mmphi_intensive()` (Intensive Morris-Mitchell Criterion)

The `mmphi_intensive()` function, as implemented in `sampling.py` calculates a sample-size invariant version of the Morris-Mitchell criterion, which will be referred to as Φ_q^I . The formula is:

66.5. A Sample-Size Invariant Version of the Morris-Mitchell Criterion

$$\Phi_q^I(X) = \left(\frac{1}{M} \sum_{l=1}^m J_l d_l^{-q} \right)^{1/q}$$

where:

- $M = \binom{n}{2} = \frac{n(n-1)}{2}$ is the total number of unique pairs of points in the design X .
- The other terms J_l , d_l , q are the same as in `mmphi()`.

The key mathematical difference is the normalization factor $\frac{1}{M}$ inside the parentheses before the outer exponent $1/q$ is applied.

- `mmphi()`: Calculates $(\text{SumTerm})^{1/q}$, where $\text{SumTerm} = \sum J_l d_l^{-q}$.
- `mmphi_intensive()`: Calculates $\left(\frac{\text{SumTerm}}{M}\right)^{1/q}$.

By dividing the sum $\sum J_l d_l^{-q}$ by M (the total number of pairs), `mmphi_intensive()` effectively calculates an *average* contribution per pair to the $-q$ -th power of distance, before taking the q -th root. This normalization makes the criterion less dependent on the absolute number of points n and allows for more meaningful comparisons of space-fillingness between designs of different sizes. A smaller value indicates a better (more space-filling) design for both criteria.

66.5.2. Plotting the Two Morris-Mitchell Criteria for Different Sample Sizes

Figure 66.11 shows the comparison of the two Morris-Mitchell criteria for different sample sizes using the `plot_mmphi_vs_n_lhs` function. The red line represents the standard Morris-Mitchell criterion, while the blue line represents the sample-size invariant version. Note the difference in the y-axis scales, which highlights how the sample-size invariant version remains consistent across varying sample sizes.

```
def plot_mmphi_vs_n_lhs(k_dim: int,
                        seed: int,
                        n_min: int = 10,
                        n_max: int = 100,
                        n_step: int = 5,
                        q_phi: float = 2.0,
                        p_phi: float = 2.0):
    """
    Generates LHS designs for varying n, calculates mmphi and mmphi_intensive,
    and plots them against the number of samples (n).

    Args:
        k_dim (int): Number of dimensions for the LHS design.
```

```

seed (int): Random seed for reproducibility.
n_min (int): Minimum number of samples.
n_max (int): Maximum number of samples.
n_step (int): Step size for increasing n.
q_phi (float): Exponent q for the Morris-Mitchell criteria.
p_phi (float): Distance norm p for the Morris-Mitchell criteria.
"""
n_values = list(range(n_min, n_max + 1, n_step))
if not n_values:
    print("Warning: n_values list is empty. Check n_min, n_max, and n_step.")
    return
mmphi_results = []
mmphi_intensive_results = []
lhs_generator = SpaceFilling(k=k_dim, seed=seed)
print(f"Calculating for n from {n_min} to {n_max} with step {n_step}...")
for n_points in n_values:
    if n_points < 2 : # mmphi requires at least 2 points to calculate distances
        print(f"Skipping n={n_points} as it's less than 2.")
        mmphi_results.append(np.nan)
        mmphi_intensive_results.append(np.nan)
        continue
    try:
        X_design = lhs_generator.generate_qms_lhs_design(n_points=n_points)
        phi = mmphi(X_design, q=q_phi, p=p_phi)
        phi_intensive, _, _ = mmphi_intensive(X_design, q=q_phi, p=p_phi)
        mmphi_results.append(phi)
        mmphi_intensive_results.append(phi_intensive)
    except Exception as e:
        print(f"Error calculating for n={n_points}: {e}")
        mmphi_results.append(np.nan)
        mmphi_intensive_results.append(np.nan)

fig, ax1 = plt.subplots(figsize=(9, 6))

color = 'tab:red'
ax1.set_xlabel('Number of Samples (n)')
ax1.set_ylabel('mmphi (Phiq)', color=color)
ax1.plot(n_values, mmphi_results, color=color, marker='o', linestyle='-', label='n')
ax1.tick_params(axis='y', labelcolor=color)
ax1.grid(True, linestyle='--', alpha=0.7)

ax2 = ax1.twinx() # instantiate a second axes that shares the same x-axis
color = 'tab:blue'
ax2.set_ylabel('mmphi_intensive (PhiqI)', color=color) # we already handled the x-axis

```

66.5. A Sample-Size Invariant Version of the Morris-Mitchell Criterion

```
ax2.plot(n_values, mmphi_intensive_results, color=color, marker='x', linestyle='--', label='mmphi_intensive')
ax2.tick_params(axis='y', labelcolor=color)

fig.tight_layout() # otherwise the right y-label is slightly clipped
plt.title(f'Morris-Mitchell Criteria vs. Number of Samples (n)\nLHS (k={k_dim}, q={q_phi}, p={p_phi})')
# Add legends
lines, labels = ax1.get_legend_handles_labels()
lines2, labels2 = ax2.get_legend_handles_labels()
ax2.legend(lines + lines2, labels + labels2, loc='best')
plt.show()
```

```
N_DIM = 2
RANDOM_SEED = 42
plot_mmphi_vs_n_lhs(k_dim=N_DIM, seed=RANDOM_SEED, n_min=10, n_max=100, n_step=5)
```

Calculating for n from 10 to 100 with step 5...

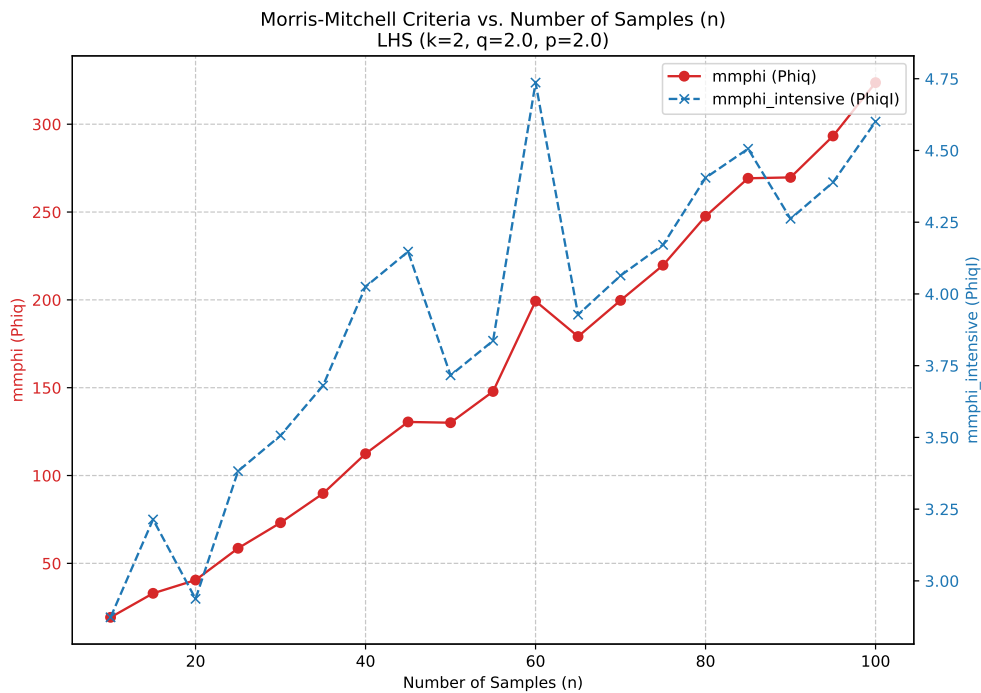


Figure 66.11.: Comparison of the two Morris-Mitchell Criteria for Different Sample Sizes

66.6. Jupyter Notebook

Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

67. Constructing a Surrogate

Note

This section is based on chapter 2 in Forrester, Sóbester, and Keane (2008).

Definition 67.1 (Black Box Problem). We are trying to learn a mapping that converts the vector $\vec{x}$ into a scalar output y , i.e., we are trying to learn a function

$$y = f(x).$$

If function is hidden (“lives in a black box”), so that the physics of the problem is not known, the problem is called a black box problem.

This black box could take the form of either a physical or computer experiment, for example, a finite element code, which calculates the maximum stress (σ) for given product dimensions ($\vec{x}$).

Definition 67.2 (Generic Solution). The generic solution method is to collect the output values $y^{(1)}, y^{(2)}, \dots, y^{(n)}$ that result from a set of inputs $\vec{x}^{(1)}, \vec{x}^{(2)}, \dots, \vec{x}^{(n)}$ and find a best guess $\hat{f}(\vec{x})$ for the black box mapping f , based on these known observations.

67.1. Stage One: Preparing the Data and Choosing a Modelling Approach

The first step is the identification, through a small number of observations, of the inputs that have a significant impact on f ; that is the determination of the shortest design variable vector $\vec{x} = \{x_1, x_2, \dots, x_k\}^T$ that, by sweeping the ranges of all of its variables, can still elicit most of the behavior the black box is capable of. The ranges of the various design variables also have to be established at this stage.

The second step is to recruit n of these k -vectors into a list

$$X = \{\vec{x}^{(1)}, \vec{x}^{(2)}, \dots, \vec{x}^{(n)}\}^T,$$

where each $\vec{x}^{(i)}$ is a k -vector. The corresponding responses are collected in a vector such that this represents the design space as thoroughly as possible.

67. Constructing a Surrogate

In the surrogate modeling process, the number of samples n is often limited, as it is constrained by the computational cost (money and/or time) associated with obtaining each observation.

It is advisable to scale $\vec{x}$ at this stage into the unit cube $[0, 1]^k$, a step that can simplify the subsequent mathematics and prevent multidimensional scaling issues.

We now focus on the attempt to learn f through data pairs

$$\{(\vec{x}^{(1)}, y^{(1)}), (\vec{x}^{(2)}, y^{(2)}), \dots, (\vec{x}^{(n)}, y^{(n)})\}.$$

This supervised learning process essentially involves searching across the space of possible functions $\hat{f}$ that would replicate observations of f . This space of functions is infinite. Any number of hypersurfaces could be drawn to pass through or near the known observations, accounting for experimental error. However, most of these would generalize poorly; they would be practically useless at predicting responses at new sites, which is the ultimate goal.

Example 67.1 (The Needle(s) in the Haystack Function). An extreme example is the ‘needle(s) in the haystack’ function:

$$f(x) = \begin{cases} y^{(1)}, & \text{if } x = \vec{x}^{(1)} \\ y^{(2)}, & \text{if } x = \vec{x}^{(2)} \\ \vdots & \\ y^{(n)}, & \text{if } x = \vec{x}^{(n)} \\ 0, & \text{otherwise.} \end{cases}$$

While this predictor reproduces all training data, it seems counter-intuitive and unsettling to predict 0 everywhere else for most engineering functions. Although there is a small chance that the function genuinely resembles the equation above and we sampled exactly where the needles are, it is highly unlikely.

There are countless other configurations, perhaps less contrived, that still generalize poorly. This suggests a need for systematic means to filter out nonsensical predictors. In our approach, we embed the structure of f into the model selection algorithm and search over its parameters to fine-tune the approximation to observations. For instance, consider one of the simplest models,

$$f(x, \vec{w}) = \vec{w}^T \vec{x} + v. \quad (67.1)$$

Learning f with this model implies that its structure—a hyperplane—is predetermined, and the fitting process involves finding the $k + 1$ parameters (the slope vector $\vec{w}$ and the intercept v) that best fit the data. This will be accomplished in Stage Two.

Complicating this further is the noise present in observed responses (we assume design vectors $\vec{x}$ are not corrupted). Here, we focus on learning from such data, which sometimes risks overfitting.

67.2. Stage Two: Parameter Estimation and Training

Definition 67.3 (Overfitting). Overfitting occurs when the model becomes too flexible and captures not only the underlying trend but also the noise in the data.

In the surrogate modeling process, the second stage as described in Section 67.2, addresses this issue of complexity control by estimating the parameters of the fixed structure model. However, foresight is necessary even at the model type selection stage.

Model selection often involves physics-based considerations, where the modeling technique is chosen based on expected underlying responses.

Example 67.2 (Model Selection). Modeling stress in an elastically deformed solid due to small strains may justify using a simple linear approximation. Without insights into the physics, and if one fails to account for the simplicity of the data, a more complex and excessively flexible model may be incorrectly chosen. Although parameter estimation might still adjust the approximation to become linear, an opportunity to develop a simpler and robust model may be lost.

- Simple linear (or polynomial) models, despite their lack of flexibility, have advantages like applicability in further symbolic computations.
- Conversely, if we incorrectly assume a quadratic process when multiple peaks and troughs exist, the parameter estimation stage will not compensate for an unsuitable model choice. A quadratic model is too rigid to fit a multimodal function, regardless of parameter adjustments.

67.2. Stage Two: Parameter Estimation and Training

Assuming that Stage One helped identify the k critical design variables, acquire the learning data set, and select a generic model structure $f(\vec{x}, \vec{w})$, the task now is to estimate parameters $\vec{w}$ to ensure the model fits the data optimally. Among several estimation criteria, we will discuss two methods here.

Definition 67.4 (Maximum Likelihood Estimation). Given a set of parameters $\vec{w}$, the model $f(\vec{x}, \vec{w})$ allows computation of the probability of the data set

$$\{(\vec{x}^{(1)}, y^{(1)} \pm \epsilon), (\vec{x}^{(2)}, y^{(2)} \pm \epsilon), \dots, (\vec{x}^{(n)}, y^{(n)} \pm \epsilon)\}$$

resulting from f (where ϵ is a small error margin around each data point).

i Maximum Likelihood Estimation

?@sec-max-likelihood presents a more detailed discussion of the maximum likelihood estimation (MLE) method.

67. Constructing a Surrogate

Taking **eq-likelihood-mvn** and assuming errors ϵ are independently and normally distributed with standard deviation σ , the probability of the data set is given by:

$$P = \frac{1}{(2\pi\sigma^2)^{n/2}} \exp \left[-\frac{1}{2\sigma^2} \sum_{i=1}^n (y^{(i)} - f(\vec{x}^{(i)}, \vec{w}))^2 \epsilon \right].$$

Intuitively, this is equivalent to the likelihood of the parameters given the data. Accepting this intuitive relationship as a mathematical one aids in model parameter estimation. This is achieved by maximizing the likelihood or, more conveniently, minimizing the negative of its natural logarithm:

$$\min_{\vec{w}} \sum_{i=1}^n \frac{[y^{(i)} - f(\vec{x}^{(i)}, \vec{w})]^2}{2\sigma^2} + \frac{n}{2} \ln \epsilon. \quad (67.2)$$

If we assume σ and ϵ are constants, Equation 67.2 simplifies to the well-known least squares criterion:

$$\min_{\vec{w}} \sum_{i=1}^n [y^{(i)} - f(\vec{x}^{(i)}, \vec{w})]^2.$$

Cross-validation is another method used to estimate model performance.

Definition 67.5 (Cross-Validation). Cross-validation splits the data randomly into q roughly equal subsets, and then cyclically removing each subset and fitting the model to the remaining $q-1$ subsets. A loss function L is then computed to measure the error between the predictor and the withheld subset for each iteration, with contributions summed over all q iterations. More formally, if a mapping $\theta : \{1, \dots, n\} \rightarrow \{1, \dots, q\}$ describes the allocation of the n training points to one of the q subsets and $f^{(-\theta(i))}(\vec{x})$ is the predicted value by removing the subset $\theta(i)$ (i.e., the subset where observation i belongs), the cross-validation measure, used as an estimate of prediction error, is:

$$CV = \frac{1}{n} \sum_{i=1}^n L(y^{(i)}, f^{(-\theta(i))}(\vec{x}^{(i)})). \quad (67.3)$$

Introducing the squared error as the loss function and considering our generic model f still dependent on undetermined parameters, we write Equation 67.3 as:

$$CV = \frac{1}{n} \sum_{i=1}^n [y^{(i)} - f^{(-\theta(i))}(\vec{x}^{(i)})]^2. \quad (67.4)$$

The extent to which Equation 67.4 is an unbiased estimator of true risk depends on q . It is shown that if $q = n$, the leave-one-out cross-validation (LOOCV) measure is

almost unbiased. However, LOOCV can have high variance because subsets are very similar. Hastie, Tibshirani, and Friedman (2017)) suggest using compromise values like $q = 5$ or $q = 10$. Using fewer subsets also reduces the computational cost of the cross-validation process, see also Arlot, Celisse, et al. (2010) and Kohavi (1995).

67.3. Stage Three: Model Testing

If there is a sufficient amount of observational data, a random subset should be set aside initially for model testing. Hastie, Tibshirani, and Friedman (2017) recommend setting aside approximately $0.25n$ of $\vec{x} \rightarrow y$ pairs for testing purposes. These observations must remain untouched during Stages One and Two, as their sole purpose is to evaluate the testing error—the difference between true and approximated function values at the test sites—once the model has been built. Interestingly, if the main goal is to construct an initial surrogate for seeding a global refinement criterion-based strategy (as discussed in Section 3.2 in Forrester, Sóbester, and Keane (2008)), the model testing phase might be skipped.

It is noted that, ideally, parameter estimation (Stage Two) should also rely on a separate subset. However, observational data is rarely abundant enough to afford this luxury (if the function is cheap to evaluate and evaluation sites are selectable, a surrogate model might not be necessary).

When data are available for model testing and the primary objective is a globally accurate model, using either a root mean square error (RMSE) metric or the correlation coefficient (r^2) is recommended. To test the model, a test data set of size n_t is used alongside predictions at the corresponding locations to calculate these metrics.

The RMSE is defined as follows:

Definition 67.6 (Root Mean Square Error (RMSE)).

$$\text{RMSE} = \sqrt{\frac{1}{n_t} \sum_{i=1}^{n_t} (y^{(i)} - \hat{y}^{(i)})^2},$$

Ideally, the RMSE should be minimized, acknowledging its limitation by errors in the objective function f calculation. If the error level is known, like a standard deviation, the aim might be to achieve an RMSE within this value. Often, the target is an RMSE within a specific percentage of the observed data's objective value range.

The squared correlation coefficient r , see [?@eq-pears-corr](#), between the observed y and predicted $\hat{y}$ values can be computed as:

$$r^2 = \left(\frac{\text{cov}(y, \hat{y})}{\sqrt{\text{var}(y)\text{var}(\hat{y})}} \right)^2, \quad (67.5)$$

67. Constructing a Surrogate

Equation 67.5 and can be expanded as:

$$r^2 = \left(\frac{n_t \sum_{i=1}^{n_t} y^{(i)} \hat{y}^{(i)} - \sum_{i=1}^{n_t} y^{(i)} \sum_{i=1}^{n_t} \hat{y}^{(i)}}{\sqrt{\left(n_t \sum_{i=1}^{n_t} (y^{(i)})^2 - \left(\sum_{i=1}^{n_t} y^{(i)} \right)^2 \right) \left(n_t \sum_{i=1}^{n_t} (\hat{y}^{(i)})^2 - \left(\sum_{i=1}^{n_t} \hat{y}^{(i)} \right)^2 \right)}} \right)^2.$$

The correlation coefficient r^2 does not require scaling the data sets and only compares landscape shapes, not values. An $r^2 > 0.8$ typically indicates a surrogate with good predictive capability.

The methods outlined provide quantitative assessments of model accuracy, yet visual evaluations can also be insightful. In general, the RMSE will not reach zero but will stabilize around a low value. At this point, the surrogate model is saturated with data, and further additions do not enhance the model globally (though local improvements can occur at newly added points if using an interpolating model).

Example 67.3 (The Tea and Sugar Analogy). Forrester, Söbester, and Keane (2008) illustrates this saturation point using a comparison with a cup of tea and sugar. The tea represents the surrogate model, and sugar represents data. Initially, the tea is unsweetened, and adding sugar increases its sweetness. Eventually, a saturation point is reached where no more sugar dissolves, and the tea cannot get any sweeter. Similarly, a more flexible model, like one with additional parameters or employing interpolation rather than regression, can increase the saturation point—akin to making a hotter cup of tea for dissolving more sugar.

67.4. Jupyter Notebook

Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

68. Response Surface Methods

This part deals with numerical implementations of optimization methods. The goal is to understand the implementation of optimization methods and to solve real-world problems numerically and efficiently. We will focus on the implementation of surrogate models, because they are the most efficient way to solve real-world problems.

Starting point is the well-established response surface methodology (RSM). It will be extended to the design and analysis of computer experiments (DACE). The DACE methodology is a modern extension of the response surface methodology. It is based on the use of surrogate models, which are used to replace the real-world problem with a simpler problem. The simpler problem is then solved numerically. The solution of the simpler problem is then used to solve the real-world problem.

! Numerical methods: Goals

- Understand implementation of optimization methods
- Solve real-world problems numerically and efficiently

68.1. What is RSM?

Response Surface Methods (RSM) refer to a collection of statistical and mathematical tools that are valuable for developing, improving, and optimizing processes. The overarching theme of RSM involves studying how input variables that control a product or process can potentially influence a response that measures performance or quality characteristics.

The advantages of RSM include a rich literature, well-established methods often used in manufacturing, the importance of careful experimental design combined with a well-understood model, and the potential to add significant value to scientific inquiry, process refinement, optimization, and more. However, there are also drawbacks to RSM, such as the use of simple and crude surrogates, the hands-on nature of the methods, and the limitation of local methods.

RSM is related to various fields, including Design of Experiments (DoE), quality management, reliability, and productivity. Its applications are widespread in industry and manufacturing, focusing on designing, developing, and formulating new products and improving existing ones, as well as from laboratory research. RSM is commonly

applied in domains such as materials science, manufacturing, applied chemistry, climate science, and many others.

An example of RSM involves studying the relationship between a response variable, such as yield (y) in a chemical process, and two process variables: reaction time (ξ_1) and reaction temperature (ξ_2). The provided code illustrates this scenario, following a variation of the so-called “banana function.”

In the context of visualization, RSM offers the choice between 3D plots and contour plots. In a 3D plot, the independent variables ξ_1 and ξ_2 are represented, with y as the dependent variable.

```
import numpy as np
import matplotlib.pyplot as plt

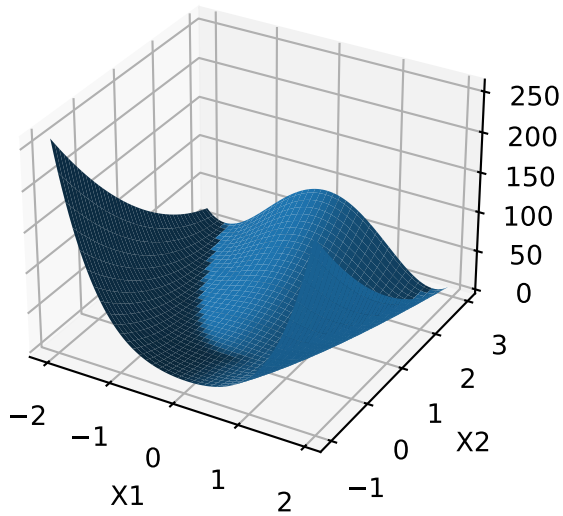
def fun_rosen(x1, x2):
    b = 10
    return (x1-1)**2 + b*(x2-x1**2)**2

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
x = np.arange(-2.0, 2.0, 0.05)
y = np.arange(-1.0, 3.0, 0.05)
X, Y = np.meshgrid(x, y)
zs = np.array(fun_rosen(np.ravel(X), np.ravel(Y)))
Z = zs.reshape(X.shape)

ax.plot_surface(X, Y, Z)

ax.set_xlabel('X1')
ax.set_ylabel('X2')
ax.set_zlabel('Y')

plt.show()
```



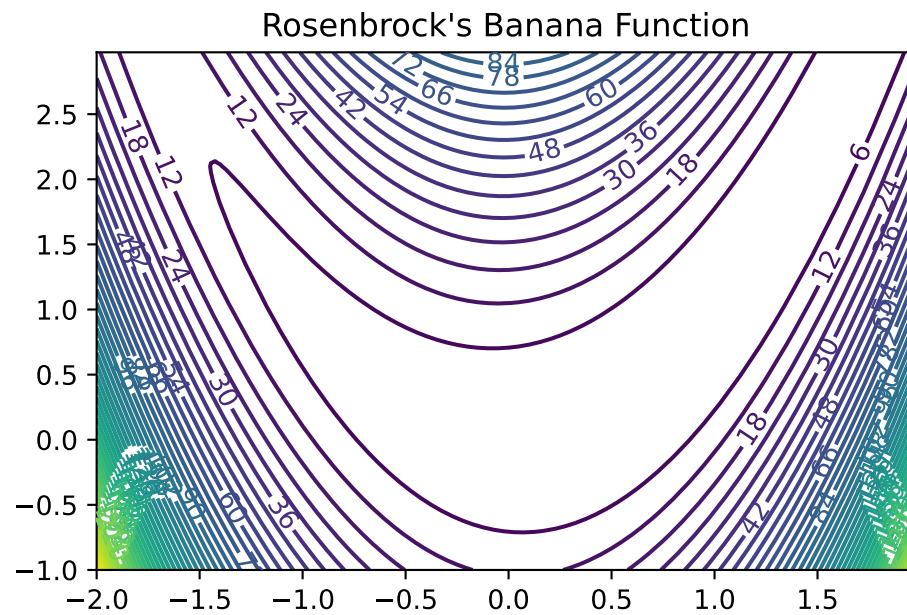
- contour plot example:

- x_1 and x_2 are the independent variables
- y is the dependent variable

```
import numpy as np
import matplotlib.cm as cm
import matplotlib.pyplot as plt

delta = 0.025
x1 = np.arange(-2.0, 2.0, delta)
x2 = np.arange(-1.0, 3.0, delta)
X1, X2 = np.meshgrid(x1, x2)
Y = fun_rosen(X1, X2)
fig, ax = plt.subplots()
CS = ax.contour(X1, X2, Y, 50)
ax.clabel(CS, inline=True, fontsize=10)
ax.set_title("Rosenbrock's Banana Function")
```

```
Text(0.5, 1.0, "Rosenbrock's Banana Function")
```



- Visual inspection: yield is optimized near (ξ_1, ξ_2)

68.1.1. Visualization: Problems in Practice

- True response surface is unknown in practice
- When yield evaluation is not as simple as a toy banana function, but a process requiring care to monitor, reconfigure and run, it's far too expensive to observe over a dense grid
- And, measuring yield may be a noisy/inexact process
- That's where stats (RSM) comes in

68.1.2. RSM: Strategies

- RSMs consist of experimental strategies for
- **exploring** the space of the process (i.e., independent/input) variables (above ξ_1 and ξ_2)
- empirical statistical **modeling** targeted toward development of an appropriate approximating relationship between the response (yield) and process variables local to a study region of interest

- **optimization** methods for sequential refinement in search of the levels or values of process variables that produce desirable responses (e.g., that maximize yield or explain variation)
- RSM used for fitting an Empirical Model
- True response surface driven by an unknown physical mechanism
- Observations corrupted by noise
- Helpful: fit an empirical model to output collected under different process configurations
- Consider response Y that depends on controllable input variables $\xi_1, \xi_2, \dots, \xi_m$
- RSM: Equations of the Empirical Model
 - $Y = f(\xi_1, \xi_2, \dots, \xi_m) + \epsilon$
 - $\mathbb{E}\{Y\} = \eta = f(\xi_1, \xi_2, \dots, \xi_m)$
 - ϵ is treated as zero mean idiosyncratic noise possibly representing
 - * inherent variation, or
 - * the effect of other systems or
 - * variables not under our purview at this time

68.1.3. RSM: Noise in the Empirical Model

- Typical simplifying assumption: $\epsilon \sim N(0, \sigma^2)$
- We seek estimates for f and σ^2 from noisy observations Y at inputs ξ

68.1.4. RSM: Natural and Coded Variables

- Inputs $\xi_1, \xi_2, \dots, \xi_m$ called **natural variables**:
 - expressed in natural units of measurement, e.g., degrees Celsius, pounds per square inch (psi), etc.
- Transformed to **coded variables** $x_1, x_2, \dots, x_m$:
 - to mitigate hassles and confusion that can arise when working with a multitude of scales of measurement
- Typical **Transformations** offering dimensionless inputs $x_1, x_2, \dots, x_m$
 - in the unit cube, or
 - scaled to have a mean of zero and standard deviation of one, are common choices.
- Empirical model becomes $\eta = f(x_1, x_2, \dots, x_m)$

68.1.5. RSM Low-order Polynomials

- Low-order polynomial make the following simplifying Assumptions
 - Learning about f is lots easier if we make some simplifying approximations
 - Appealing to **Taylor's theorem**, a low-order polynomial in a small, localized region of the input (x) space is one way forward
 - Classical RSM:
 - * disciplined application of **local analysis** and
 - * **sequential refinement** of locality through conservative extrapolation
 - Inherently a **hands-on process**

68.2. First-Order Models (Main Effects Model)

- **First-order model** (sometimes called main effects model) useful in parts of the input space where it's believed that there's little curvature in f :

$$\eta = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

- For example:

$$\eta = 50 + 8x_1 + 3x_2$$

- In practice, such a surface would be obtained by fitting a model to the outcome of a designed experiment
- First-Order Model in python Evaluated on a Grid
- Evaluate model on a grid in a double-unit square centered at the origin
- Coded units are chosen arbitrarily, although one can imagine deploying this approximating function nearby $x^{(0)} = (0, 0)$

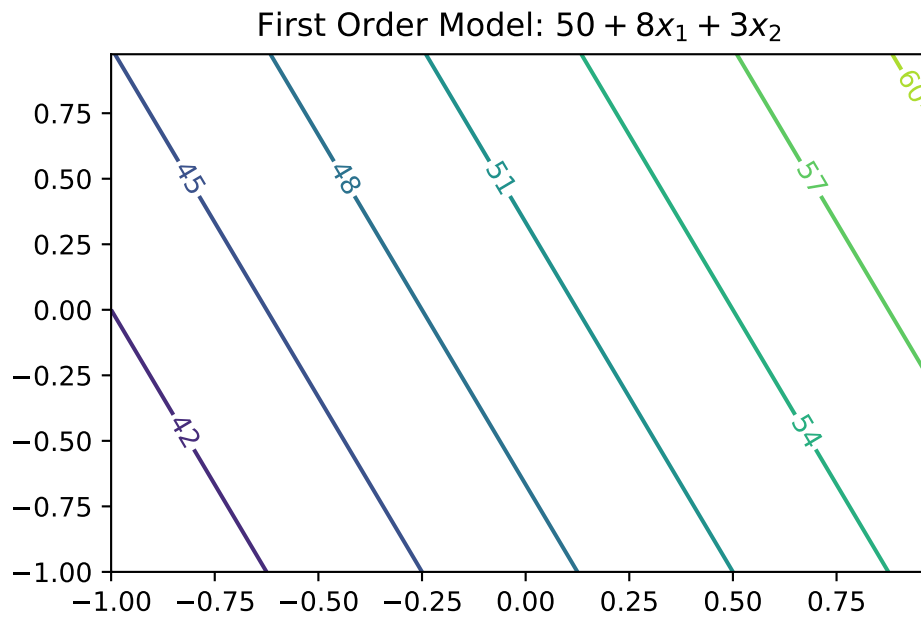
```
def fun_1(x1,x2):
    return 50 + 8*x1 + 3*x2
```

```
import numpy as np
import matplotlib.cm as cm
import matplotlib.pyplot as plt

delta = 0.025
x1 = np.arange(-1.0, 1.0, delta)
x2 = np.arange(-1.0, 1.0, delta)
X1, X2 = np.meshgrid(x1, x2)
Y = fun_1(X1,X2)
fig, ax = plt.subplots()
CS = ax.contour(X1, X2, Y)
ax.clabel(CS, inline=True, fontsize=10)
ax.set_title('First Order Model: $50 + 8x_1 + 3x_2$')
```

68.2. First-Order Models (Main Effects Model)

```
Text(0.5, 1.0, 'First Order Model: $50 + 8x_1 + 3x_2$')
```



68.2.1. First-Order Model Properties

- First-order model in 2d traces out a **plane** in $y \times (x_1, x_2)$ space
- Only be appropriate for the most trivial of response surfaces, even when applied in a highly localized part of the input space
- Adding **curvature** is key to most applications:
 - First-order model with **interactions** induces limited degree of curvature via different rates of change of y as x_1 is varied for fixed x_2 , and vice versa:

$$\eta = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_{12} x_{12}$$

- For example $\eta = 50 + 8x_1 + 3x_2 - 4x_1x_2$

68.2.2. First-order Model with Interactions in python

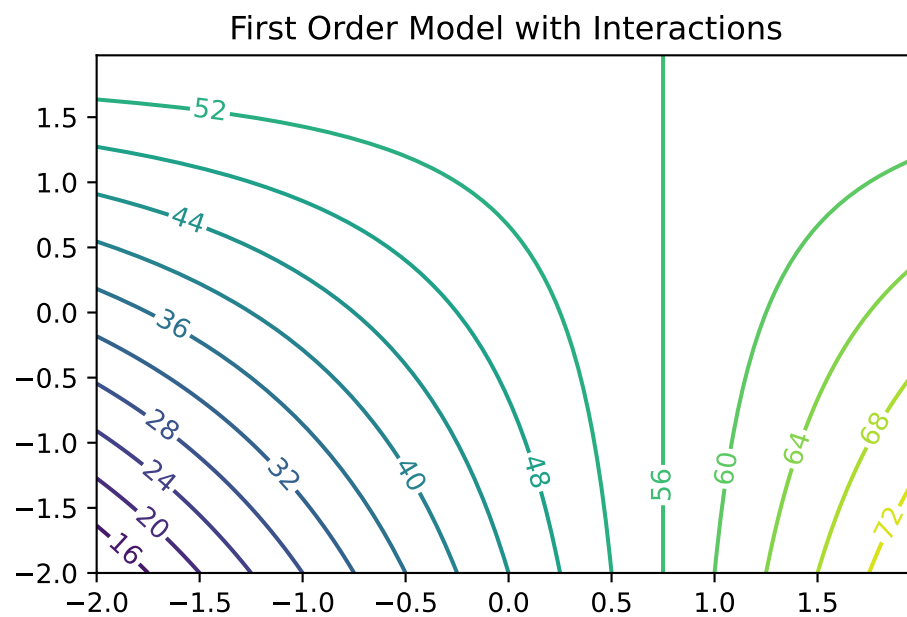
- Code below facilitates evaluations for pairs (x_1, x_2)
- Responses may be observed over a mesh in the same double-unit square

68. Response Surface Methods

```
def fun_11(x1,x2):  
    return 50 + 8 * x1 + 3 * x2 - 4 * x1 * x2
```

```
import numpy as np  
import matplotlib.cm as cm  
import matplotlib.pyplot as plt  
  
delta = 0.025  
x1 = np.arange(-2.0, 2.0, delta)  
x2 = np.arange(-2.0, 2.0, delta)  
X1, X2 = np.meshgrid(x1, x2)  
Y = fun_11(X1,X2)  
fig, ax = plt.subplots()  
CS = ax.contour(X1, X2, Y, 20)  
ax.clabel(CS, inline=True, fontsize=10)  
ax.set_title('First Order Model with Interactions')
```

```
Text(0.5, 1.0, 'First Order Model with Interactions')
```



68.2.3. Observations: First-Order Model with Interactions

- Mean response η is increasing marginally in both x_1 and x_2 , or conditional on a fixed value of the other until x_1 is 0.75
- Rate of increase slows as both coordinates grow simultaneously since the coefficient in front of the interaction term x_1x_2 is negative
- Compared to the first-order model (without interactions): surface is far more useful locally
- Least squares regressions often flag up significant interactions when fit to data collected on a design far from local optima

68.3. Second-Order Models

- Second-order model may be appropriate near local optima where f would have substantial curvature:

$$\eta = \beta_0 + \beta_1x_1 + \beta_2x_2 + \beta_{11}x_1^2 + \beta_{22}x_2^2 + \beta_{12}x_1x_2$$

- For example

$$\eta = 50 + 8x_1 + 3x_2 - 7x_1^2 - 3x_2^2 - 4x_1x_2$$

- Implementation of the Second-Order Model as `fun_2()`.

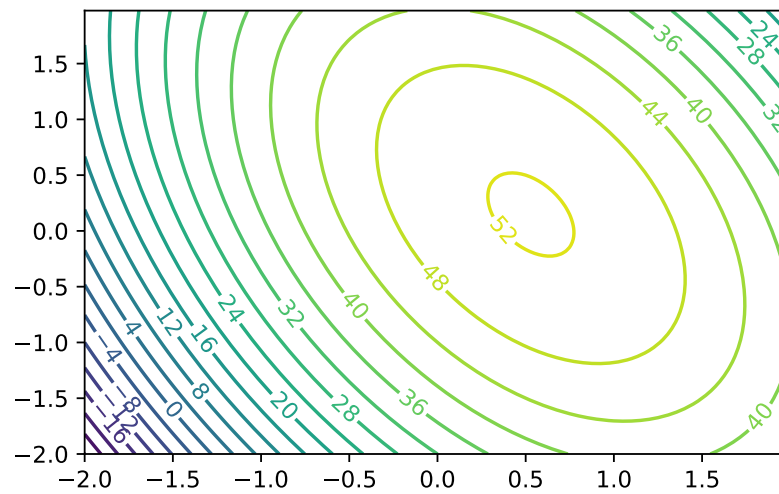
```
def fun_2(x1,x2):
    return 50 + 8 * x1 + 3 * x2 - 7 * x1**2 - 3*x2**2 - 4 * x1 * x2
```

```
import numpy as np
import matplotlib.cm as cm
import matplotlib.pyplot as plt

delta = 0.025
x1 = np.arange(-2.0, 2.0, delta)
x2 = np.arange(-2.0, 2.0, delta)
X1, X2 = np.meshgrid(x1, x2)
Y = fun_2(X1,X2)
fig, ax = plt.subplots()
CS = ax.contour(X1, X2, Y, 20)
ax.clabel(CS, inline=True, fontsize=10)
ax.set_title('Second Order Model with Interactions. Maximum near about $(0.6,0.2)$')
```

```
Text(0.5, 1.0, 'Second Order Model with Interactions. Maximum near about $(0.6,0.2)$')
```

Second Order Model with Interactions. Maximum near about (0.6, 0.2)



68.3.1. Second-Order Models: Properties

- Not all second-order models would have a single stationary point (in RSM jargon called “a simple maximum”)
- In “yield maximizing” setting we’re presuming response surface is **concave** down from a global viewpoint
 - even though local dynamics may be more nuanced
- Exact criteria depend upon the eigenvalues of a certain matrix built from those coefficients
- Box and Draper (2007) provide a diagram categorizing all of the kinds of second-order surfaces in RSM analysis, where finding local maxima is the goal

68.3.2. Example: Stationary Ridge

- Example set of coefficients describing what’s called a **stationary ridge** is provided by the code below

```
def fun_ridge(x1, x2):
    return 80 + 4*x1 + 8*x2 - 3*x1**2 - 12*x2**2 - 12*x1*x2
```

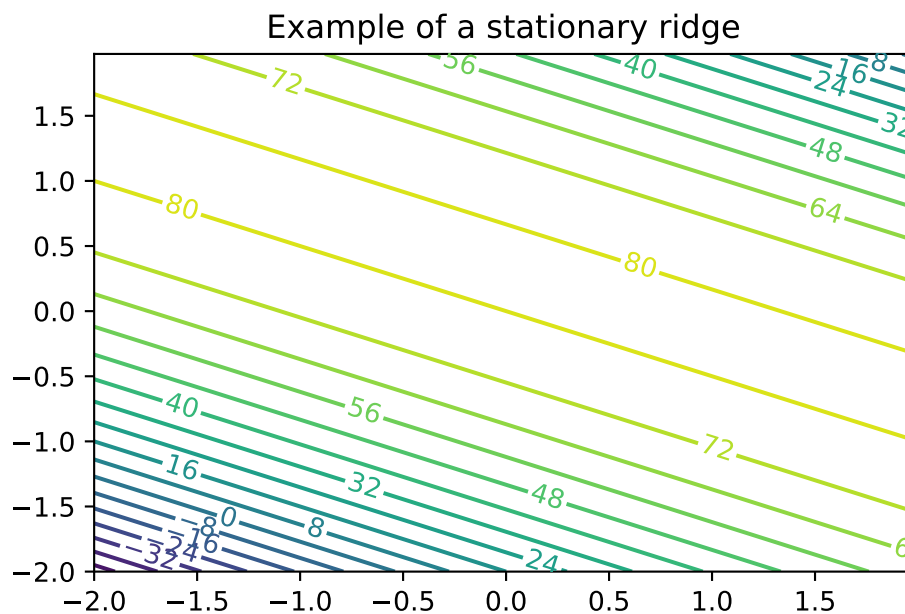
```

import numpy as np
import matplotlib.cm as cm
import matplotlib.pyplot as plt

delta = 0.025
x1 = np.arange(-2.0, 2.0, delta)
x2 = np.arange(-2.0, 2.0, delta)
X1, X2 = np.meshgrid(x1, x2)
Y = fun_ridge(X1, X2)
fig, ax = plt.subplots()
CS = ax.contour(X1, X2, Y, 20)
ax.clabel(CS, inline=True, fontsize=10)
ax.set_title('Example of a stationary ridge')

```

```
Text(0.5, 1.0, 'Example of a stationary ridge')
```



68.3.3. Observations: Second-Order Model (Ridge)

- **Ridge:** a whole line of stationary points corresponding to maxima
- Situation means that the practitioner has some flexibility when it comes to optimizing:

- can choose the precise setting of (x_1, x_2) either arbitrarily or (more commonly) by consulting some tertiary criteria

68.3.4. Example: Rising Ridge

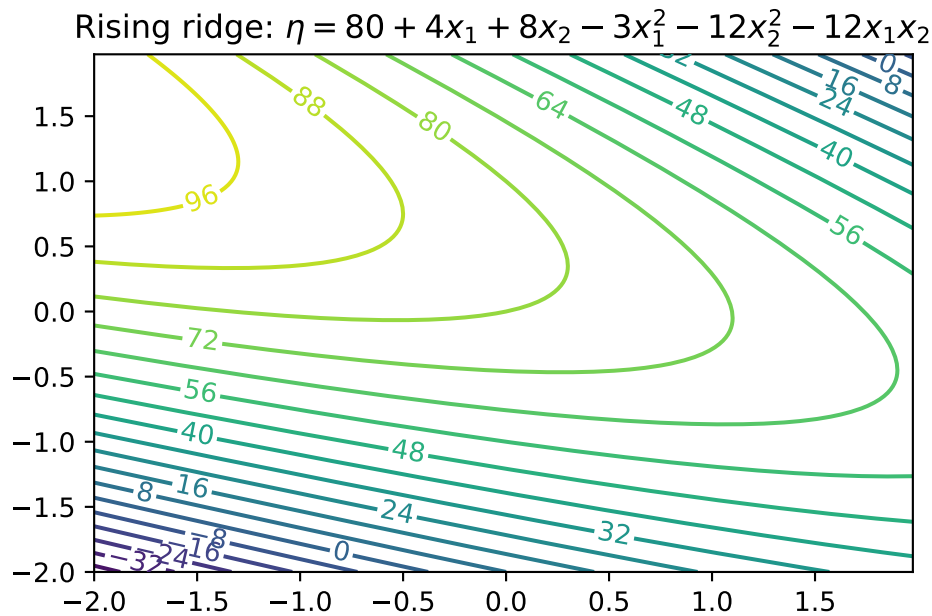
- An example of a rising ridge is implemented by the code below.

```
def fun_ridge_rise(x1, x2):
    return 80 - 4*x1 + 12*x2 - 3*x1**2 - 12*x2**2 - 12*x1*x2
```

```
import numpy as np
import matplotlib.cm as cm
import matplotlib.pyplot as plt

delta = 0.025
x1 = np.arange(-2.0, 2.0, delta)
x2 = np.arange(-2.0, 2.0, delta)
X1, X2 = np.meshgrid(x1, x2)
Y = fun_ridge_rise(X1, X2)
fig, ax = plt.subplots()
CS = ax.contour(X1, X2, Y, 20)
ax.clabel(CS, inline=True, fontsize=10)
ax.set_title('Rising ridge:  $\eta = 80 + 4x_1 + 8x_2 - 3x_1^2 - 12x_2^2 - 12x_1x_2$ ')
```

```
Text(0.5, 1.0, 'Rising ridge:  $\eta = 80 + 4x_1 + 8x_2 - 3x_1^2 - 12x_2^2 - 12x_1x_2$ ')
```

68.3.5. Summary: Rising Ridge

- The stationary point is remote to the study region
- Ccontinuum of (local) stationary points along any line going through the 2d space, excepting one that lies directly on the ridge
- Although estimated response will increase while moving along the axis of symmetry toward its stationary point, this situation indicates
 - either a poor fit by the approximating second-order function, or
 - that the study region is not yet precisely in the vicinity of a local optima—often both.

68.3.6. Falling Ridge

- Inversion of a rising ridge is a falling ridge
- Similarly indicating one is far from local optima, except that the response decreases as you move toward the stationary point
- Finding a falling ridge system can be a back-to-the-drawing-board affair.

68.3.7. Saddle Point

- Finally, we can get what's called a saddle or minimax system.

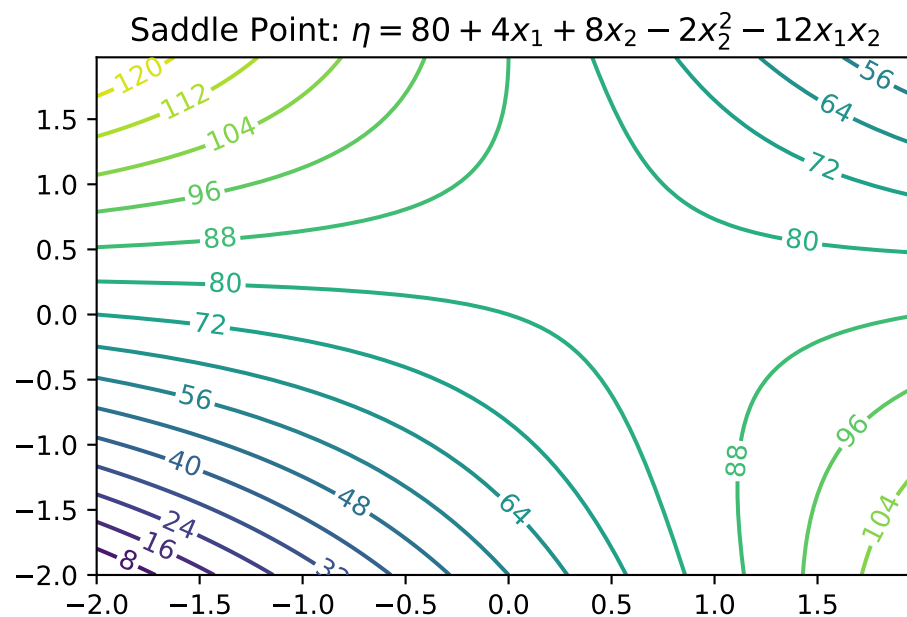
68. Response Surface Methods

```
def fun_saddle(x1, x2):
    return 80 + 4*x1 + 8*x2 - 2*x2**2 - 12*x1*x2

import numpy as np
import matplotlib.cm as cm
import matplotlib.pyplot as plt

delta = 0.025
x1 = np.arange(-2.0, 2.0, delta)
x2 = np.arange(-2.0, 2.0, delta)
X1, X2 = np.meshgrid(x1, x2)
Y = fun_saddle(X1, X2)
fig, ax = plt.subplots()
CS = ax.contour(X1, X2, Y, 20)
ax.clabel(CS, inline=True, fontsize=10)
ax.set_title('Saddle Point:  $\eta = 80 + 4x_1 + 8x_2 - 2x_2^2 - 12x_1x_2$ ')
```

```
Text(0.5, 1.0, 'Saddle Point:  $\eta = 80 + 4x_1 + 8x_2 - 2x_2^2 - 12x_1x_2$ ')
```



68.3.8. Interpretation: Saddle Points

- Likely further data collection, and/or outside expertise, is needed before determining a course of action in this situation

68.3.9. Summary: Ridge Analysis

- Finding a simple maximum, or stationary ridge, represents ideals in the spectrum of second-order approximating functions
- But getting there can be a bit of a slog
- Using models fitted from data means uncertainty due to noise, and therefore uncertainty in the type of fitted second-order model
- A ridge analysis attempts to offer a principled approach to navigating uncertainties when one is seeking local maxima
- The two-dimensional setting exemplified above is convenient for visualization, but rare in practice
- Complications compound when studying the effect of more than two process variables

68.4. General RSM Models

- General **first-order model** on m process variables $x_1, x_2, \dots, x_m$ is

$$\eta = \beta_0 + \beta_1 x_1 + \dots + \beta_m x_m$$

- General **second-order model** on m process variables

$$\eta = \beta_0 + \sum_{j=1}^m \beta_j x_j + \sum_{j=1}^m \beta_{jj} x_j^2 + \sum_{j=2}^m \sum_{k=1}^{j-1} \beta_{jk} x_j x_k.$$

68.4.1. Ordinary Least Squares

- Inference from data is carried out by **ordinary least squares** (OLS)
- For an excellent review including R examples, see Sheather (2009)
- OLS and maximum likelihood estimators (MLEs) are in the typical Gaussian linear modeling setup basically equivalent

68.5. General Linear Regression

We are considering a model, which can be written in the form

$$Y = X\beta + \epsilon,$$

where Y is an $(n \times 1)$ vector of observations (responses), X is an $(n \times p)$ matrix of known form, β is a $(1 \times p)$ vector of unknown parameters, and ϵ is an $(n \times 1)$ vector of errors. Furthermore, $E(\epsilon) = 0$, $Var(\epsilon) = \sigma^2 I$ and the ϵ_i are uncorrelated.

Using the normal equations

$$(X'X)b = X'Y,$$

the solution is given by

$$b = (X'X)^{-1}X'Y.$$

Example 68.1 (Linear Regression).

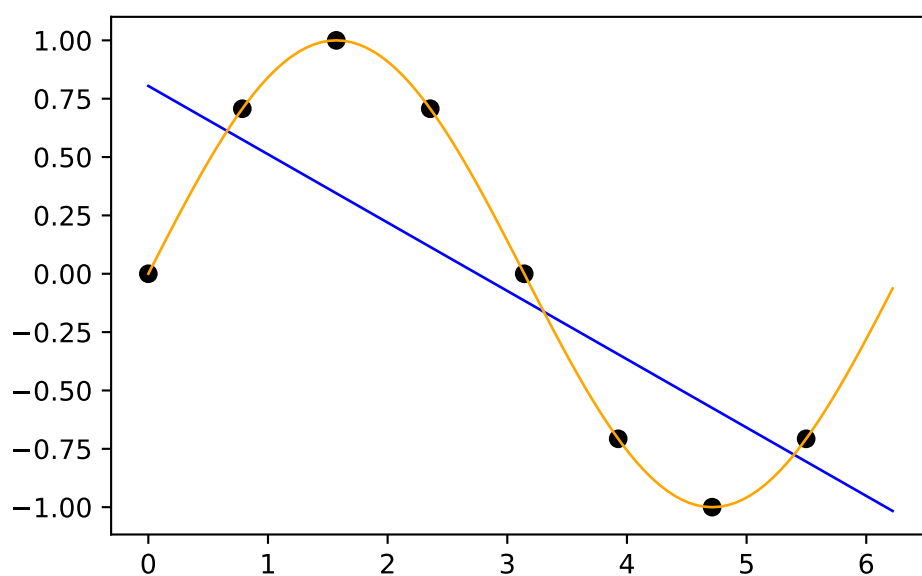
```
import numpy as np
n = 8
X = np.linspace(0, 2*np.pi, n, endpoint=False).reshape(-1,1)
print(np.round(X, 2))
y = np.sin(X)
print(np.round(y, 2))
# fit an OLS model to the data, predict the response based on the 100 x values
m = 100
x = np.linspace(0, 2*np.pi, m, endpoint=False).reshape(-1,1)
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(X, y)
y_pred = model.predict(x)
# visualize the data and the fitted model
import matplotlib.pyplot as plt
plt.scatter(X, y, color='black')
plt.plot(x, y_pred, color='blue', linewidth=1)
# add the ground truth (sine function) in orange
plt.plot(x, np.sin(x), color='orange', linewidth=1)
plt.show()
```

```
[[0.  ]
 [0.79]
 [1.57]
 [2.36]
```

```

[3.14]
[3.93]
[4.71]
[5.5 ]]
[[ 0. ]
 [ 0.71]
 [ 1.  ]
 [ 0.71]
 [ 0.  ]
 [-0.71]
 [-1.  ]
 [-0.71]]

```



68.6. Designs

- Important: Organize the data collection phase of a response surface study carefully
- **Design:** choice of x 's where we plan to observe y 's, for the purpose of approximating f
- Analyses and designs need to be carefully matched
- When using a first-order model, some designs are preferred over others
- When using a second-order model to capture curvature, a different sort of design is appropriate

68. Response Surface Methods

- Design choices often contain features enabling modeling assumptions to be challenged
 - e.g., to check if initial impressions are supported by the data ultimately collected

68.6.1. Different Designs

- **Screening designs:** determine which variables matter so that subsequent experiments may be smaller and/or more focused
- Then there are designs tailored to the form of model (first- or second-order, say) in the screened variables
- And then there are more designs still

68.7. RSM Experimentation

68.7.1. First Step

- RSM-based experimentation begins with a **first-order model**, possibly with interactions
- Presumption: current process operating **far from optimal** conditions
- Collect data and apply **method of steepest ascent** (gradient) on fitted surfaces to move to the optimum

68.7.2. Second Step

- Eventually, if all goes well after several such carefully iterated refinements, **second-order models** are used on appropriate designs in order to zero-in on ideal operating conditions
- Careful analysis of the fitted surface:
 - Ridge analysis with further refinement using gradients of, and
 - standard errors associated with, the fitted surfaces, and so on

68.7.3. Third Step

- Once the practitioner is satisfied with the full arc of
 - design(s),
 - fit(s), and
 - decision(s):

- A small experiment called **confirmation test** may be performed to check if the predicted optimal settings are realizable in practice

68.8. RSM: Review and General Considerations

- First Glimpse, RSM seems sensible, and pretty straightforward as quantitative statistics-based analysis goes
- But: RSM can get complicated, especially when input dimensions are not very low
- Design considerations are particularly nuanced, since the goal is to obtain reliable estimates of main effects, interaction, and curvature while minimizing sampling effort/expense
- RSM Downside: Inefficiency
 - Despite intuitive appeal, several RSM downsides become apparent upon reflection
 - Problems in practice
 - Stepwise nature of sequential decision making is inefficient:
 - * Not obvious how to re-use or update analysis from earlier phases, or couple with data from other sources/related experiments
- RSM Downside: Locality
 - In addition to being local in experiment-time (stepwise approach), it's local in experiment-space
 - Balance between
 - * exploration (maybe we're barking up the wrong tree) and
 - * exploitation (let's make things a little better) is modest at best
- RSM Downside: Expert Knowledge
 - Interjection of expert knowledge is limited to hunches about relevant variables (i.e., the screening phase), where to initialize search, how to design the experiments
 - Yet at the same time classical RSMs rely heavily on constant examination throughout stages of modeling and design and on the instincts of seasoned practitioners
- RSM Downside: Replicability
 - Parallel analyses, conducted according to the same best intentions, rarely lead to the same designs, model fits and so on
 - Sometimes that means they lead to different conclusions, which can be cause for concern

68.8.1. Historical Considerations about RSM

- In spite of those criticisms, however, there was historically little impetus to revise the status quo
- Classical RSM was comfortable in its skin, consistently led to improvements or compelling evidence that none can reasonably be expected
- But then in the late 20th century came an explosive expansion in computational capability, and with it a means of addressing many of those downsides

68.8.2. Status Quo

- Nowadays, field experiments and statistical models, designs and optimizations are coupled with mathematical models
- Simple equations are not regarded as sufficient to describe real-world systems anymore
- Physicists figured that out fifty years ago; industrial engineers followed, biologists, social scientists, climate scientists and weather forecasters, etc.
- Systems of equations are required, solved over meshes (e.g., finite elements), or stochastically interacting agents
- Goals for those simulation experiments are as diverse as their underlying dynamics
- Optimization of systems is common, e.g., to identify worst-case scenarios

68.8.3. The Role of Statistics

- Solving systems of equations, or interacting agents, requires computing
- Statistics involved at various stages:
 - choosing the mathematical model
 - solving by stochastic simulation (Monte Carlo)
 - designing the computer experiment
 - smoothing over idiosyncrasies or noise
 - finding optimal conditions, or
 - calibrating mathematical/computer models to data from field experiments

68.8.4. New RSM is needed: DACE

- Classical RSMs are not well-suited to any of those tasks, because
 - they lack the fidelity required to model these data
 - their intended application is too local
 - they're also too hands-on.

- Once computers are involved, a natural inclination is to automate—to remove humans from the loop and set the computer running on the analysis in order to maximize computing throughput, or minimize idle time
- **Design and Analysis of Computer Experiments** as a modern extension of RSM
- Experimentation is changing due to advances in machine learning
- **Gaussian process** (GP) regression is the canonical surrogate model
- Origins in geostatistics (gold mining)
- Wide applicability in contexts where prediction is king
- Machine learners exposed GPs as powerful predictors for all sorts of tasks:
 - from regression to classification,
 - active learning/sequential design,
 - reinforcement learning and optimization,
 - latent variable modeling, and so on

68.9. Exercises

1. Generate 3d Plots for the Contour Plots in this notebook.
2. Write a `plot_3d` function, that takes the objective function `fun` as an argument.
 - It should provide the following interface: `plot_3d(fun)`.
3. Write a `plot_contour` function, that takes the objective function `fun` as an argument:
 - It should provide the following interface: `plot_contour(fun)`.
4. Consider further arguments that might be useful for both function, e.g., ranges, size, etc.

68.10. Jupyter Notebook

Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

69. Polynomial Models

Note

- This section is based on chapter 2.2 in Forrester, Sóbester, and Keane (2008).
- The following Python packages are imported:

```
import numpy as np
import matplotlib.pyplot as plt
```

69.1. Fitting a Polynomial

We will consider one-variable cases, i.e., $k = 1$, first.

Let us consider the scalar-valued function $f : \mathbb{R} \rightarrow \mathbb{R}$ observed according to the sampling plan $X = \{x^{(1)}, x^{(2)} \dots, x^{(n)}\}^T$, yielding the responses $\vec{y} = \{y^{(1)}, y^{(2)}, \dots, y^{(n)}\}^T$.

A polynomial approximation of f of order m can be written as:

$$\hat{f}(x, m, \vec{w}) = \sum_{i=0}^m w_i x^i.$$

In the spirit of the earlier discussion of maximum likelihood parameter estimation, we seek to estimate $w = w_0, w_1, \dots, w_m^T$ through a least squares solution of:

$$\Phi \vec{w} = \vec{y}$$

where Φ is the Vandermonde matrix:

$$\Phi = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^m \\ 1 & x_2 & x_2^2 & \dots & x_2^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^m \end{bmatrix}.$$

The maximum likelihood estimate of w is given by:

$$\vec{w} = (\Phi^T \Phi)^{-1} \Phi^T y,$$

where $\Phi^+ = (\Phi^T \Phi)^{-1} \Phi^T$ is the Moore-Penrose pseudo-inverse of Φ (see Section 72.3).

The polynomial approximation of order m is essentially a truncated Taylor series expansion. While higher values of m yield more accurate approximations, they risk overfitting the noise in the data.

To prevent this, we estimate m using cross-validation. This involves minimizing the cross-validation error over a discrete set of possible orders m (e.g., $m \in 1, 2, \dots, 15$).

For each m , the data is split into q subsets. The model is trained on $q - 1$ subsets, and the error is computed on the left-out subset. This process is repeated for all subsets, and the cross-validation error is summed. The order m with the smallest cross-validation error is chosen.

69.2. Polynomial Fitting in Python

69.2.1. Fitting the Polynomial

```
from sklearn.model_selection import KFold
def polynomial_fit(X, Y, max_order=15, q=5):
    """
    Fits a one-variable polynomial to one-dimensional data using cross-validation.

    Args:
        X (array-like): Training data vector (independent variable).
        Y (array-like): Training data vector (dependent variable).
        max_order (int): Maximum polynomial order to consider. Default is 15.
        q (int): Number of cross-validation folds. Default is 5.

    Returns:
        best_order (int): The optimal polynomial order.
        coeff (array): Coefficients of the best-fit polynomial.
        mnstd (tuple): Normalization parameters (mean, std) for X.
    """
    X = np.array(X)
    Y = np.array(Y)
    n = len(X)
    # Normalize X
    mnstd = (np.mean(X), np.std(X))
    X_norm = (X - mnstd[0]) / mnstd[1]
```

```

# Cross-validation setup
kf = KFold(n_splits=q, shuffle=True, random_state=42)
cross_val_errors = np.zeros(max_order)
for order in range(1, max_order + 1):
    fold_errors = []
    for train_idx, val_idx in kf.split(X_norm):
        X_train, X_val = X_norm[train_idx], X_norm[val_idx]
        Y_train, Y_val = Y[train_idx], Y[val_idx]
        # Fit polynomial
        coeff = np.polyfit(X_train, Y_train, order)
        # Predict on validation set
        Y_pred = np.polyval(coeff, X_val)
        # Compute mean squared error
        mse = np.mean((Y_val - Y_pred) ** 2)
        fold_errors.append(mse)
    cross_val_errors[order - 1] = np.mean(fold_errors)
# Find the best order
best_order = np.argmin(cross_val_errors) + 1
# Fit the best polynomial on the entire dataset
best_coeff = np.polyfit(X_norm, Y, best_order)
return best_order, best_coeff, mnstd

```

69.2.2. Explaining the k -fold Cross-Validation

The line

```

kf = KFold(n_splits=q, shuffle=True, random_state=42)

```

initializes a k -Fold cross-validator object from the `sklearn.model_selection` library. The `n_splits` parameter specifies the number of folds. The data will be divided into q parts. In each iteration of the cross-validation, one part will be used as the validation set, and the remaining $q-1$ parts will be used as the training set.

The `kf.split` method takes the dataset `X_norm` as input and yields pairs of index arrays for each fold: * `train_idx`: In each iteration, `train_idx` is an array containing the indices of the data points that belong to the training set for that specific fold. * `val_idx`: Similarly, `val_idx` is an array containing the indices of the data points that belong to the validation (or test) set for that specific fold.

The loop will run q times (the number of splits). In each iteration, a different fold serves as the validation set, while the other $q-1$ folds form the training set.

Here's a Python example to demonstrate the values of `train_idx` and `val_idx`:

```

# Sample data (e.g., X_norm)
X_norm = np.array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
print(f"Original data indices: {np.arange(len(X_norm))}\n")
# Number of splits (folds)
q = 3 # Let's use 3 folds for this example
# Initialize KFold
kf = KFold(n_splits=q, shuffle=True, random_state=42)
# Iterate through the splits and print the indices
fold_number = 1
for train_idx, val_idx in kf.split(X_norm):
    print(f"--- Fold {fold_number} ---")
    print(f"Train indices: {train_idx}")
    print(f"Validation indices: {val_idx}")
    print(f"Training data for this fold: {X_norm[train_idx]}")
    print(f"Validation data for this fold: {X_norm[val_idx]}\n")
    fold_number += 1

```

Original data indices: [0 1 2 3 4 5 6 7 8 9]

--- Fold 1 ---

Train indices: [2 3 4 6 7 9]

Validation indices: [0 1 5 8]

Training data for this fold: [0.3 0.4 0.5 0.7 0.8 1.]

Validation data for this fold: [0.1 0.2 0.6 0.9]

--- Fold 2 ---

Train indices: [0 1 3 4 5 6 8]

Validation indices: [2 7 9]

Training data for this fold: [0.1 0.2 0.4 0.5 0.6 0.7 0.9]

Validation data for this fold: [0.3 0.8 1.]

--- Fold 3 ---

Train indices: [0 1 2 5 7 8 9]

Validation indices: [3 4 6]

Training data for this fold: [0.1 0.2 0.3 0.6 0.8 0.9 1.]

Validation data for this fold: [0.4 0.5 0.7]

69.2.3. Making Predictions

To make predictions, we can use the coefficients. The data is standardized around its mean in the polynomial function, which is why the vector `mnstd` is required. The coefficient vector is computed based on the normalized data, and this must be taken into account if further analytical calculations are performed on the fitted model.

The polynomial approximation of C_D is:

$$C_D(x) = w_8x^8 + w_7x^7 + \dots + w_1x + w_0,$$

where x is normalized as:

$$\bar{x} = \frac{x - \mu(X)}{\sigma(X)}$$

```
def predict_polynomial_fit(X, coeff, mnstd):
    """
    Generates predictions for the polynomial fit.

    Args:
        X (array-like): Original independent variable data.
        coeff (array): Coefficients of the best-fit polynomial.
        mnstd (tuple): Normalization parameters (mean, std) for X.

    Returns:
        tuple: De-normalized predicted X values and corresponding Y predictions.
    """
    # Normalize X
    X_norm = (X - mnstd[0]) / mnstd[1]

    # Generate predictions
    X_pred = np.linspace(min(X_norm), max(X_norm), 100)
    Y_pred = np.polyval(coeff, X_pred)

    # De-normalize X for plotting
    X_pred_original = X_pred * mnstd[1] + mnstd[0]

    return X_pred_original, Y_pred
```

69.2.4. Plotting the Results

```
def plot_polynomial_fit(X, Y, X_pred_original, Y_pred, best_order, y_true=None):
    """
    Visualizes the polynomial fit.

    Args:
        X (array-like): Original independent variable data.
```

```

        Y (array-like): Original dependent variable data.
        X_pred_original (array): De-normalized predicted X values.
        Y_pred (array): Predicted Y values.
        y_true (array): True Y values.
        best_order (int): The optimal polynomial order.
    """
    plt.scatter(X, Y, label="Training Data", color="grey", marker="o")
    plt.plot(X_pred_original, Y_pred, label=f"Order {best_order} Polynomial", color="blue")
    if y_true is not None:
        plt.plot(X, y_true, label="True Function", color="blue", linestyle="--")
    plt.title(f"Polynomial Fit (Order {best_order})")
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.legend()
    plt.show()

```

69.3. Example One: Aerofoil Drag

The circles in Figure 69.1 represent 101 drag coefficient values obtained through a numerical simulation by iterating each member of a family of aerofoils towards a target lift value (see the Appendix, Section A.3 in Forrester, Sóbester, and Keane (2008)). The members of the family have different shapes, as determined by the sampling plan:

$$X = x_1, x_2, \dots, x_{101}$$

The responses are:

$$C_D = \{C_D^{(1)}, C_D^{(2)}, \dots, C_D^{(101)}\}$$

These responses are corrupted by “noise,” which are deviations of the systematic variety caused by small changes in the computational mesh from one design to the next.

The original data is measured in natural units, i.e., from -0.3 unit to 0.1 unit. The data is normalized to the range of 0 to 1 for the computation with the `aerofoilcd` function. The data is then fitted with a polynomial of order m . To obtain the best polynomial through this data, the following Python code can be used:

```

from spotpython.surrogate.functions.forr08a import aerofoilcd
import numpy as np
import matplotlib.pyplot as plt
X = np.linspace(-0.3, 0.1, 101)
# normalize the data so that it will be in the range of 0 to 1

```


69.3. Example One: Aerofoil Drag

```
a = np.min(X)
b = np.max(X)
X_cod = (X - a) / (b - a)
y = aerofoilcd(X_cod)
best_order, best_coeff, mnstd = polynomial_fit(X, y)
X_pred_original, Y_pred = predict_polynomial_fit(X, best_coeff, mnstd)

plot_polynomial_fit(X, y, X_pred_original, Y_pred, best_order)
```

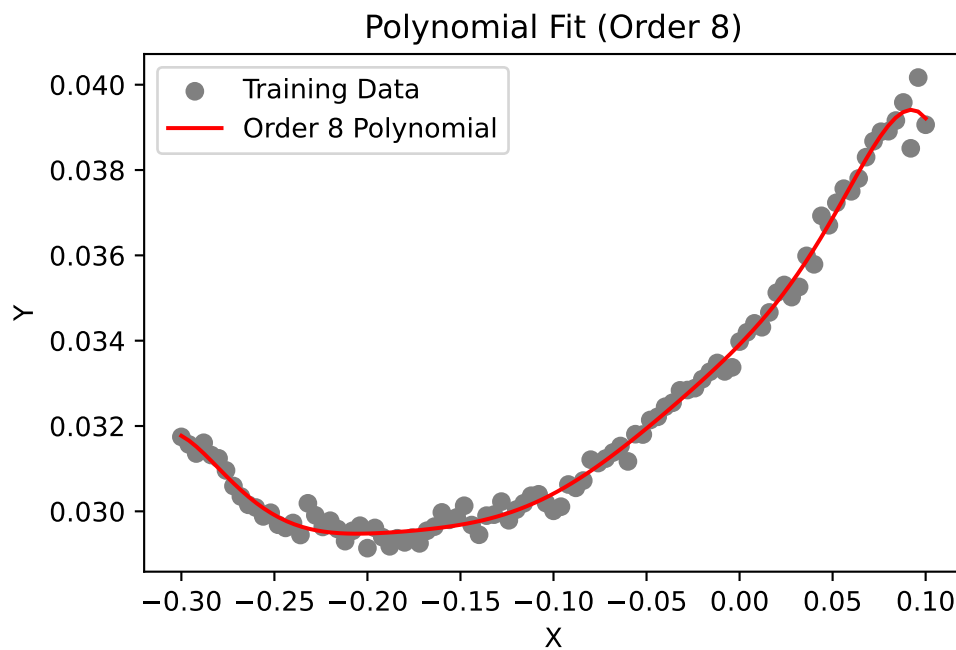


Figure 69.1.: Aerofoil drag data

Figure 69.1 shows an eighth-order polynomial fitted through the aerofoil drag data. The order was selected via cross-validation, and the coefficients were determined through likelihood maximization. Results, i.e, the best polynomial order and coefficients, are printed in the console. The coefficients are stored in the vector `best_coeff`, which contains the coefficients of the polynomial in descending order. The first element is the coefficient of x^8 , and the last element is the constant term. The vector `mnstd`, containing the mean and standard deviation of X , is:

69. Polynomial Models

```
print(f"Best polynomial order: {best_order}\n")
print(f"Coefficients (starting with w0):\n {best_coeff}\n")
print(f"Normalization parameters (mean, std):\n {mnstd}\n")
```

Best polynomial order: 8

Coefficients (starting with w0):

```
[-0.00022964 -0.00014636  0.00116742  0.00052988 -0.0016912  -0.00047398
  0.00244373  0.00270342  0.03041508]
```

Normalization parameters (mean, std):

```
(np.float64(-0.09999999999999999), np.float64(0.11661903789690602))
```

69.4. Example Two: A Multimodal Test Case

Let us consider the one-variable test function:

$$f(x) = (6x - 2)^2 \sin(12x - 4).$$

```
import numpy as np
from spotpython.surrogate.functions.forr08a import onevar
X = np.linspace(0, 1, 51)
y_true = onevar(X)
# initialize random seed
np.random.seed(42)
y = y_true + np.random.normal(0, 1, len(X))*1.1
best_order, best_coeff, mnstd = polynomial_fit(X, y)
X_pred_original, Y_pred = predict_polynomial_fit(X, best_coeff, mnstd)
```

```
plot_polynomial_fit(X, y, X_pred_original, Y_pred, best_order, y_true=y_true)
```

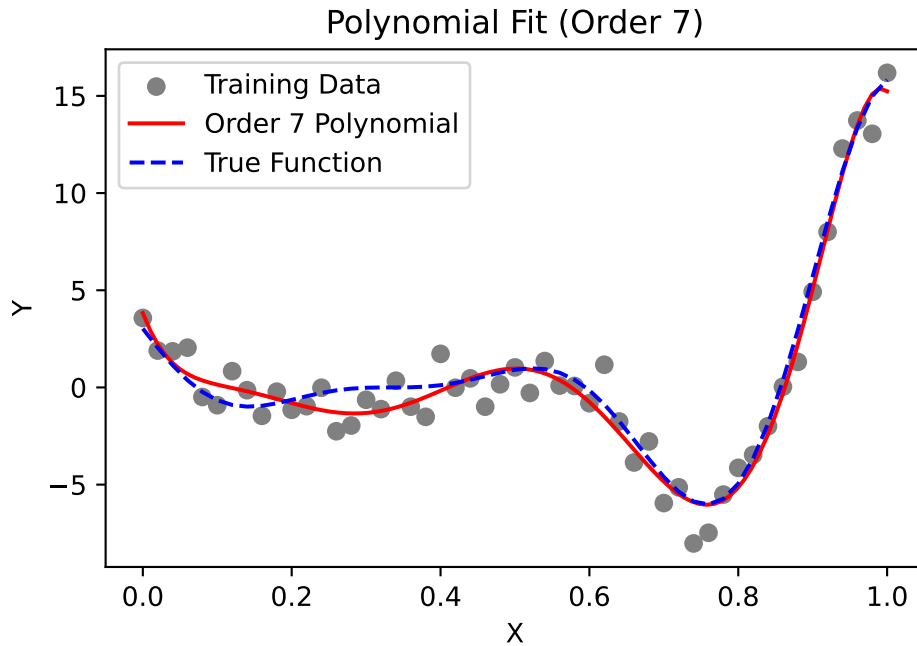


Figure 69.2.: Onevar function

This function, depicted by the dotted line in Figure 69.2, has local minima of different depths, which can be deceptive to some surrogate-based optimization procedures. Here, we use it as an example of a multimodal function for polynomial fitting.

We generate the training data (depicted by circles in Figure 69.2) by adding normally distributed noise to the function. Figure 69.2 shows a seventh-order polynomial fitted through the noisy data. This polynomial was selected as it minimizes the cross-validation metric.

69.5. Extending to Multivariable Polynomial Models

While the examples above focus on the one-variable case, real-world engineering problems typically involve multiple input variables. For k -variable problems, polynomial approximation becomes significantly more complex but follows the same fundamental principles. For a k -dimensional input space, the polynomial approximation can be expressed as:

69. Polynomial Models

$$\hat{f}(\vec{x}) = \sum_{i=1}^N w_i \phi_i(\vec{x}),$$

where $\phi_i(\vec{x})$ represents multivariate basis functions, and N is the total number of terms in the polynomial. Unlike the univariate case, these basis functions include all possible combinations of variables up to the selected polynomial order m , which might result in a “basis function explosion” as the number of variables increases.

For a third-order polynomial ($m = 3$) with three variables ($k = 3$), the complete set of basis functions would include 20 terms:

$$\text{Constant term: } 1 \quad (69.1)$$

$$\text{First-order terms: } x_1, x_2, x_3 \quad (69.2)$$

$$\text{Second-order terms: } x_1^2, x_2^2, x_3^2, x_1x_2, x_1x_3, x_2x_3 \quad (69.3)$$

$$\text{Third-order terms: } x_1^3, x_2^3, x_3^3, x_1^2x_2, x_1^2x_3, x_2^2x_1, x_2^2x_3, x_3^2x_1, x_3^2x_2, x_1x_2x_3 \quad (69.4)$$

The total number of terms grows combinatorially as $N = \binom{k+m}{m}$, which quickly becomes prohibitive as dimensionality increases. For example, a 10-variable cubic polynomial requires $\binom{13}{3} = 286$ coefficients! This exponential growth creates three interrelated challenges:

- **Model Selection:** Determining the appropriate polynomial order m that balances complexity with generalization ability
- **Coefficient Estimation:** Computing the potentially large number of weights $\vec{w}$ while avoiding numerical instability
- **Term Selection:** Identifying which specific basis functions should be included, as many may be irrelevant to the response

Several techniques have been developed to address these challenges:

- Regularization methods (LASSO, ridge regression) that penalize model complexity
- Stepwise regression algorithms that incrementally add or remove terms
- Dimension reduction techniques that project the input space to lower dimensions
- Orthogonal polynomials that improve numerical stability for higher-order models

These limitations of polynomial models in higher dimensions motivate the exploration of more flexible surrogate modeling approaches like Radial Basis Functions and Kriging, which we’ll examine in subsequent sections.

69.6. Jupyter Notebook

i Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

70. Radial Basis Function Models

Note

- This section is based on chapter 2.3 in Forrester, Sóbester, and Keane (2008).
- The following Python packages are imported:

```
import numpy as np
import matplotlib.pyplot as plt
```

70.1. Radial Basis Function Models

Scientists and engineers frequently tackle complex functions by decomposing them into a “vocabulary” of simpler, well-understood basic functions. These fundamental building blocks possess properties that make them easier to analyze mathematically and implement computationally. We explored this concept earlier with multivariable polynomials, where complex behaviors were modeled using combinations of polynomial terms such as 1, x_1 , x_2 , x_1^2 , and x_1x_2 . This approach is not limited to polynomials; it extends to various function classes, including trigonometric functions, exponential functions, and even more complex structures.

While Fourier analysis—perhaps the most widely recognized example of this approach—excels at representing periodic phenomena through sine and cosine functions, the focus in Forrester, Sóbester, and Keane (2008) is broader. They aim to approximate arbitrary smooth, continuous functions using strategically positioned basis functions. Specifically, radial basis function (RBF) models employ symmetrical basis functions centered at selected points distributed throughout the design space. These basis functions have the unique property that their output depends only on the distance from their center point.

First, we give a definition of the Euclidean distance, which is the most common distance measure used in RBF models.

Definition 70.1 (Euclidean Distance). The Euclidean distance between two points in a k -dimensional space is defined as:

70. Radial Basis Function Models

$$\|\vec{x} - \vec{c}\| = \sqrt{\sum_{i=1}^k (x_i - c_i)^2}, \quad (70.1)$$

where:

- $\vec{x} = (x_1, x_2, \dots, x_d)$ is the first point,
- $\vec{c} = (c_1, c_2, \dots, c_d)$ is the second point, and
- k is the number of dimensions.

The Euclidean distance measure represents the straight-line distance between two points in Euclidean space.

Using the Euclidean distance, we can define the radial basis function (RBF) model.

Definition 70.2 (Radial Basis Function (RBF)). Mathematically, a radial basis function ψ can be expressed as:

$$\psi(\vec{x}) = \psi(\|\vec{x} - \vec{c}\|), \quad (70.2)$$

where $\vec{x}$ is the input vector, $\vec{c}$ is the center of the function, and $\|\vec{x} - \vec{c}\|$ denotes the Euclidean distance between $\vec{x}$ and $\vec{c}$.

In the context of RBFs, the Euclidean distance calculation determines how much influence a particular center point $\vec{c}$ has on the prediction at point $\vec{x}$.

We will first examine interpolating RBF models, which assume noise-free data and pass exactly through all training points. This approach provides an elegant mathematical foundation before we consider more practical scenarios where data contains measurement or process noise.

70.1.1. Fitting Noise-Free Data

Let us consider the scalar valued function f observed without error, according to the sampling plan $X = \{\vec{x}^{(1)}, \vec{x}^{(2)}, \dots, \vec{x}^{(n)}\}^T$, yielding the responses $\vec{y} = \{y^{(1)}, y^{(2)}, \dots, y^{(n)}\}^T$.

For a given set of n_c centers $\vec{c}^{(i)}$, we would like to express the RBF model as a linear combination of the basis functions centered at these points. The goal is to find the weights $\vec{w}$ that minimize the error between the predicted and observed values. Thus, we seek a radial basis function approximation to f of the fixed form:

$$\hat{f}(\vec{x}) = \sum_{i=1}^{n_c} w_i \psi(\|\vec{x} - \vec{c}^{(i)}\|), \quad (70.3)$$

where

- w_i are the weights of the n_c basis functions,
- $\vec{c}^{(i)}$ are the n_c centres of the basis functions, and
- ψ is a radial basis function.

The notation $\|\cdot\|$ denotes the Euclidean distance between two points in the design space as defined in Equation 70.1.

70.1.1.1. Selecting Basis Functions: From Fixed to Parametric Forms

When implementing a radial basis function model, we initially have one undetermined parameter per basis function: the weight applied to each function's output. This simple parameterization remains true when we select from several standard fixed-form basis functions, such as:

- Linear ($\psi(r) = r$): The simplest form, providing a response proportional to distance
- Cubic ($\psi(r) = r^3$): Offers stronger emphasis on points farther from the center
- Thin plate spline ($\psi(r) = r^2 \ln r$): Models the physical bending of a thin sheet, providing excellent smoothness properties

While these fixed basis functions are computationally efficient, they offer limited flexibility in how they generalize across the design space. For more adaptive modeling power, we can employ parametric basis functions that introduce additional tunable parameters:

- Gaussian ($\psi(r) = e^{-r^2/(2\sigma^2)}$): Produces bell-shaped curves with σ controlling the width of influence
- Multiquadric ($\psi(r) = (r^2 + \sigma^2)^{1/2}$): Provides broader coverage with less localized effects
- Inverse multiquadric ($\psi(r) = (r^2 + \sigma^2)^{-1/2}$): Offers sharp peaks near centers with asymptotic behavior

The parameter σ in these functions serves as a shape parameter that controls how rapidly the function's influence decays with distance. This added flexibility enables significantly better generalization, particularly when modeling complex responses, though at the cost of a more involved parameter estimation process requiring optimization of both weights and shape parameters.

Example 70.1 (Gaussian RBF). Using the general definition of a radial basis function (Equation 70.2), we can express the Gaussian RBF as:

$$\psi(\vec{x}) = \exp\left(-\frac{\|\vec{x} - \vec{c}\|^2}{2\sigma^2}\right) = \exp\left(-\frac{\sum_{j=1}^k (x_j - c_j)^2}{2\sigma^2}\right) \quad (70.4)$$

where:

70. Radial Basis Function Models

- $\vec{x}$ is the input vector,
- $\vec{c}$ is the center vector,
- $\|\vec{x} - \vec{c}\|$ is the Euclidean distance between the input and center, and
- σ is the width parameter that controls how quickly the function's response diminishes with distance from the center.

The Gaussian RBF produces a bell-shaped response that reaches its maximum value of 1 when $\vec{x} = \vec{c}$ and asymptotically approaches zero as the distance increases. The parameter σ determines how “localized” the response is—smaller values create a narrower peak with faster decay, while larger values produce a broader, more gradual response across the input space. Figure 70.1 shows the Gaussian RBF for different values of σ in an one-dimensional space. The center of the RBF is set at 0, and the width parameter σ varies to illustrate how it affects the shape of the function.

```
def gaussian_rbf(x, center, sigma):  
    """  
    Compute the Gaussian Radial Basis Function.  
  
    Args:  
        x (ndarray): Input points  
        center (float): Center of the RBF  
        sigma (float): Width parameter  
  
    Returns:  
        ndarray: RBF values  
    """  
    return np.exp(-((x - center)**2) / (2 * sigma**2))
```

The sum of Gaussian RBFs can be visualized by summing the individual Gaussian RBFs centered at different points as shown in Figure 70.2. The following code snippet demonstrates how to create this plot showing the sum of three Gaussian RBFs with different centers and a common width parameter σ .

70.1.1.2. The Interpolation Condition: Elegant Solutions Through Linear Systems

A remarkable property of radial basis function models is that regardless of which basis functions we choose—parametric or fixed—determining the weights $\vec{w}$ remains straightforward through interpolation. The core principle is elegantly simple: we require our model to exactly reproduce the observed data points:

$$\hat{f}(\vec{x}^{(i)}) = y^{(i)}, \quad i = 1, 2, \dots, n. \quad (70.5)$$

This constraint produces one of the most powerful aspects of RBF modeling: while the system in Equation 70.5 is linear with respect to the weights $\vec{w}$, the resulting predictor

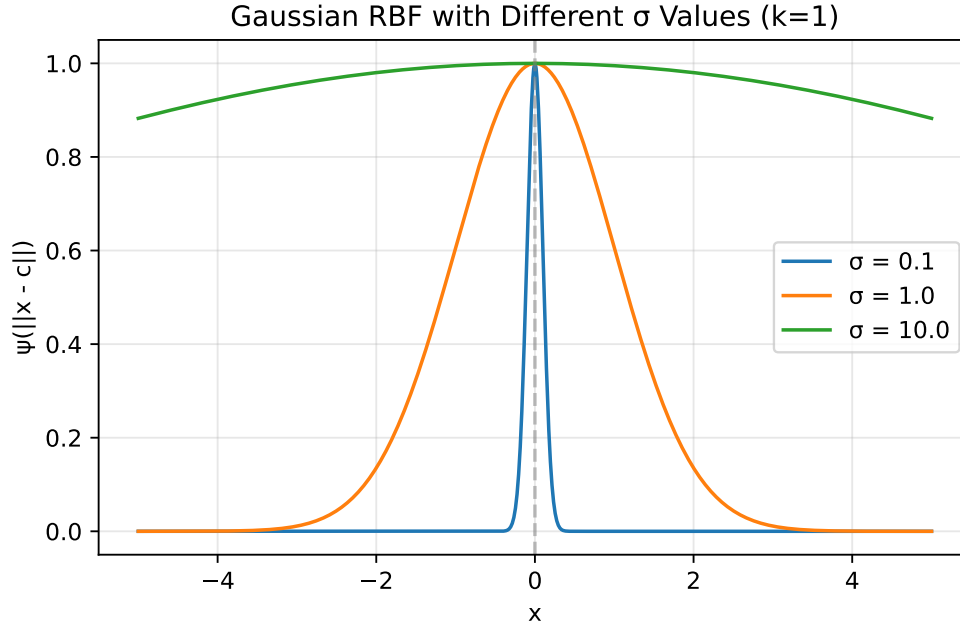


Figure 70.1.: Gaussian RBF

$\hat{f}$ can capture highly nonlinear relationships in the data. The RBF approach transforms a complex nonlinear modeling problem into a solvable linear algebra problem.

For a unique solution to exist, we require that the number of basis functions equals the number of data points ($n_c = n$). The standard practice, which greatly simplifies implementation, is to center each basis function at a training data point, setting $\vec{c}^{(i)} = \vec{x}^{(i)}$ for all $i = 1, 2, \dots, n$. This choice allows us to express the interpolation condition as a compact matrix equation:

$$\Psi \vec{w} = \vec{y}.$$

Here, Ψ represents the Gram matrix (also called the design matrix or kernel matrix), whose elements measure the similarity between data points:

$$\Psi_{i,j} = \psi(\|\vec{x}^{(i)} - \vec{x}^{(j)}\|), \quad i, j = 1, 2, \dots, n.$$

The solution for the weight vector becomes:

$$\vec{w} = \Psi^{-1} \vec{y}.$$

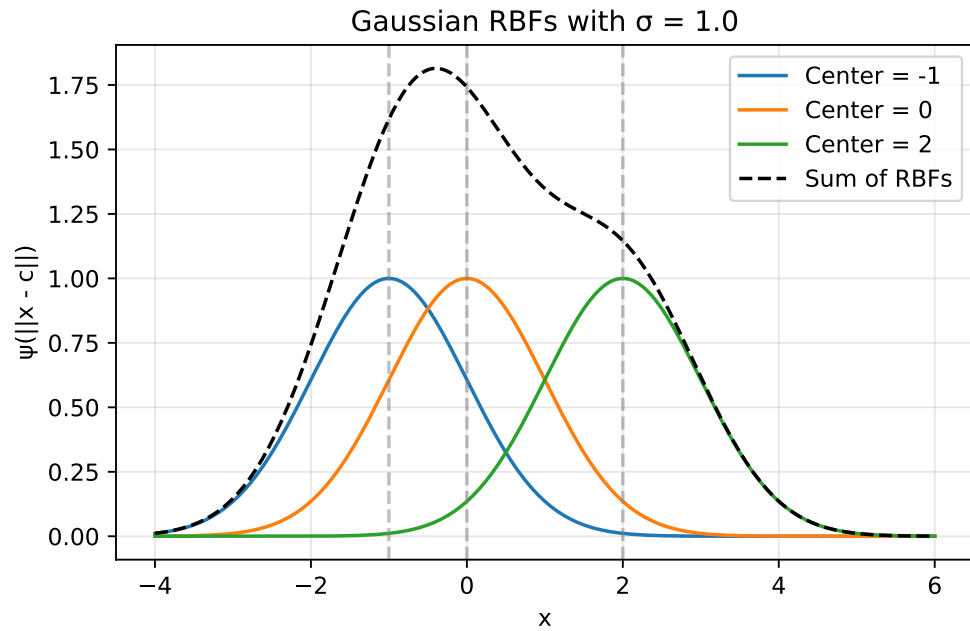


Figure 70.2.: Sum of Gaussian RBFs

This matrix inversion step is the computational core of the RBF model fitting process, and the numerical properties of this operation depend critically on the chosen basis function. Different basis functions produce Gram matrices with distinct conditioning properties, directly affecting both computational stability and the model's generalization capabilities.

70.1.2. Numerical Stability Through Positive Definite Matrices

A significant advantage of Gaussian and inverse multiquadric basis functions lies in their mathematical guarantees. Vapnik (1998) demonstrated that these functions always produce symmetric positive definite Gram matrices when using strictly positive definite kernels (see Section 72.4), which is a critical property for numerical reliability. Unlike other basis functions that may lead to ill-conditioned systems, these functions ensure the existence of unique, stable solutions.

This positive definiteness enables the use of Cholesky factorization, which offers substantial computational advantages over standard matrix inversion techniques. The Cholesky approach reduces the computational cost (reducing from $O(n^3)$ to roughly $O(n^3/3)$) while significantly improving numerical stability when handling the inevitable rounding errors in floating-point arithmetic. This robustness to numerical issues explains why

Gaussian and inverse multiquadric basis functions remain the preferred choice in many practical RBF implementations.

Furthermore, the positive definiteness guarantee provides theoretical assurances about the model's interpolation properties—ensuring that the RBF interpolant exists and is unique for any distinct set of centers. This mathematical foundation gives practitioners confidence in the method's reliability, particularly for complex engineering applications where model stability is paramount.

The computational advantage stems from how a symmetric positive definite matrix Ψ can be efficiently decomposed into the product of an upper triangular matrix U and its transpose:

$$\Psi = U^T U.$$

This decomposition transforms the system

$$\Psi \vec{w} = \vec{y}$$

into

$$U^T U \vec{w} = \vec{y},$$

which can be solved through two simpler triangular systems:

- First solve $U^T \vec{v} = \vec{y}$ for the intermediate vector $\vec{v}$
- Then solve $U \vec{w} = \vec{v}$ for the desired weights $\vec{w}$

In Python implementations, this process is elegantly handled using NumPy's or SciPy's Cholesky decomposition functions, followed by specialized solvers that exploit the triangular structure:

```
from scipy.linalg import cholesky, cho_solve
# Compute the Cholesky factorization
L = cholesky(Psi, lower=True) # L is the lower triangular factor
weights = cho_solve((L, True), y) # Efficient solver for (L L^T)w = y
```

70.1.3. III-Conditioning

An important numerical consideration in RBF modeling is that points positioned extremely close to each other in the input space X can lead to severe ill-conditioning of the Gram matrix (Micchelli 1986). This ill-conditioning manifests as nearly linearly dependent rows and columns in Ψ , potentially causing the Cholesky factorization to fail.

While this problem rarely arises with initial space-filling experimental designs (such as Latin Hypercube or quasi-random sequences), it frequently emerges during sequential

optimization processes that adaptively add infill points in promising regions. As these clusters of points concentrate in areas of high interest, the condition number of the Gram matrix deteriorates, jeopardizing numerical stability.

Several mitigation strategies exist: regularization through ridge-like penalties (modifying the standard RBF interpolation problem by adding a penalty term to the diagonal of the Gram matrix. This creates a literal “ridge” along the diagonal of the matrix), removing nearly coincident points, clustering, or applying more sophisticated approaches. One theoretically elegant solution involves augmenting non-conditionally positive definite basis functions with polynomial terms (Keane and Nair 2005). This technique not only improves conditioning but also ensures polynomial reproduction properties, enhancing the approximation quality for certain function classes while maintaining numerical stability.

Beyond determining $\vec{w}$, there is, of course, the additional task of estimating any other parameters introduced via the basis functions. A typical example is the σ of the Gaussian basis function, usually taken to be the same for all basis functions, though a different one can be selected for each centre, as is customary in the case of the Kriging basis function, to be discussed shortly (once again, we trade additional parameter estimation complexity versus increased flexibility and, hopefully, better generalization).

70.1.4. Parameter Optimization: A Two-Level Approach

When building RBF models, we face two distinct parameter estimation challenges:

- Determining the weights ($\vec{w}$): These parameters ensure our model precisely reproduces the training data. For any fixed basis function configuration, we can calculate these weights directly through linear algebra as shown earlier.
- Optimizing shape parameters (like σ in Gaussian RBF): These parameters control how the model generalizes to new, unseen data. Unlike weights, there’s no direct formula to find their optimal values.

To address this dual challenge, we employ a nested optimization strategy (inner and outer levels):

70.1.4.1. Inner Level ($\vec{w}$)

For each candidate value of shape parameters (e.g., σ), we determine the corresponding optimal weights $\vec{w}$ by solving the linear system. The `estim_weights()` method implements the inner level optimization by calculating the optimal weights $\vec{w}$ for a given shape parameter (σ):

```

def estim_weights(self):
    # [...]

    # Construct the Phi (Psi) matrix
    self.Phi = np.zeros((n, n))
    for i in range(n):
        for j in range(i+1):
            self.Phi[i, j] = self.basis(d[i, j], self.sigma)
            self.Phi[j, i] = self.Phi[i, j]

    # Calculate weights using appropriate method
    if self.code == 4 or self.code == 6:
        # Use Cholesky factorization for Gaussian or inverse multiquadric
        try:
            L = cholesky(self.Phi, lower=True)
            self.weights = cho_solve((L, True), self.y)
            self.success = True
        except np.linalg.LinAlgError:
            # Error handling...
    else:
        # Use direct solve for other basis functions
        try:
            self.weights = np.linalg.solve(self.Phi, self.y)
            self.success = True
        except np.linalg.LinAlgError:
            # Error handling...

    return self

```

This method:

- Creates the Gram matrix (Phi) based on distances between points
- Solves the linear system $\Psi \vec{w} = \vec{y}$ for weights
- Uses appropriate numerical methods based on the basis function type (Cholesky factorization or direct solve)

70.1.4.2. Outer Level (σ)

We use cross-validation to evaluate how well the model generalizes with different shape parameter values. The outer level optimization is implemented within the `fit()` method, where cross-validation is used to evaluate different σ values:

```

def fit(self):
    if self.code < 4:
        # Fixed basis function, only w needs estimating
        self.estim_weights()
    else:
        # Basis function requires a sigma, estimate first using cross-validation
        # [...]

        # Generate candidate sigma values
        sigmas = np.logspace(-2, 2, 30)

        # Setup cross-validation (determine number of folds)
        # [...]

        cross_val = np.zeros(len(sigmas))

        # For each candidate sigma value
        for sig_index, sigma in enumerate(sigmas):
            print(f"Computing cross-validation metric for Sigma={sigma:.4f}...")

            # Perform k-fold cross-validation
            for j in range(len(from_idx)):
                # Create and fit model on training subset
                temp_model = Rbf(
                    X=X_orig[xs_temp],
                    y=y_orig[xs_temp],
                    code=self.code
                )
                temp_model.sigma = sigma

                # Call inner level optimization
                temp_model.estim_weights()

                # Evaluate on held-out data
                # [...]

            # Select best sigma based on cross-validation performance
            min_cv_index = np.argmin(cross_val)
            best_sig = sigmas[min_cv_index]

        # Use the best sigma for final model
        self.sigma = best_sig
        self.estim_weights() # Call inner level again with optimal sigma

```


The outer level:

- Generates a range of candidate σ values
- For each σ , performs k-fold cross-validation:
 - Creates models on subsets of the data
 - Calls the inner level method (`estim_weights()`) to determine weights
 - Evaluates prediction quality on held-out data
- Selects the σ that minimizes cross-validation error
- Performs a final call to the inner level method with the optimal σ

This two-level approach is particularly critical for parametric basis functions (Gaussian, multiquadric, etc.), where the wrong choice of shape parameter could lead to either overfitting (too much flexibility) or underfitting (too rigid). Cross-validation provides an unbiased estimate of how well different parameter choices will perform on new data, helping us balance the trade-off between fitting the training data perfectly and generalizing well.

70.2. Python Implementation of the RBF Model

Section 70.2 shows a Python implementation of this parameter estimation process (based on a cross-validation routine), which will represent the surrogate, once its parameters have been estimated. The model building process is very simple.

Instead of using a dictionary for bookkeeping, we implement a Python class `Rbf` that encapsulates all the necessary data and functionality. The class stores the sampling plan X as the `X` attribute and the corresponding n -vector of responses y as the `y` attribute. The `code` attribute specifies the type of basis function to be used. After fitting the model, the class will also contain the estimated parameter values $\vec{w}$ and, if a parametric basis function is used, σ . These are stored in the `weights` and `sigma` attributes respectively.

Finally, a note on prediction error estimation. We have already indicated that the guarantee of a positive definite Ψ is one of the advantages of Gaussian radial basis functions. They also possess another desirable feature: it is relatively easy to estimate their prediction error at any $\vec{x}$ in the design space. Additionally, the expectation function of the improvement in minimum (or maximum) function value with respect to the minimum (or maximum) known so far can also be calculated quite easily, both of these features being very useful when the optimization of f is the goal of the surrogate modelling process.

70.2.1. The Rbf Class

The Rbf class implements the Radial Basis Function model. It encapsulates all the data and methods needed for fitting the model and making predictions.

```
import numpy as np
from scipy.linalg import cholesky, cho_solve
import numpy.random as rnd

class Rbf:
    """Radial Basis Function model implementation.

    Attributes:
        X (ndarray): The sampling plan (input points).
        y (ndarray): The response vector.
        code (int): Type of basis function to use.
        weights (ndarray, optional): The weights vector (set after fitting).
        sigma (float, optional): Parameter for parametric basis functions.
        Phi (ndarray, optional): The Gram matrix (set during fitting).
        success (bool, optional): Flag indicating successful fitting.
    """

    def __init__(self, X=None, y=None, code=3):
        """Initialize the RBF model.

        Args:
            X (ndarray, optional):
                The sampling plan.
            y (ndarray, optional):
                The response vector.
            code (int, optional):
                Type of basis function.
                Default is 3 (thin plate spline).
        """
        self.X = X
        self.y = y
        self.code = code
        self.weights = None
        self.sigma = None
        self.Phi = None
        self.success = None

    def basis(self, r, sigma=None):
        """Compute the value of the basis function.
```

```

Args:
    r (float): Radius (distance)
    sigma (float, optional): Parameter for parametric basis functions

Returns:
    float: Value of the basis function
"""
# Use instance sigma if not provided
if sigma is None and hasattr(self, 'sigma'):
    sigma = self.sigma

if self.code == 1:
    # Linear function
    return r
elif self.code == 2:
    # Cubic
    return r**3
elif self.code == 3:
    # Thin plate spline
    if r < 1e-200:
        return 0
    else:
        return r**2 * np.log(r)
elif self.code == 4:
    # Gaussian
    return np.exp(-(r**2)/(2*sigma**2))
elif self.code == 5:
    # Multi-quadric
    return (r**2 + sigma**2)**0.5
elif self.code == 6:
    # Inverse Multi-Quadric
    return (r**2 + sigma**2)**(-0.5)
else:
    raise ValueError("Invalid basis function code")

def estim_weights(self):
    """Estimates the basis function weights if sigma is known or not required.

Returns:
    self: The updated model instance
"""
    # Check if sigma is required but not provided
    if self.code > 3 and self.sigma is None:
        raise ValueError("The basis function requires a sigma parameter")

```

```

# Number of points
n = len(self.y)

# Build distance matrix
d = np.zeros((n, n))
for i in range(n):
    for j in range(i+1):
        d[i, j] = np.linalg.norm(self.X[i] - self.X[j])
        d[j, i] = d[i, j]

# Construct the Phi (Psi) matrix
self.Phi = np.zeros((n, n))
for i in range(n):
    for j in range(i+1):
        self.Phi[i, j] = self.basis(d[i, j], self.sigma)
        self.Phi[j, i] = self.Phi[i, j]

# Calculate weights using appropriate method
if self.code == 4 or self.code == 6:
    # Use Cholesky factorization for Gaussian or inverse multiquadric
    try:
        L = cholesky(self.Phi, lower=True)
        self.weights = cho_solve((L, True), self.y)
        self.success = True
    except np.linalg.LinAlgError:
        print("Cholesky factorization failed.")
        print("Two points may be too close together.")
        self.weights = None
        self.success = False
else:
    # Use direct solve for other basis functions
    try:
        self.weights = np.linalg.solve(self.Phi, self.y)
        self.success = True
    except np.linalg.LinAlgError:
        self.weights = None
        self.success = False

return self

def fit(self):
    """Estimates the parameters of the Radial Basis Function model.

    Returns:

```

```

        self: The updated model instance
    """
    if self.code < 4:
        # Fixed basis function, only w needs estimating
        self.estim_weights()
    else:
        # Basis function also requires a sigma, estimate first
        # Save original model data
        X_orig = self.X.copy()
        y_orig = self.y.copy()

        # Direct search between 10^-2 and 10^2
        sigmas = np.logspace(-2, 2, 30)

        # Number of cross-validation subsets
        if len(self.X) < 6:
            q = 2
        elif len(self.X) < 15:
            q = 3
        elif len(self.X) < 50:
            q = 5
        else:
            q = 10

        # Number of sample points
        n = len(self.X)

        # X split into q randomly selected subsets
        xs = rnd.permutation(n)
        full_xs = xs.copy()

        # The beginnings of the subsets...
        from_idx = np.arange(0, n, n//q)
        if from_idx[-1] >= n:
            from_idx = from_idx[:-1]

        # ...and their ends
        to_idx = np.zeros_like(from_idx)
        for i in range(len(from_idx) - 1):
            to_idx[i] = from_idx[i+1] - 1
        to_idx[-1] = n - 1

        cross_val = np.zeros(len(sigmas))

```

```

# Cycling through the possible values of Sigma
for sig_index, sigma in enumerate(sigmas):
    print(f"Computing cross-validation metric for Sigma={sigma:.4f}...")

    cross_val[sig_index] = 0

    # Model fitting to subsets of the data
    for j in range(len(from_idx)):
        removed = xs[from_idx[j]:to_idx[j]+1]
        xs_temp = np.delete(xs, np.arange(from_idx[j], to_idx[j]+1))

        # Create a temporary model for CV
        temp_model = Rbf(
            X=X_orig[xs_temp],
            y=y_orig[xs_temp],
            code=self.code
        )
        temp_model.sigma = sigma

        # Sigma and subset chosen, now estimate w
        temp_model.estim_weights()

        if temp_model.weights is None:
            cross_val[sig_index] = 1e20
            xs = full_xs.copy()
            break

        # Compute vector of predictions at the removed sites
        pr = np.zeros(len(removed))
        for jj, idx in enumerate(removed):
            pr[jj] = temp_model.predict(X_orig[idx])

        # Calculate cross-validation error
        cross_val[sig_index] += np.sum((y_orig[removed] - pr)**2) / len(r

    xs = full_xs.copy()

    # Now attempt Cholesky on the full set, in case the subsets could
    # be fitted correctly, but the complete X could not
    temp_model = Rbf(
        X=X_orig,
        y=y_orig,
        code=self.code
    )

```

```

        temp_model.sigma = sigma
        temp_model.estim_weights()

        if temp_model.weights is None:
            cross_val[sig_index] = 1e20
            print("Failed to fit complete sample data.")

        # Find the best sigma
        min_cv_index = np.argmin(cross_val)
        best_sig = sigmas[min_cv_index]

        # Set the best sigma and recompute weights
        print(f"Selected sigma={best_sig:.4f}")
        self.sigma = best_sig
        self.estim_weights()

    return self

def predict(self, x):
    """Calculates the value of the Radial Basis Function surrogate model at x.

    Args:
        x (ndarray): Point at which to make prediction

    Returns:
        float: Predicted value
    """
    # Calculate distances to all sample points
    d = np.zeros(len(self.X))
    for k in range(len(self.X)):
        d[k] = np.linalg.norm(x - self.X[k])

    # Calculate basis function values
    phi = np.zeros(len(self.X))
    for k in range(len(self.X)):
        phi[k] = self.basis(d[k], self.sigma)

    # Calculate prediction
    y = np.dot(phi, self.weights)
    return y

```

70.3. RBF Example: The One-Dimensional sin Function

```

import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import cholesky, cho_solve

# Define the data points for fitting
x_centers = np.array([np.pi/2, np.pi, 3*np.pi/2]).reshape(-1, 1) # Centers for RBFs
y_values = np.sin(x_centers.flatten()) # Sine values at these points

# Create and fit the RBF model
rbf_model = Rbf(X=x_centers, y=y_values, code=4) # Code 4 is Gaussian RBF
rbf_model.sigma = 1.0 # Set sigma parameter directly
rbf_model.estimate_weights() # Calculate optimal weights

# Print the weights
print("RBF model weights:", rbf_model.weights)

# Create a grid for visualization
x_grid = np.linspace(0, 2*np.pi, 1000).reshape(-1, 1)
y_true = np.sin(x_grid.flatten()) # True sine function

# Generate predictions using the RBF model
y_pred = np.zeros(len(x_grid))
for i in range(len(x_grid)):
    y_pred[i] = rbf_model.predict(x_grid[i])

# Calculate individual basis functions for visualization
basis_funcs = np.zeros((len(x_grid), len(x_centers)))
for i in range(len(x_grid)):
    for j in range(len(x_centers)):
        # Calculate distance
        distance = np.linalg.norm(x_grid[i] - x_centers[j])
        # Compute basis function value scaled by its weight
        basis_funcs[i, j] = rbf_model.basis(distance, rbf_model.sigma) * rbf_model.weights[j]

# Plot the results
plt.figure(figsize=(6, 4))

# Plot the true sine function
plt.plot(x_grid, y_true, 'k-', label='True sine function', linewidth=2)

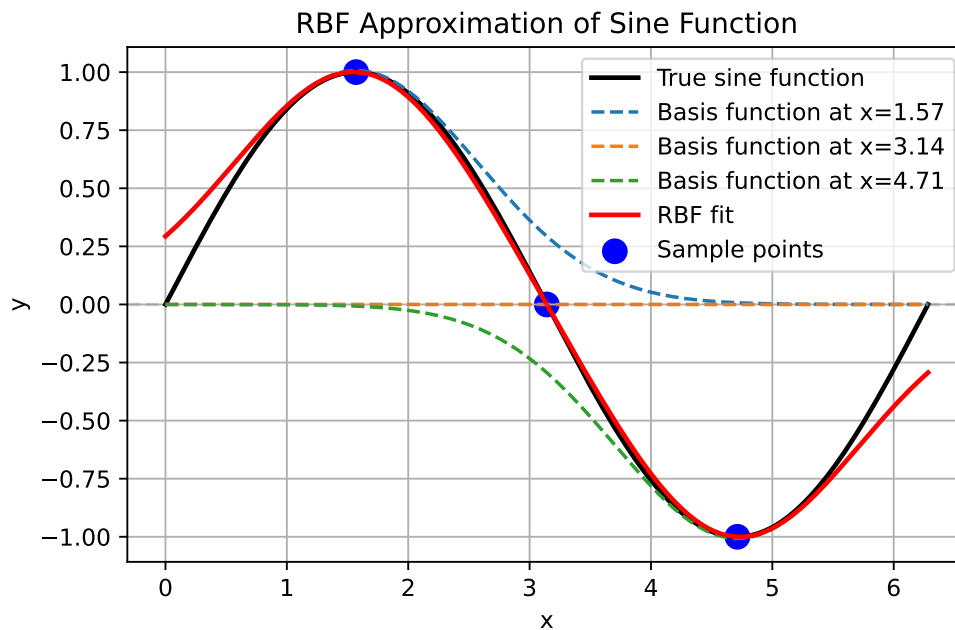
# Plot individual basis functions

```


70.3. RBF Example: The One-Dimensional $\sin$ Function

```
for i in range(len(x_centers)):  
    plt.plot(x_grid, basis_funcs[:, i], '--',  
             label=f'Basis function at x={x_centers[i][0]:.2f}')  
  
# Plot the RBF fit (sum of basis functions)  
plt.plot(x_grid, y_pred, 'r-', label='RBF fit', linewidth=2)  
  
# Plot the sample points  
plt.scatter(x_centers, y_values, color='blue', s=100, label='Sample points')  
  
# Add horizontal line at y=0  
plt.axhline(y=0, color='gray', linestyle='--', alpha=0.3)  
  
plt.title('RBF Approximation of Sine Function')  
plt.xlabel('x')  
plt.ylabel('y')  
plt.grid(True)  
plt.legend()  
plt.tight_layout()  
plt.show()
```

RBF model weights: [1.00724398e+00 2.32104414e-16 -1.00724398e+00]



70.4. RBF Example: The Two-Dimensional dome Function

The `dome` function is an example of a test function that can be used to evaluate the performance of the Radial Basis Function model. It is a simple mathematical function defined over a two-dimensional space.

```
def dome(x) -> float:
    """
    Dome test function.

    Args:
        x (ndarray): Input vector (1D array of length 2)

    Returns:
        float: Function value

    Examples:
        dome(np.array([0.5, 0.5]))
    """
    return np.sum(1 - (2*x - 1)**2) / len(x)
```

The following code demonstrates how to use the Radial Basis Function model to approximate a function. It generates a Latin Hypercube sample, computes the objective function values, estimates the model parameters, and plots the results.

```
def generate_rbf_data(n_samples=10, grid_points=41):
    """
    Generates data for RBF visualization.

    Args:
        n_samples (int): Number of samples for the RBF model
        grid_points (int): Number of grid points for prediction

    Returns:
        tuple: (rbf_model, X, Y, Z, Z_0) - Model and grid data for plotting
    """
    from spotpython.utils.sampling import bestlh as best_lh
    # Generate sampling plan
    X_samples = best_lh(n_samples, 2, population=10, iterations=100)
    # Compute objective function values
    y_samples = np.zeros(len(X_samples))
    for i in range(len(X_samples)):
```

70.4. RBF Example: The Two-Dimensional dome Function

```
    y_samples[i] = dome(X_samples[i])
# Create and fit RBF model
rbf_model = Rbf(X=X_samples, y=y_samples, code=3) # Thin plate spline
rbf_model.fit()
# Generate grid for prediction
x = np.linspace(0, 1, grid_points)
y = np.linspace(0, 1, grid_points)
X, Y = np.meshgrid(x, y)
Z_0 = np.zeros_like(X)
Z = np.zeros_like(X)

# Evaluate model at grid points
for i in range(len(x)):
    for j in range(len(y)):
        Z_0[j, i] = dome(np.array([x[i], y[j]]))
        Z[j, i] = rbf_model.predict(np.array([x[i], y[j]]))

return rbf_model, X, Y, Z, Z_0

def plot_rbf_results(rbf_model, X, Y, Z, Z_0=None, n_contours=10):
    """
    Plots RBF approximation results.

    Args:
        rbf_model (Rbf): Fitted RBF model
        X (ndarray): Grid X-coordinates
        Y (ndarray): Grid Y-coordinates
        Z (ndarray): RBF model predictions
        Z_0 (ndarray, optional): True function values for comparison
        n_contours (int): Number of contour levels to plot
    """
    import matplotlib.pyplot as plt

    plt.figure(figsize=(10, 8))

    # Plot the contour
    contour = plt.contour(X, Y, Z, n_contours)

    if Z_0 is not None:
        contour_0 = plt.contour(X, Y, Z_0, n_contours, colors='k', linestyle='dashed')

    # Plot the sample points
    plt.scatter(rbf_model.X[:, 0], rbf_model.X[:, 1],
               c='r', marker='o', s=50)
```

```
plt.title('RBF Approximation (Thin Plate Spline)')
plt.xlabel('x1')
plt.ylabel('x2')
plt.colorbar(label='f(x1, x2)')
plt.show()
```

Figure 70.3 shows the contour plots of the underlying function $f(x_1, x_2) = 0.5[-(2x_1 - 1)^2 - (2x_2 - 1)^2]$ and its thin plate spline radial basis function approximation, along with the 10 points of a Morris-Mitchell optimal Latin hypercube sampling plan (obtained via `best_lh()`).

```
rbf_model, X, Y, Z, Z_0 = generate_rbf_data(n_samples=10, grid_points=41)
plot_rbf_results(rbf_model, X, Y, Z, Z_0)
```

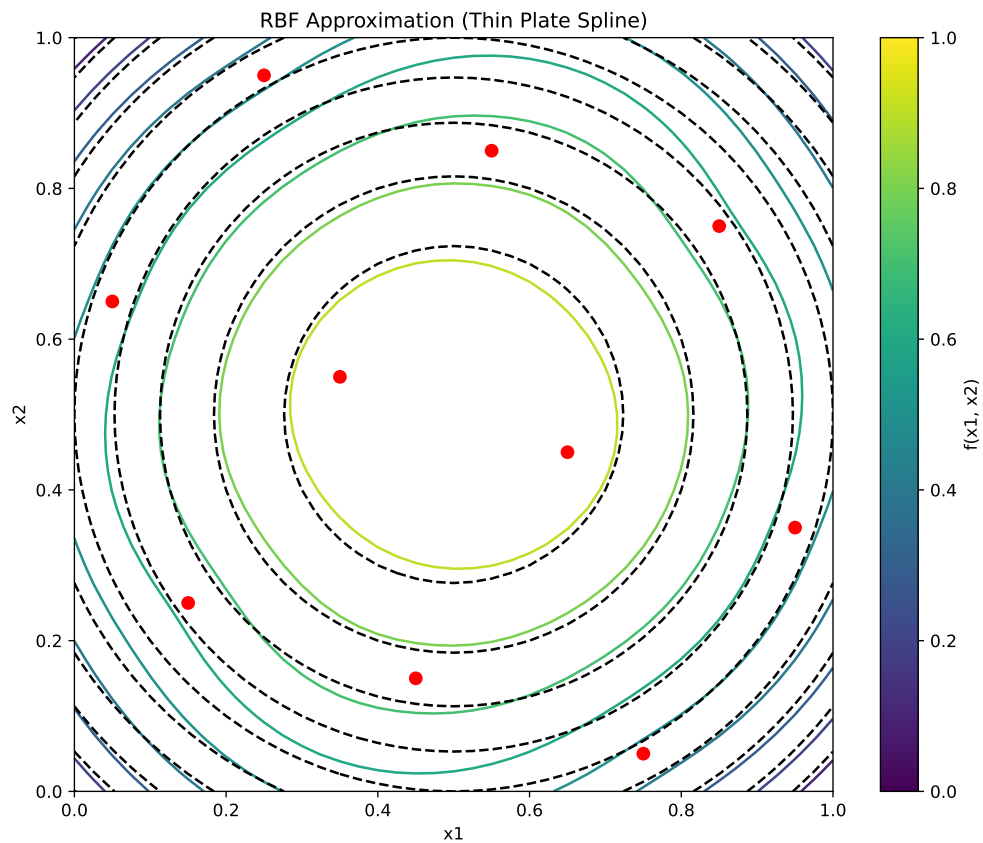


Figure 70.3.: RBF Approximation.

70.4.1. The Connection Between RBF Models and Neural Networks

Radial basis function models share a profound architectural similarity with artificial neural networks, specifically with what's known as RBF networks. This connection provides valuable intuition about how RBF models function. A radial basis function model can be viewed as a specialized neural network with the following structure:

- Input Layer: Receives the feature vector $\vec{x}$
- Hidden Layer: Contains neurons (basis functions) that compute radial distances
- Output Layer: Produces a weighted sum of the hidden unit activations

Unlike traditional neural networks that use dot products followed by nonlinear activation functions, RBF networks measure the distance between inputs and learned center points. This distance is then transformed by the radial basis function.

Mathematically, the equivalence between RBF models and RBF networks can be expressed as follows:

The RBF model equation:

$$\hat{f}(\vec{x}) = \sum_{i=1}^{n_c} w_i \psi(\|\vec{x} - \vec{c}^{(i)}\|)$$

directly maps to the following neural network components:

- $\vec{x}$: Input vector
- $\vec{c}^{(i)}$: Center vectors for each hidden neuron
- $\psi(\cdot)$: Activation function (Gaussian, inverse multiquadric, etc.)
- w_i : Output weights
- $\hat{f}(\vec{x})$: Network output

Example 70.2 (Comparison of RBF Networks and Traditional Neural Networks). Consider approximating a simple 1D function $f(x) = \sin(2\pi x)$ over the interval $[0, 1]$:

The neural network approach would use multiple layers with neurons computing $\sigma(w \cdot x + b)$. It would require a large number of neurons and layers to capture the sine wave's complexity. The network would learn both weights and biases, making it less interpretable.

The RBF network approach, on the other hand, places basis functions at strategic points (e.g., 5 evenly spaced centers). Each neuron computes $\psi(\|x - c_i\|)$ (e.g., using Gaussian RBF). The output layer combines these values with learned weights. If we place Gaussian RBFs with $\sigma = 0.15$ at 0.1, 0.3, 0.5, 0.7, 0.9, each neuron responds strongly when the input is close to its center and weakly otherwise. The network can then learn weights that, when multiplied by these response patterns and summed, closely approximate the sine function.

70. Radial Basis Function Models

This locality property gives RBF networks a notable advantage: they offer more interpretable internal representations and often require fewer neurons for certain types of function approximation compared to traditional multilayer perceptrons.

The key insight is that while standard neural networks create complex decision boundaries through compositions of hyperplanes, RBF networks directly model functions using a set of overlapping “bumps” positioned strategically in the input space.

70.5. Radial Basis Function Models for Noisy Data

When the responses $\vec{y} = y^{(1)}, y^{(2)}, \dots, y^{(n)T}$ contain measurement or simulation noise, the standard RBF interpolation approach can lead to overfitting—the model captures both the underlying function and the random noise. This compromises generalization performance on new data points. Two principal strategies address this challenge:

70.5.1. Ridge Regularization Approach

The most straightforward solution involves introducing regularization through the parameter λ (Poggio and Girosi 1990). This is implemented by adding λ to the diagonal elements of the Gram matrix, creating a “ridge” that improves numerical stability. Mathematically, the weights are determined by:

$$\vec{w} = (\Psi + \lambda I)^{-1} \vec{y},$$

where I is an $n \times n$ identity matrix. This regularized solution balances two competing objectives:

- fitting the training data accurately versus
- keeping the magnitude of weights controlled to prevent overfitting.

Theoretically, optimal performance is achieved when λ equals the variance of the noise in the response data $\vec{y}$ (Keane and Nair 2005). Since this information is rarely available in practice, λ is typically estimated through cross-validation alongside other model parameters.

70.5.2. Reduced Basis Approach

An alternative strategy involves reducing m , the number of basis functions. This might result in a non-square Ψ matrix. With a non-square Ψ matrix, the weights are found through least squares minimization:

$$\vec{w} = (\Psi^T \Psi)^{-1} \Psi^T \vec{y}$$

This approach introduces an important design decision: which subset of points should serve as basis function centers? Several selection strategies exist:

- Clustering methods that identify representative points
- Greedy algorithms that sequentially select influential centers
- Support vector regression techniques (discussed elsewhere in the literature)

Additional parameters such as the width parameter σ in Gaussian bases can be optimized through cross-validation to minimize generalization error estimates.

The ridge regularization and reduced basis approaches can be combined, allowing for a flexible modeling framework, though at the cost of a more complex parameter estimation process. This hybrid approach often yields superior results for highly noisy datasets or when the underlying function has varying complexity across the input space.

The broader challenge of building accurate models from noisy observations is examined comprehensively in the context of Kriging models, which provide a statistical framework for explicitly modeling both the underlying function and the noise process.

70.6. Jupyter Notebook

i Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

71. Kriging (Gaussian Process Regression)

i Note

- This section is based on chapter 2.4 in Forrester, Sóbester, and Keane (2008).
- The following Python packages are imported:

```
import matplotlib.pyplot as plt
import numpy as np
from numpy import (array, zeros, power, ones, exp, multiply,
                  eye, linspace, spacing, sqrt, arange,
                  append, ravel)
from numpy.linalg import cholesky, solve
from scipy.spatial.distance import squareform, pdist, cdist
```

71.1. From Gaussian RBF to Kriging Basis Functions

Kriging can be explained using the concept of radial basis functions (RBFs), which were introduced in Chapter 70. An RBF is a real-valued function whose value depends only on the distance from a certain point, called the center, usually in a multidimensional space. The basis function is a function of the distance between the point $\vec{x}$ and the center $\vec{x}^{(i)}$. Other names for basis functions are *kernel* or *covariance* functions.

Definition 71.1 (The Kriging Basis Functions). Kriging (also known as Gaussian Process Regression) uses k -dimensional basis functions of the form

$$\psi^{(i)}(\vec{x}) = \psi(\vec{x}^{(i)}, \vec{x}) = \exp \left(- \sum_{j=1}^k \theta_j |x_j^{(i)} - x_j|^{p_j} \right), \quad (71.1)$$

where $\vec{x}$ and $\vec{x}^{(i)}$ denote the k -dim vector $\vec{x} = (x_1, \dots, x_k)^T$ and $\vec{x}^{(i)} = (x_1^{(i)}, \dots, x_k^{(i)})^T$, respectively.

□

71. Kriging (Gaussian Process Regression)

Kriging uses a specialized basis function that offers greater flexibility than standard RBFs. Examining Equation 71.1, we can observe how Kriging builds upon and extends the Gaussian basis concept. The key enhancements of Kriging over Gaussian RBF can be summarized as follows:

- Dimension-specific width parameters: While a Gaussian RBF uses a single width parameter $1/\sigma^2$, Kriging employs a vector $\vec{\theta} = (\theta_1, \theta_2, \dots, \theta_k)^T$. This allows the model to automatically adjust its sensitivity to each input dimension, effectively performing automatic feature relevance determination.
- Flexible smoothness control: The Gaussian RBF fixes the exponent at 2, producing uniformly smooth functions. In contrast, Kriging's dimension-specific exponents $\vec{p} = (p_1, p_2, \dots, p_k)^T$ (typically with $p_j \in [1, 2]$) enable precise control over smoothness properties in each dimension.
- Unifying framework: When all exponents are set to $p_j = 2$ and all width parameters are equal ($\theta_j = 1/\sigma^2$ for all j), the Kriging basis function reduces exactly to the Gaussian RBF. This makes Gaussian RBF a special case within the more general Kriging framework.

These enhancements make Kriging particularly well-suited for engineering problems where variables may operate at different scales or exhibit varying degrees of smoothness across dimensions. For now, we will only consider Kriging interpolation. We will cover Kriging regression later.

71.2. Building the Kriging Model

Consider sample data X and $\vec{y}$ from n locations that are available in matrix form: X is a $(n \times k)$ matrix, where k denotes the problem dimension and $\vec{y}$ is a $(n \times 1)$ vector. We want to find an expression for predicted values at a new point $\vec{x}$, denoted as $\hat{y}$.

We start with an abstract, not really intuitive concept: The observed responses $\vec{y}$ are considered as if they are from a stochastic process, which will be denoted as

$$\begin{pmatrix} Y(\vec{x}^{(1)}) \\ \vdots \\ Y(\vec{x}^{(n)}) \end{pmatrix}. \quad (71.2)$$

The set of random vectors from Equation 71.2 (also referred to as a *random field*) has a mean of $\vec{1}\mu$, which is a $(n \times 1)$ vector. The random vectors are correlated with each other using the basis function expression from Equation 71.1:

$$\text{cor}(Y(\vec{x}^{(i)}), Y(\vec{x}^{(l)})) = \exp\left(-\sum_{j=1}^k \theta_j |x_j^{(i)} - x_j^{(l)}|^{p_j}\right). \quad (71.3)$$

Using Equation 71.3, we can compute the $(n \times n)$ correlation matrix Ψ of the observed sample data as shown in Equation 71.4,

$$\Psi = \begin{pmatrix} \text{cor}(Y(\vec{x}^{(1)}), Y(\vec{x}^{(1)})) & \dots & \text{cor}(Y(\vec{x}^{(1)}), Y(\vec{x}^{(n)})) \\ \vdots & \ddots & \vdots \\ \text{cor}(Y(\vec{x}^{(n)}), Y(\vec{x}^{(1)})) & \dots & \text{cor}(Y(\vec{x}^{(n)}), Y(\vec{x}^{(n)})) \end{pmatrix}, \quad (71.4)$$

and a covariance matrix as shown in Equation 71.5,

$$\text{Cov}(Y, Y) = \sigma^2 \Psi. \quad (71.5)$$

This assumed correlation between the sample data reflects our expectation that an engineering function will behave in a certain way and it will be smoothly and continuous.

Remark 71.1 (Note on Stochastic Processes). See `?@sec-random-samples-gp` for a more detailed discussion on realizations of stochastic processes.

□

We now have a set of n random variables ($\mathbf{Y}$) that are correlated with each other as described in the $(n \times n)$ correlation matrix Ψ , see Equation 71.4. The correlations depend on the absolute distances in dimension j between the i -th and the l -th sample point $|x_j^{(i)} - x_j^{(l)}|$ and the corresponding parameters p_j and θ_j for dimension j . The correlation is intuitive, because when

- two points move close together, then $|x_j^{(i)} - x_j| \rightarrow 0$ and $\exp(-|x_j^{(i)} - x_j|^{p_j}) \rightarrow 1$ (these points show very close correlation and $Y(x_j^{(i)}) = Y(x_j)$).
- two points move far apart, then $|x_j^{(i)} - x_j| \rightarrow \infty$ and $\exp(-|x_j^{(i)} - x_j|^{p_j}) \rightarrow 0$ (these points show very low correlation).

Example 71.1 (Correlations for different p_j). Three different correlations are shown in Figure 71.1: $p_j = 0.1, 1, 2$. The smoothness parameter p_j affects the correlation:

- With $p_j = 0.1$, there is basically no immediate correlation between the points and there is a near discontinuity between the points $Y(\vec{x}_j^{(i)})$ and $Y(\vec{x}_j)$.
- With $p_j = 2$, the correlation is more smooth and we have a continuous gradient through $x_j^{(i)} - x_j$.

Reducing p_j increases the rate at which the correlation initially drops with distance. This is shown in Figure 71.1.

□

Example 71.2 (Correlations for different θ). Figure 71.2 visualizes the correlation between two points $Y(\vec{x}_j^{(i)})$ and $Y(\vec{x}_j)$ for different values of θ . The parameter θ can be seen as a *width* parameter:

71. Kriging (Gaussian Process Regression)

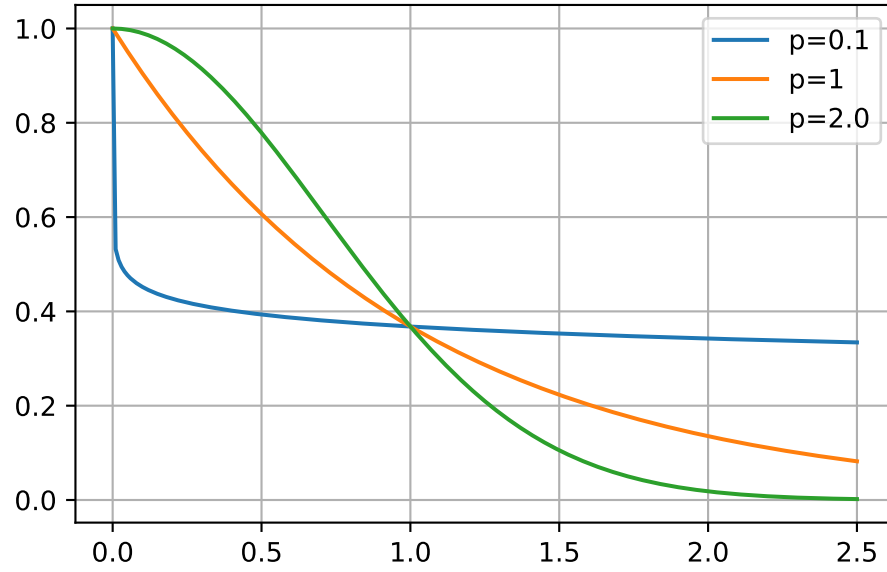


Figure 71.1.: Correlations with varying p . θ set to 1.

- low θ_j means that all points will have a high correlation, with $Y(x_j)$ being similar across the sample.
- high θ_j means that there is a significant difference between the $Y(x_j)$'s.
- θ_j is a measure of how *active* the function we are approximating is.
- High θ_j indicate important parameters, see Figure 71.2.

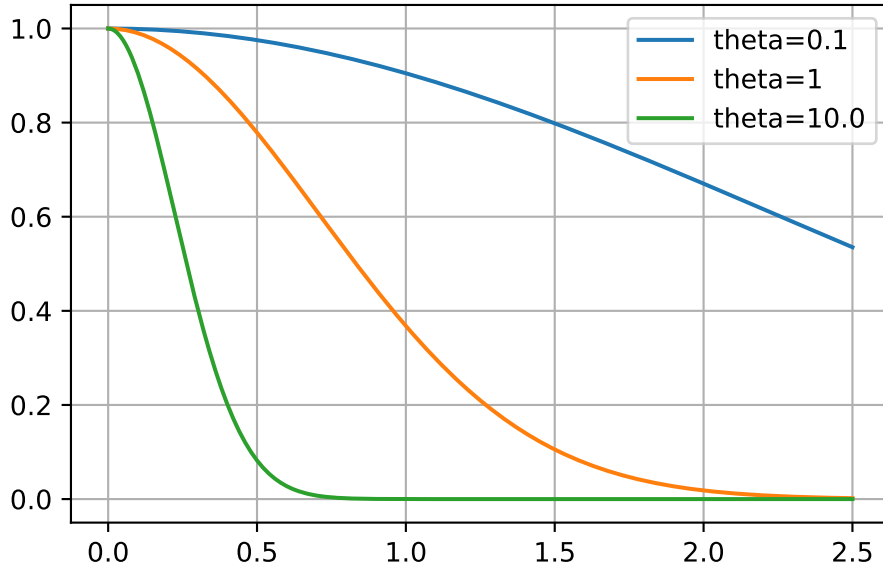
□

Considering the activity parameter θ is useful in high-dimensional problems where it is difficult to visualize the design landscape and the effect of the variable is unknown. By examining the elements of the vector θ , we can identify the most important variables and focus on them. This is a crucial step in the optimization process, as it allows us to reduce the dimensionality of the problem and focus on the most important variables.

Example 71.3 (The Correlation Matrix (Detailed Computation)). Let $n = 4$ and $k = 3$. The sample plan is represented by the following matrix X :

$$X = \begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \\ x_{41} & x_{42} & x_{43} \end{pmatrix}$$

To compute the elements of the matrix Ψ , the following k (one for each of the k dimensions) (n, n) -matrices have to be computed:

Figure 71.2.: Correlations with varying θ . p set to 2.

- For $k = 1$, i.e., the first column of X :

$$D_1 = \begin{pmatrix} x_{11} - x_{11} & x_{11} - x_{21} & x_{11} - x_{31} & x_{11} - x_{41} \\ x_{21} - x_{11} & x_{21} - x_{21} & x_{21} - x_{31} & x_{21} - x_{41} \\ x_{31} - x_{11} & x_{31} - x_{21} & x_{31} - x_{31} & x_{31} - x_{41} \\ x_{41} - x_{11} & x_{41} - x_{21} & x_{41} - x_{31} & x_{41} - x_{41} \end{pmatrix}$$

- For $k = 2$, i.e., the second column of X :

$$D_2 = \begin{pmatrix} x_{12} - x_{12} & x_{12} - x_{22} & x_{12} - x_{32} & x_{12} - x_{42} \\ x_{22} - x_{12} & x_{22} - x_{22} & x_{22} - x_{32} & x_{22} - x_{42} \\ x_{32} - x_{12} & x_{32} - x_{22} & x_{32} - x_{32} & x_{32} - x_{42} \\ x_{42} - x_{12} & x_{42} - x_{22} & x_{42} - x_{32} & x_{42} - x_{42} \end{pmatrix}$$

- For $k = 3$, i.e., the third column of X :

$$D_3 = \begin{pmatrix} x_{13} - x_{13} & x_{13} - x_{23} & x_{13} - x_{33} & x_{13} - x_{43} \\ x_{23} - x_{13} & x_{23} - x_{23} & x_{23} - x_{33} & x_{23} - x_{43} \\ x_{33} - x_{13} & x_{33} - x_{23} & x_{33} - x_{33} & x_{33} - x_{43} \\ x_{43} - x_{13} & x_{43} - x_{23} & x_{43} - x_{33} & x_{43} - x_{43} \end{pmatrix}$$

Since the matrices are symmetric and the main diagonals are zero, it is sufficient to

71. Kriging (Gaussian Process Regression)

compute the following matrices:

$$D_1 = \begin{pmatrix} 0 & x_{11} - x_{21} & x_{11} - x_{31} & x_{11} - x_{41} \\ 0 & 0 & x_{21} - x_{31} & x_{21} - x_{41} \\ 0 & 0 & 0 & x_{31} - x_{41} \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$D_2 = \begin{pmatrix} 0 & x_{12} - x_{22} & x_{12} - x_{32} & x_{12} - x_{42} \\ 0 & 0 & x_{22} - x_{32} & x_{22} - x_{42} \\ 0 & 0 & 0 & x_{32} - x_{42} \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$D_3 = \begin{pmatrix} 0 & x_{13} - x_{23} & x_{13} - x_{33} & x_{13} - x_{43} \\ 0 & 0 & x_{23} - x_{33} & x_{23} - x_{43} \\ 0 & 0 & 0 & x_{33} - x_{43} \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

We will consider $p_l = 2$. The differences will be squared and multiplied by θ_i , i.e.:

$$D_1 = \theta_1 \begin{pmatrix} 0 & (x_{11} - x_{21})^2 & (x_{11} - x_{31})^2 & (x_{11} - x_{41})^2 \\ 0 & 0 & (x_{21} - x_{31})^2 & (x_{21} - x_{41})^2 \\ 0 & 0 & 0 & (x_{31} - x_{41})^2 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$D_2 = \theta_2 \begin{pmatrix} 0 & (x_{12} - x_{22})^2 & (x_{12} - x_{32})^2 & (x_{12} - x_{42})^2 \\ 0 & 0 & (x_{22} - x_{32})^2 & (x_{22} - x_{42})^2 \\ 0 & 0 & 0 & (x_{32} - x_{42})^2 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$D_3 = \theta_3 \begin{pmatrix} 0 & (x_{13} - x_{23})^2 & (x_{13} - x_{33})^2 & (x_{13} - x_{43})^2 \\ 0 & 0 & (x_{23} - x_{33})^2 & (x_{23} - x_{43})^2 \\ 0 & 0 & 0 & (x_{33} - x_{43})^2 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

The sum of the three matrices $D = D_1 + D_2 + D_3$ will be calculated next:

$$\begin{pmatrix} 0 & \theta_1(x_{11} - x_{21})^2 + \theta_2(x_{12} - x_{22})^2 + \theta_3(x_{13} - x_{23})^2 & \theta_1(x_{11} - x_{31})^2 + \theta_2(x_{12} - x_{32})^2 + \theta_3(x_{13} - x_{33})^2 & \theta_1(x_{11} - x_{41})^2 + \theta_2(x_{12} - x_{42})^2 + \theta_3(x_{13} - x_{43})^2 \\ 0 & 0 & \theta_1(x_{21} - x_{31})^2 + \theta_2(x_{22} - x_{32})^2 + \theta_3(x_{23} - x_{33})^2 & \theta_1(x_{21} - x_{41})^2 + \theta_2(x_{22} - x_{42})^2 + \theta_3(x_{23} - x_{43})^2 \\ 0 & 0 & 0 & \theta_1(x_{31} - x_{41})^2 + \theta_2(x_{32} - x_{42})^2 + \theta_3(x_{33} - x_{43})^2 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Finally,

$$\Psi = \exp(-D)$$

is computed.

Next, we will demonstrate how this computation can be implemented in Python. We will consider four points in three dimensions and compute the correlation matrix Ψ using the basis function from Equation 71.1. These points are placed at the origin, at the unit vectors, and at the points (100,100,100) and (101,100,100). So, they form two clusters: one at the origin and one at (100,100,100).

```
theta = np.array([1,2,3])
X = np.array([ [1,0,0], [0,1,0], [100, 100, 100], [101, 100, 100]])
X
```

```
array([[ 1,  0,  0],
       [ 0,  1,  0],
       [100, 100, 100],
       [101, 100, 100]])
```

```
def build_Psi(X, theta):
    n = X.shape[0]
    k = X.shape[1]
    D = zeros((k, n, n))
    for l in range(k):
        for i in range(n):
            for j in range(i, n):
                D[l, i, j] = theta[l]*(X[i,l] - X[j,l])**2
    D = sum(D)
    D = D + D.T
    return exp(-D)
```

```
Psi = build_Psi(X, theta)
Psi
```

```
array([[1.          , 0.04978707, 0.          , 0.          ],
       [0.04978707, 1.          , 0.          , 0.          ],
       [0.          , 0.          , 1.          , 0.36787944],
       [0.          , 0.          , 0.36787944, 1.          ]])
```

□

Example 71.4 (Example: The Correlation Matrix (Using Existing Functions)). The same result as computed in Example 71.3 can be obtained with existing python functions, e.g., from the package `scipy`.

71. Kriging (Gaussian Process Regression)

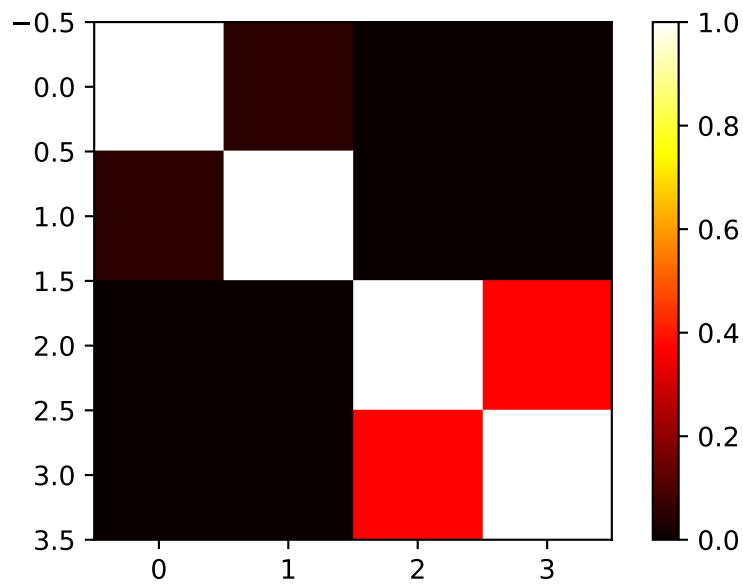


Figure 71.3.: Correlation matrix Ψ .

```
def build_Psi(X, theta, eps=sqrt(spacing(1))):
    return exp(- squareform(pdist(X,
                                   metric='sqeuclidean',
                                   out=None,
                                   w=theta))) + multiply(eye(X.shape[0]),
                                                         eps)

Psi = build_Psi(X, theta, eps=.0)
Psi
```

```
array([[1.00000000, 0.04978707, 0.00000000, 0.00000000],
       [0.04978707, 1.00000000, 0.00000000, 0.00000000],
       [0.00000000, 0.00000000, 1.00000000, 0.36787944],
       [0.00000000, 0.00000000, 0.36787944, 1.00000000]])
```

The condition number of the correlation matrix Ψ is a measure of how well the matrix can be inverted. A high condition number indicates that the matrix is close to singular, which can lead to numerical instability in computations involving the inverse of the matrix, see Section 72.2.


```
np.linalg.cond(Psi)
```

```
np.float64(2.163953413738652)
```

□

71.3. MLE to estimate θ and p

71.3.1. The Log-Likelihood

Until now, the observed data $\vec{y}$ was not used. We know what the correlations mean, but how do we estimate the values of θ_j and where does our observed data y come in? To estimate the values of $\vec{\theta}$ and $\vec{p}$, they are chosen to maximize the likelihood of $\vec{y}$,

$$L = L(Y(\vec{x}^{(1)}), \dots, Y(\vec{x}^{(n)}) | \mu, \sigma) = \frac{1}{(2\pi\sigma^2)^{n/2}} \exp \left[-\frac{\sum_{i=1}^n (Y(\vec{x}^{(i)}) - \mu)^2}{2\sigma^2} \right], \quad (71.6)$$

where μ is the mean of the observed data $\vec{y}$ and σ is the standard deviation of the errors ϵ , which can be expressed in terms of the sample data

$$L = \frac{1}{(2\pi\sigma^2)^{n/2} |\vec{\Psi}|^{1/2}} \exp \left[-\frac{(\vec{y} - \vec{1}\mu)^T \vec{\Psi}^{-1} (\vec{y} - \vec{1}\mu)}{2\sigma^2} \right]. \quad (71.7)$$

Remark 71.2. The transition from Equation 71.6 to Equation 71.7 reflects a shift from assuming independent errors in the observed data to explicitly modeling the *correlation structure* between the observed responses, which is a key aspect of the stochastic process framework used in methods like Kriging. It can be explained as follows:

1. **Initial Likelihood Expression (assuming independent errors):** Equation 71.6 is an expression for the likelihood of the data set, which is based on the assumption that the errors ϵ are *independently randomly distributed according to a normal distribution with standard deviation σ* . This form is characteristic of the likelihood of n independent observations $Y(\vec{x}^{(i)})$, each following a normal distribution with mean μ and variance σ^2 .
2. **Using Vector Notation.** The sum in the exponent, i.e.,

$$\sum_{i=1}^n (Y(\vec{x}^{(i)}) - \mu)^2$$

is equivalent to

$$(\vec{y} - \vec{1}\mu)^T (\vec{y} - \vec{1}\mu),$$

assuming $Y(\vec{x}^{(i)}) = y^{(i)}$ and using vector notation for $\vec{y}$ and $\vec{1}\mu$.

71. Kriging (Gaussian Process Regression)

3. **Assuming Independent Observations:** Equation 71.6 assumes that the observations are independent, which means that the covariance between any two observations $Y(\vec{x}^{(i)})$ and $Y(\vec{x}^{(l)})$ is zero for $i \neq l$. In this case, the covariance matrix of the observations would be a diagonal matrix with σ^2 along the diagonal (i.e., $\sigma^2 I$), where I is the identity matrix.
4. **Stochastic Process and Correlation:** In the context of Kriging, the observed responses $\vec{y}$ are considered as if they are from a *stochastic process* or *random field*. This means the random variables $Y(\vec{x}^{(i)})$ at different locations $\vec{x}^{(i)}$ are not independent, but they correlated with each other. This correlation is described by an $(n \times n)$ **correlation matrix** Ψ , which is used instead of $\sigma^2 I$. The strength of the correlation between two points $Y(\vec{x}^{(i)})$ and $Y(\vec{x}^{(l)})$ depends on the distance between them and model parameters θ_j and p_j .
5. **Multivariate Normal Distribution:** When random variables are correlated, their joint probability distribution is generally described by a *multivariate distribution*. Assuming the stochastic process follows a Gaussian process, the joint distribution of the observed responses $\vec{y}$ is a **multivariate normal distribution**. A multivariate normal distribution for a vector $\vec{Y}$ with mean vector $\vec{\mu}$ and covariance matrix Σ has a probability density function given by:

$$p(\vec{y}) = \frac{1}{\sqrt{(2\pi)^n |\Sigma|}} \exp \left[-\frac{1}{2} (\vec{y} - \vec{\mu})^T \Sigma^{-1} (\vec{y} - \vec{\mu}) \right].$$

6. **Connecting the Expressions:** In the stochastic process framework, the following holds:

- The mean vector of the observed data $\vec{y}$ is $\vec{1}\mu$.
- The covariance matrix Σ is constructed by considering both the variance σ^2 and the correlations Ψ .
- The covariance between $Y(\vec{x}^{(i)})$ and $Y(\vec{x}^{(l)})$ is $\sigma^2 \text{cor}(Y(\vec{x}^{(i)}), Y(\vec{x}^{(l)}))$.
- Therefore, the covariance matrix is $\Sigma = \sigma^2 \vec{\Psi}$.
- Substituting $\vec{\mu} = \vec{1}\mu$ and $\Sigma = \sigma^2 \vec{\Psi}$ into the multivariate normal PDF formula, we get:

$$\Sigma^{-1} = (\sigma^2 \vec{\Psi})^{-1} = \frac{1}{\sigma^2} \vec{\Psi}^{-1}$$

and

$$|\Sigma| = |\sigma^2 \vec{\Psi}| = (\sigma^2)^n |\vec{\Psi}|.$$

The PDF becomes:

$$p(\vec{y}) = \frac{1}{\sqrt{(2\pi)^n (\sigma^2)^n |\vec{\Psi}|}} \exp \left[-\frac{1}{2} (\vec{y} - \vec{1}\mu)^T \left(\frac{1}{\sigma^2} \vec{\Psi}^{-1} \right) (\vec{y} - \vec{1}\mu) \right]$$

and simplifies to:

$$p(\vec{y}) = \frac{1}{(2\pi\sigma^2)^{n/2} |\vec{\Psi}|^{1/2}} \exp \left[-\frac{1}{2\sigma^2} (\vec{y} - \vec{1}\mu)^T \vec{\Psi}^{-1} (\vec{y} - \vec{1}\mu) \right].$$

This is the **likelihood of the sample data** $\vec{y}$ given the parameters μ , σ , and the correlation structure defined by the parameters within $\vec{\Psi}$ (i.e., $\vec{\theta}$ and $\vec{p}$).

In summary, the Equation 71.6 represents the likelihood under a simplified assumption of independent errors, whereas Equation 71.7 is the likelihood derived from the assumption that the observed data comes from a **multivariate normal distribution** where observations are correlated according to the matrix $\vec{\Psi}$. Equation 71.7, using the sample data vector $\vec{y}$ and the correlation matrix $\vec{\Psi}$, properly accounts for the dependencies between data points inherent in the stochastic process model. Maximizing this likelihood is how the correlation parameters $\vec{\theta}$ and $\vec{p}$ are estimated in Kriging.

□

Equation 71.7 can be formulated as the log-likelihood:

$$\ln(L) = -\frac{n}{2} \ln(2\pi\sigma) - \frac{1}{2} \ln |\vec{\Psi}| - \frac{(\vec{y} - \vec{1}\mu)^T \vec{\Psi}^{-1} (\vec{y} - \vec{1}\mu)}{2\sigma^2}. \quad (71.8)$$

71.3.2. Differentiation with Respect to μ

Looking at the log-likelihood function, only the last term depends on μ :

$$\frac{1}{2\sigma^2} (\vec{y} - \vec{1}\mu)^T \vec{\Psi}^{-1} (\vec{y} - \vec{1}\mu)$$

To differentiate this with respect to the scalar μ , we can use matrix calculus rules.

Let $\mathbf{v} = \vec{y} - \vec{1}\mu$. $\vec{y}$ is a constant vector with respect to μ , and $\vec{1}\mu$ is a vector whose derivative with respect to the scalar μ is $\vec{1}$. So, $\frac{\partial \mathbf{v}}{\partial \mu} = -\vec{1}$.

The term is in the form $\mathbf{v}^T \mathbf{A} \mathbf{v}$, where $\mathbf{A} = \vec{\Psi}^{-1}$ is a symmetric matrix. The derivative of $\mathbf{v}^T \mathbf{A} \mathbf{v}$ with respect to $\mathbf{v}$ is $2\mathbf{A} \mathbf{v}$ as explained in Remark 71.3.

Remark 71.3 (Derivative of a Quadratic Form). Consider the derivative of $\mathbf{v}^T \mathbf{A} \mathbf{v}$ with respect to $\mathbf{v}$:

- The derivative of a scalar function $f(\mathbf{v})$ with respect to a vector $\mathbf{v}$ is a vector (the gradient).
- For a quadratic form $\mathbf{v}^T \mathbf{A} \mathbf{v}$, where $\mathbf{A}$ is a matrix and $\mathbf{v}$ is a vector, the general formula for the derivative with respect to $\mathbf{v}$ is $\frac{\partial}{\partial \mathbf{v}} (\mathbf{v}^T \mathbf{A} \mathbf{v}) = \mathbf{A} \mathbf{v} + \mathbf{A}^T \mathbf{v}$. (This is a standard result in matrix calculus and explained in Equation 72.1).
- Since $\mathbf{A} = \vec{\Psi}^{-1}$ is a symmetric matrix, its transpose $\mathbf{A}^T$ is equal to $\mathbf{A}$.
- Substituting $\mathbf{A}^T = \mathbf{A}$ into the general derivative formula, we get $\mathbf{A} \mathbf{v} + \mathbf{A} \mathbf{v} = 2\mathbf{A} \mathbf{v}$.

□

71. Kriging (Gaussian Process Regression)

Using the chain rule for differentiation with respect to the scalar μ :

$$\frac{\partial}{\partial \mu}(\mathbf{v}^T \mathbf{A} \mathbf{v}) = 2 \left(\frac{\partial \mathbf{v}}{\partial \mu} \right)^T \mathbf{A} \mathbf{v}$$

Substituting $\frac{\partial \mathbf{v}}{\partial \mu} = -\vec{1}$ and $\mathbf{v} = \vec{y} - \vec{1}\mu$:

$$\frac{\partial}{\partial \mu}(\vec{y} - \vec{1}\mu)^T \vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu) = 2(-\vec{1})^T \vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu) = -2\vec{1}^T \vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu)$$

Now, differentiate the full log-likelihood term depending on μ :

$$\frac{\partial}{\partial \mu} \left(-\frac{1}{2\sigma^2}(\vec{y} - \vec{1}\mu)^T \vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu) \right) = -\frac{1}{2\sigma^2} (-2\vec{1}^T \vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu)) = \frac{1}{\sigma^2} \vec{1}^T \vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu)$$

Setting this to zero for maximization gives:

$$\frac{1}{\sigma^2} \vec{1}^T \vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu) = 0.$$

Rearranging gives:

$$\vec{1}^T \vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu) = 0.$$

Solving for μ gives:

$$\vec{1}^T \vec{\Psi}^{-1} \vec{y} = \mu \vec{1}^T \vec{\Psi}^{-1} \vec{1}.$$

71.3.3. Differentiation with Respect to σ

Let $\nu = \sigma^2$ for simpler differentiation notation. The log-likelihood becomes:

$$\ln(L) = C_1 - \frac{n}{2} \ln(\nu) - \frac{(\vec{y} - \vec{1}\mu)^T \vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu)}{2\nu},$$

where $C_1 = -\frac{n}{2} \ln(2\pi) - \frac{1}{2} \ln |\vec{\Psi}|$ is a constant with respect to $\nu = \sigma^2$.

We differentiate with respect to ν :

$$\frac{\partial \ln(L)}{\partial \nu} = \frac{\partial}{\partial \nu} \left(-\frac{n}{2} \ln(\nu) \right) + \frac{\partial}{\partial \nu} \left(-\frac{(\vec{y} - \vec{1}\mu)^T \vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu)}{2\nu} \right).$$

The first term's derivative is straightforward:

$$\frac{\partial}{\partial \nu} \left(-\frac{n}{2} \ln(\nu) \right) = -\frac{n}{2} \cdot \frac{1}{\nu} = -\frac{n}{2\sigma^2}.$$

71.3. MLE to estimate θ and p

For the second term, let $C_2 = (\vec{y} - \vec{1}\mu)^T \vec{\Psi}^{-1} (\vec{y} - \vec{1}\mu)$. This term is constant with respect to σ^2 . The derivative is:

$$\frac{\partial}{\partial \nu} \left(-\frac{C_2}{2\nu} \right) = -\frac{C_2}{2} \frac{\partial}{\partial \nu} (\nu^{-1}) = -\frac{C_2}{2} (-\nu^{-2}) = \frac{C_2}{2\nu^2} = \frac{(\vec{y} - \vec{1}\mu)^T \vec{\Psi}^{-1} (\vec{y} - \vec{1}\mu)}{2(\sigma^2)^2}.$$

Combining the derivatives, the gradient of the log-likelihood with respect to σ^2 is:

$$\frac{\partial \ln(L)}{\partial \sigma^2} = -\frac{n}{2\sigma^2} + \frac{(\vec{y} - \vec{1}\mu)^T \vec{\Psi}^{-1} (\vec{y} - \vec{1}\mu)}{2(\sigma^2)^2}.$$

Setting this to zero for maximization gives:

$$-\frac{n}{2\sigma^2} + \frac{(\vec{y} - \vec{1}\mu)^T \vec{\Psi}^{-1} (\vec{y} - \vec{1}\mu)}{2(\sigma^2)^2} = 0.$$

71.3.4. Results of the Optimizations

Optimization of the log-likelihood by taking derivatives with respect to μ and σ results in

$$\hat{\mu} = \frac{\vec{1}^T \vec{\Psi}^{-1} \vec{y}^T}{\vec{1}^T \vec{\Psi}^{-1} \vec{1}^T} \quad (71.9)$$

and

$$\hat{\sigma}^2 = \frac{(\vec{y} - \vec{1}\mu)^T \vec{\Psi}^{-1} (\vec{y} - \vec{1}\mu)}{n}. \quad (71.10)$$

71.3.5. The Concentrated Log-Likelihood Function

Combining the equations, i.e., substituting Equation 71.9 and Equation 71.10 into Equation 71.8 leads to the concentrated log-likelihood function:

$$\ln(L) \approx -\frac{n}{2} \ln(\hat{\sigma}) - \frac{1}{2} \ln |\vec{\Psi}|. \quad (71.11)$$

Remark 71.4 (The Concentrated Log-Likelihood).

- The first term in Equation 71.11 requires information about the measured point (observations) y_i .
- To maximize $\ln(L)$, optimal values of $\vec{\theta}$ and $\vec{p}$ are determined numerically, because the function (Equation 71.11) is not differentiable.

□

71.3.6. Optimizing the Parameters $\vec{\theta}$ and $\vec{p}$

The concentrated log-likelihood function is very quick to compute. We do not need a statistical model, because we are only interested in the maximum likelihood estimate (MLE) of θ and p . Optimizers such as Nelder-Mead, Conjugate Gradient, or Simulated Annealing can be used to determine optimal values for θ and p . After the optimization, the correlation matrix Ψ is build with the optimized θ and p values. This is best (most likely) Kriging model for the given data y .

Observing Figure 71.2, there's significant change between $\theta = 0.1$ and $\theta = 1$, just as there is between $\theta = 1$ and $\theta = 10$. Hence, it is sensible to search for θ on a logarithmic scale. Suitable search bounds typically range from 10^{-3} to 10^2 , although this is not a stringent requirement. Importantly, the scaling of the observed data does not affect the values of $\hat{\theta}$, but the scaling of the design space does. Therefore, it is advisable to consistently scale variable ranges between zero and one to ensure consistency in the degree of activity $\hat{\theta}_j$ represents across different problems.

71.3.7. Correlation and Covariance Matrices Revisited

The covariance matrix Σ is constructed by considering both the variance σ^2 and the correlation matrix Ψ . They are related as follows:

1. **Covariance vs. Correlation:** Covariance is a measure of the joint variability of two random variables, while correlation is a standardized measure of this relationship, ranging from -1 to 1. The relationship between covariance and correlation for two random variables X and Y is given by $\text{cor}(X, Y) = \text{cov}(X, Y) / (\sigma_X \sigma_Y)$, where σ_X and σ_Y are their standard deviations.
2. **The Covariance Matrix Σ :** The **covariance matrix** Σ (or $\text{Cov}(Y, Y)$ for the vector $\vec{Y}$) captures the **pairwise covariances** between all elements of the vector of observed responses.
3. **Connecting σ^2 and Ψ to Σ :** In the Kriging framework described, the variance of each observation is often assumed to be constant, σ^2 . The covariance between any two observations $Y(\vec{x}^{(i)})$ and $Y(\vec{x}^{(l)})$ is given by σ^2 multiplied by their correlation. That is,

$$\text{cov}(Y(\vec{x}^{(i)}), Y(\vec{x}^{(l)})) = \sigma^2 \text{cor}(Y(\vec{x}^{(i)}), Y(\vec{x}^{(l)})).$$

This relationship holds for *all* pairs of points. When expressed in matrix form, the covariance matrix Σ is the product of the variance σ^2 (a scalar) and the correlation matrix Ψ :

$$\Sigma = \sigma^2 \Psi.$$

In essence, the correlation matrix Ψ defines the *structure* or *shape* of the dependencies between the data points based on their locations. The parameter σ^2 acts as a **scaling factor** that converts these unitless correlation values (which are between -1 and 1) into

actual covariance values with units of variance, setting the overall level of variability in the system.

So, σ^2 tells us about the general spread or variability of the underlying process, while Ψ tells you *how* that variability is distributed and how strongly points are related to each other based on their positions. Together, they completely define the covariance structure of your observed data in the multivariate normal distribution used in Kriging.

71.4. Implementing an MLE of the Model Parameters

The matrix algebra necessary for calculating the likelihood is the most computationally intensive aspect of the Kriging process. It is crucial to ensure that the code implementation is as efficient as possible.

Given that Ψ (our correlation matrix) is symmetric, only half of the matrix needs to be computed before adding it to its transpose. When calculating the log-likelihood, several matrix inversions are required. The fastest approach is to conduct one Cholesky factorization and then apply backward and forward substitution for each inverse.

The Cholesky factorization is applicable only to positive-definite matrices, which Ψ generally is. However, if Ψ becomes nearly singular, such as when the $\vec{x}^{(i)}$'s are densely packed, the Cholesky factorization might fail. In these cases, one could employ an LU-decomposition, though the result might be unreliable. When Ψ is near singular, the best course of action is to either use regression techniques or, as we do here, assign a poor likelihood value to parameters generating the near singular matrix, thus diverting the MLE search towards better-conditioned Ψ matrices.

When working with correlation matrices, increasing the values on the main diagonal of a matrix will increase the absolute value of its determinant. A critical numerical consideration in calculating the concentrated log-likelihood is that for poorly conditioned matrices, $\det(\Psi)$ approaches zero, leading to potential numerical instability. To address this issue, it is advisable to calculate $\ln(|\Psi|)$ in Equation 71.11 using twice the sum of the logarithms of the diagonal elements of the Cholesky factorization. This approach provides a more numerically stable method for computing the log-determinant, as the Cholesky decomposition $\Psi = LL^T$ allows us to express $\ln(|\Psi|) = 2 \sum_{i=1}^n \ln(L_{ii})$, avoiding the direct computation of potentially very small determinant values.

71.5. Kriging Prediction

We will use the Kriging correlation Ψ to predict new values based on the observed data. The presentation follows the approach described in Forrester, Sóbester, and Keane (2008) and Bartz et al. (2022).

71. Kriging (Gaussian Process Regression)

Main idea for prediction is that the new $Y(\vec{x})$ should be consistent with the old sample data X . For a new prediction $\hat{y}$ at $\vec{x}$, the value of $\hat{y}$ is chosen so that it maximizes the likelihood of the sample data X and the prediction, given the (optimized) correlation parameter $\vec{\theta}$ and $\vec{p}$ from above. The observed data $\vec{y}$ is augmented with the new prediction $\hat{y}$ which results in the augmented vector $\vec{\tilde{y}} = (\vec{y}^T, \hat{y})^T$. A vector of correlations between the observed data and the new prediction is defined as

$$\vec{\psi} = \begin{pmatrix} \text{cor}(Y(\vec{x}^{(1)}), Y(\vec{x})) \\ \vdots \\ \text{cor}(Y(\vec{x}^{(n)}), Y(\vec{x})) \end{pmatrix} = \begin{pmatrix} \psi^{(1)} \\ \vdots \\ \psi^{(n)} \end{pmatrix}.$$

Definition 71.2 (The Augmented Correlation Matrix). The augmented correlation matrix is constructed as

$$\tilde{\Psi} = \begin{pmatrix} \tilde{\Psi} & \vec{\psi} \\ \vec{\psi}^T & 1 \end{pmatrix}.$$

□

The log-likelihood of the augmented data is

$$\ln(L) = -\frac{n}{2} \ln(2\pi) - \frac{n}{2} \ln(\hat{\sigma}^2) - \frac{1}{2} \ln |\tilde{\Psi}| - \frac{(\vec{\tilde{y}} - \vec{1}\hat{\mu})^T \tilde{\Psi}^{-1} (\vec{\tilde{y}} - \vec{1}\hat{\mu})}{2\hat{\sigma}^2}, \quad (71.12)$$

where $\vec{1}$ is a vector of ones and $\hat{\mu}$ and $\hat{\sigma}^2$ are the MLEs from Equation 71.9 and Equation 71.10. Only the last term in Equation 71.12 depends on $\hat{y}$, so we need only consider this term in the maximization. Details can be found in Forrester, Sóbester, and Keane (2008). Finally, the MLE for $\hat{y}$ can be calculated as

$$\hat{y}(\vec{x}) = \hat{\mu} + \vec{\psi}^T \tilde{\Psi}^{-1} (\vec{\tilde{y}} - \vec{1}\hat{\mu}). \quad (71.13)$$

Equation 71.13 reveals two important properties of the Kriging predictor:

- **Basis functions:** The basis function impacts the vector $\vec{\psi}$, which contains the n correlations between the new point $\vec{x}$ and the observed locations. Values from the n basis functions are added to a mean base term μ with weightings

$$\vec{w} = \tilde{\Psi}^{(-1)} (\vec{\tilde{y}} - \vec{1}\hat{\mu}).$$

- **Interpolation:** The predictions interpolate the sample data. When calculating the prediction at the i th sample point, $\vec{x}^{(i)}$, the i th column of $\tilde{\Psi}^{-1}$ is $\vec{\psi}$, and $\vec{\psi} \tilde{\Psi}^{-1}$ is the i th unit vector. Hence,

$$\hat{y}(\vec{x}^{(i)}) = y^{(i)}.$$

71.6. Kriging Example: Sinusoid Function

Toy example in 1d where the response is a simple sinusoid measured at eight equally spaced x -locations in the span of a single period of oscillation.

71.6.1. Calculating the Correlation Matrix Ψ

The correlation matrix Ψ is based on the pairwise squared distances between the input locations. Here we will use $n = 8$ sample locations and θ is set to 1.0.

```
n = 8
X = np.linspace(0, 2*np.pi, n, endpoint=False).reshape(-1,1)
print(np.round(X, 2))
```

```
[[0.  ]
 [0.79]
 [1.57]
 [2.36]
 [3.14]
 [3.93]
 [4.71]
 [5.5 ]]
```

Evaluate at sample points

```
y = np.sin(X)
print(np.round(y, 2))
```

```
[[ 0.  ]
 [ 0.71]
 [ 1.  ]
 [ 0.71]
 [ 0.  ]
 [-0.71]
 [-1.  ]
 [-0.71]]
```

We have the data points shown in Table 71.1.

71. Kriging (Gaussian Process Regression)

Table 71.1.: Data points for the sinusoid function

x	y
0.0	0.0
0.79	0.71
1.57	1.0
2.36	0.71
3.14	0.0
3.93	-0.71
4.71	-1.0
5.5	-0.71

The data points are visualized in Figure 71.4.

```
plt.plot(X, y, "bo")
plt.title(f"Sin(x) evaluated at {n} points")
plt.grid()
plt.show()
```

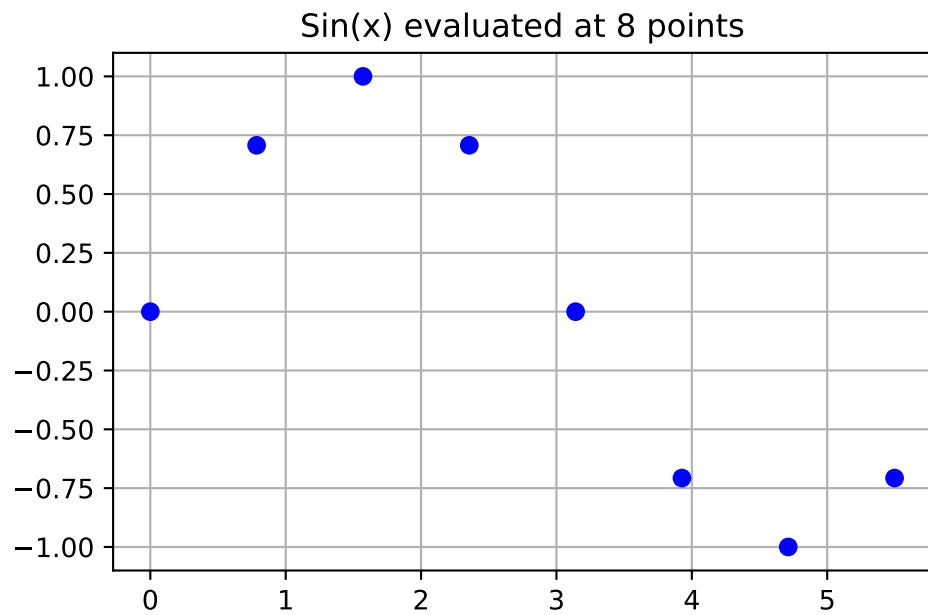


Figure 71.4.: Sin(x) evaluated at 8 points.

71.6.2. Computing the Ψ Matrix

We will use the `build_Psi` function from Example 71.4 to compute the correlation matrix Ψ . θ should be an array of one value, because we are only working in one dimension ($k = 1$).

```
theta = np.array([1.0])
Psi = build_Psi(X, theta)
print(np.round(Psi, 2))
```

```
[[1.  0.54 0.08 0.  0.  0.  0.  0. ]
 [0.54 1.  0.54 0.08 0.  0.  0.  0. ]
 [0.08 0.54 1.  0.54 0.08 0.  0.  0. ]
 [0.  0.08 0.54 1.  0.54 0.08 0.  0. ]
 [0.  0.  0.08 0.54 1.  0.54 0.08 0. ]
 [0.  0.  0.  0.08 0.54 1.  0.54 0.08]
 [0.  0.  0.  0.  0.08 0.54 1.  0.54]
 [0.  0.  0.  0.  0.  0.08 0.54 1. ]]
```

Figure 71.5 visualizes the $(8, 8)$ correlation matrix Ψ .

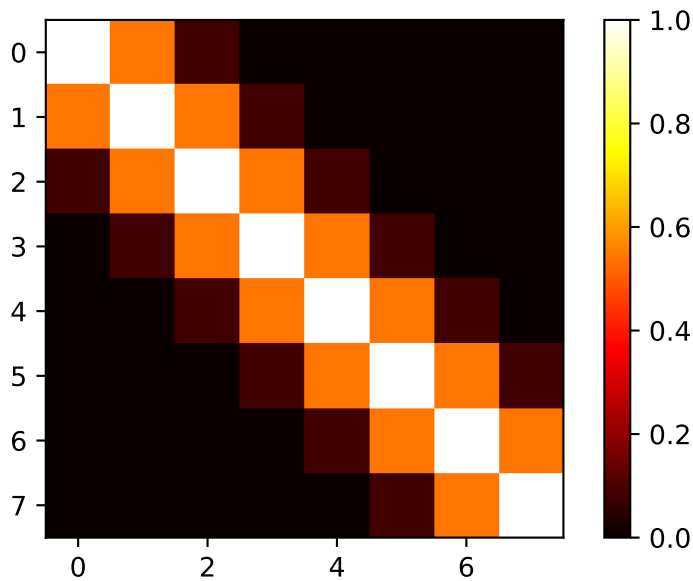


Figure 71.5.: Correlation matrix Ψ for the sinusoid function.

71. Kriging (Gaussian Process Regression)

71.6.3. Selecting the New Locations

We would like to predict at $m = 100$ new locations (or testign locations) in the interval $[0, 2\pi]$. The new locations are stored in the variable $\mathbf{x}$.

```
m = 100
x = np.linspace(0, 2*np.pi, m, endpoint=False).reshape(-1,1)
```

71.6.4. Computing the ψ Vector

Distances between testing locations x and training data locations X .

```
def build_psi(X, x, theta, eps=sqrt(spacing(1))):
    n = X.shape[0]
    k = X.shape[1]
    m = x.shape[0]
    psi = zeros((n, m))
    theta = theta * ones(k)
    D = zeros((n, m))
    D = cdist(x.reshape(-1, k),
              X.reshape(-1, k),
              metric='sqeuclidean',
              out=None,
              w=theta)
    psi = exp(-D)
    # return psi transpose to be consistent with the literature
    print(f"Dimensions of psi: {psi.T.shape}")
    return(psi.T)

psi = build_psi(X, x, theta)
```

Dimensions of psi: (8, 100)

Figure 71.6 visualizes the (8,100) prediction matrix ψ .

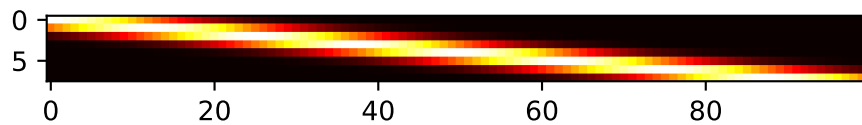


Figure 71.6.: Visualization of the predition matrix ψ

71.6.5. Predicting at New Locations

Computation of the predictive equations.

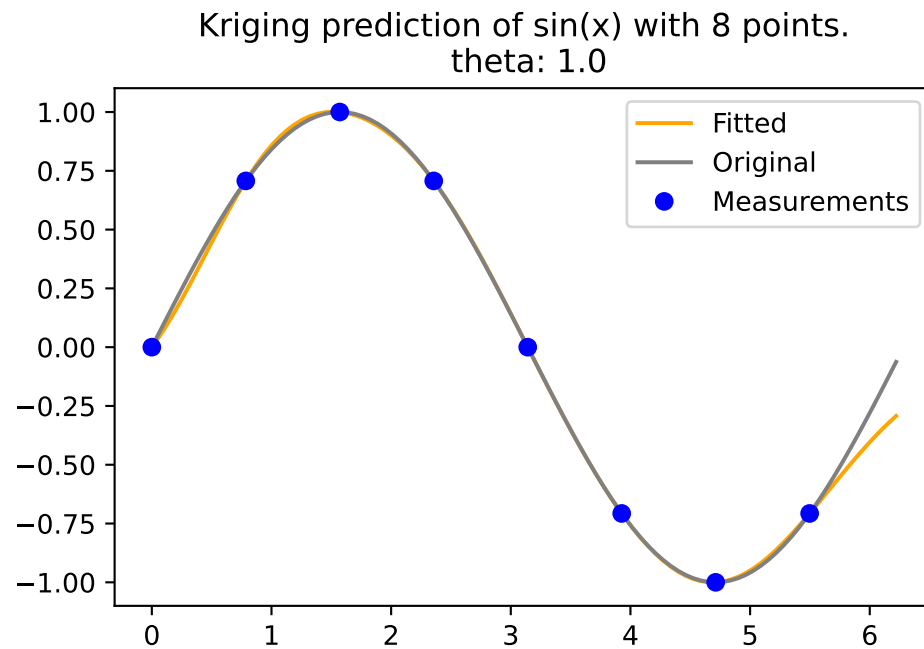
```
U = cholesky(Psi).T
one = np.ones(n).reshape(-1,1)
mu = (one.T.dot(solve(U, solve(U.T, y)))) / one.T.dot(solve(U, solve(U.T, one)))
f = mu * ones(m).reshape(-1,1) + psi.T.dot(solve(U, solve(U.T, y - one * mu)))
print(f"Dimensions of f: {f.shape}")
```

Dimensions of f: (100, 1)

To compute f , Equation 71.13 is used.

71.6.6. Visualization

```
plt.plot(x, f, color = "orange", label="Fitted")
plt.plot(x, np.sin(x), color = "grey", label="Original")
plt.plot(X, y, "bo", label="Measurements")
plt.title("Kriging prediction of sin(x) with {} points.\n theta: {}".format(n, theta[0]))
plt.legend(loc='upper right')
plt.show()
```



71.6.7. The Complete Python Code for the Example

Here is the self-contained Python code for direct use in a notebook:

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import (array, zeros, power, ones, exp, multiply, eye, linspace, spacing, s
from numpy.linalg import cholesky, solve
from scipy.spatial.distance import squareform, pdist, cdist

# --- 1. Kriging Basis Functions (Defining the Correlation) ---
# The core of Kriging uses a specialized basis function for correlation:
#  $\psi(x^{(i)}, x) = \exp(-\sum_{j=1}^k \theta_{\theta_j} |x_j^{(i)} - x_j|^{p_j})$ 
# For this 1D example ( $k=1$ ), and with  $p_j=2$  (squared Euclidean distance implicit from
# and  $\theta_{\theta_j} = \theta$  (a single value), it simplifies.

def build_Psi(X, theta, eps=sqrt(spacing(1))):
    """
    Computes the correlation matrix Psi based on pairwise squared Euclidean distances
    between input locations, scaled by theta.
    Adds a small epsilon to the diagonal for numerical stability (nugget effect).
```

71.6. Kriging Example: Sinusoid Function

```
"""
# Calculate pairwise squared Euclidean distances (D) between points in X
D = squareform(pdist(X, metric='sqeuclidean', out=None, w=theta))
# Compute Psi = exp(-D)
Psi = exp(-D)
# Add a small value to the diagonal for numerical stability (nugget)
# This is often done in Kriging implementations, though a regression method
# with a 'nugget' parameter (Lambda) is explicitly mentioned for noisy data later.
# The source code snippet for build_Psi explicitly includes `multiply(eye(X.shape), eps)`.
# FIX: Use X.shape to get the number of rows for the identity matrix
Psi += multiply(eye(X.shape[0]), eps) # Corrected line
return Psi

def build_psi(X_train, x_predict, theta):
    """
    Computes the correlation vector (or matrix) psi between new prediction locations
    and training data locations.
    """
    # Calculate pairwise squared Euclidean distances (D) between prediction points (x_predict)
    # and training points (X_train).
    # `cdist` computes distances between each pair of the two collections of inputs.
    D = cdist(x_predict, X_train, metric='sqeuclidean', out=None, w=theta)
    # Compute psi = exp(-D)
    psi = exp(-D)
    return psi.T # Return transpose to be consistent with literature (n x m or n x 1)

# --- 2. Data Points for the Sinusoid Function Example ---
# The example uses a 1D sinusoid measured at eight equally spaced x-locations [153, Table 9.1].
n = 8 # Number of sample locations
X_train = np.linspace(0, 2 * np.pi, n, endpoint=False).reshape(-1, 1) # Generate x-locations
y_train = np.sin(X_train) # Corresponding y-values (sine of x)

print("--- Training Data (X_train, y_train) ---")
print("x values:\n", np.round(X_train, 2))
print("y values:\n", np.round(y_train, 2))
print("-" * 40)

# Visualize the data points
plt.figure(figsize=(8, 5))
plt.plot(X_train, y_train, "bo", label=f"Measurements ({n} points)")
plt.title(f"Sin(x) evaluated at {n} points")
plt.xlabel("x")
plt.ylabel("sin(x)")
plt.grid(True)
```

71. Kriging (Gaussian Process Regression)

```
plt.legend()
plt.show()

# --- 3. Calculating the Correlation Matrix (Psi) ---
# Psi is based on pairwise squared distances between input locations.
# theta is set to 1.0 for this 1D example.
theta = np.array([1.0])
Psi = build_Psi(X_train, theta)

print("\n--- Computed Correlation Matrix (Psi) ---")
print("Dimensions of Psi:", Psi.shape) # Should be (8, 8)
print("First 5x5 block of Psi:\n", np.round(Psi[:5,:5], 2))
print("-" * 40)

# --- 4. Selecting New Locations (for Prediction) ---
# We want to predict at m = 100 new locations in the interval [0, 2*pi].
m = 100 # Number of new locations
x_predict = np.linspace(0, 2 * np.pi, m, endpoint=True).reshape(-1, 1)

print("\n--- New Locations for Prediction (x_predict) ---")
print(f"Number of prediction points: {m}")
print("First 5 prediction points:\n", np.round(x_predict[:5], 2).flatten())
print("-" * 40)

# --- 5. Computing the psi Vector ---
# This vector contains correlations between each of the n observed data points
# and each of the m new prediction locations.
psi = build_psi(X_train, x_predict, theta)

print("\n--- Computed Prediction Correlation Matrix (psi) ---")
print("Dimensions of psi:", psi.shape) # Should be (8, 100)
print("First 5x5 block of psi:\n", np.round(psi[:5,:5], 2))
print("-" * 40)

# --- 6. Predicting at New Locations (Kriging Prediction) ---
# The Maximum Likelihood Estimate (MLE) for y_hat is calculated using the formula:
# y_hat(x) = mu_hat + psi.T @ Psi_inv @ (y - 1 * mu_hat) [p. 2 of previous response, a
# Matrix inversion is efficiently performed using Cholesky factorization.

# Step 6a: Cholesky decomposition of Psi
U = cholesky(Psi).T # Note: `cholesky` in numpy returns lower triangular L, we need U

# Step 6b: Calculate mu_hat (estimated mean)
# mu_hat = (one.T @ Psi_inv @ y) / (one.T @ Psi_inv @ one) [p. 2 of previous response]
```


71.6. Kriging Example: Sinusoid Function

```
one = np.ones(n).reshape(-1, 1) # Vector of ones
mu_hat = (one.T @ solve(U, solve(U.T, y_train))) / (one.T @ solve(U, solve(U.T, one)))
mu_hat = mu_hat.item() # Extract scalar value

print("\n--- Kriging Prediction Calculation ---")
print(f"Estimated mean (mu_hat): {np.round(mu_hat, 4)}")

# Step 6c: Calculate predictions f (y_hat) at new locations
# f = mu_hat * ones(m) + psi.T @ Psi_inv @ (y - one * mu_hat)
f_predict = mu_hat * np.ones(m).reshape(-1, 1) + psi.T @ solve(U, solve(U.T, y_train - one * mu_hat))

print(f"Dimensions of predicted values (f_predict): {f_predict.shape}") # Should be (100, 1)
print("First 5 predicted f values:\n", np.round(f_predict[:5], 2).flatten())
print("-" * 40)

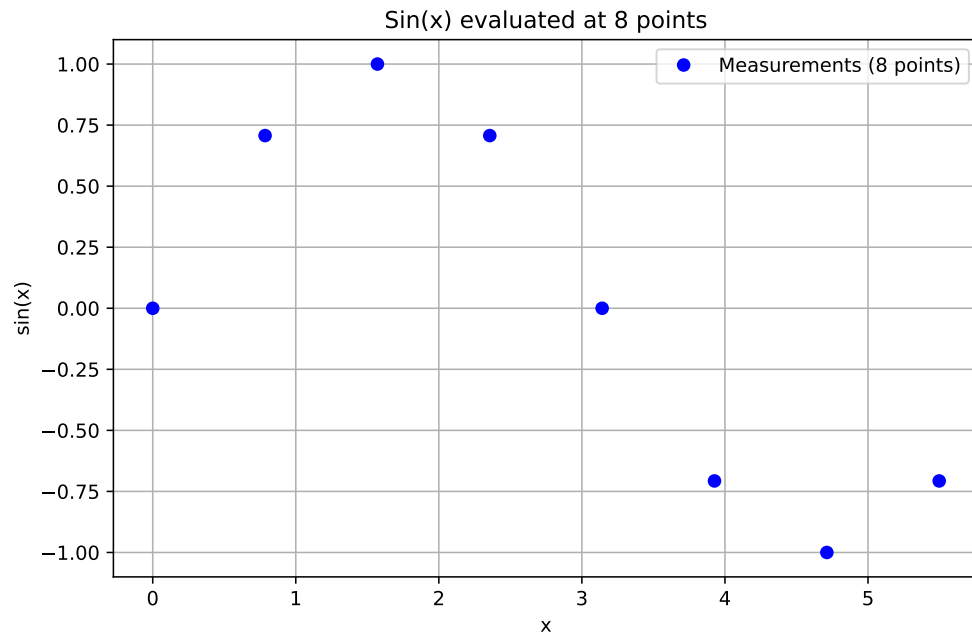
# --- 7. Visualization ---
# Plot the original sinusoid function, the measured points, and the Kriging predictions.

plt.figure(figsize=(10, 6))
plt.plot(x_predict, f_predict, color="orange", label="Kriging Prediction")
plt.plot(x_predict, np.sin(x_predict), color="grey", linestyle='--', label="True Sinusoid Function")
plt.plot(X_train, y_train, "bo", markersize=8, label="Measurements")
plt.title(f"Kriging prediction of sin(x) with {n} points. (theta: {theta})")
plt.xlabel("x")
plt.ylabel("y")
plt.legend(loc='upper right')
plt.grid(True)
plt.show()

--- Training Data (X_train, y_train) ---
x values:
[[0. ]
 [0.79]
 [1.57]
 [2.36]
 [3.14]
 [3.93]
 [4.71]
 [5.5 ]]
y values:
[[ 0. ]
 [ 0.71]
 [ 1. ]
 [ 0.71]
```

71. Kriging (Gaussian Process Regression)

```
[ 0. ]
[-0.71]
[-1. ]
[-0.71]]
```



--- Computed Correlation Matrix (Psi) ---

Dimensions of Psi: (8, 8)

First 5x5 block of Psi:

```
[[1.  0.54 0.08 0.  0. ]
 [0.54 1.  0.54 0.08 0. ]
 [0.08 0.54 1.  0.54 0.08]
 [0.  0.08 0.54 1.  0.54]
 [0.  0.  0.08 0.54 1.  ]]
```

--- New Locations for Prediction (x_predict) ---

Number of prediction points: 100

First 5 prediction points:

```
[0.  0.06 0.13 0.19 0.25]
```

71.6. Kriging Example: Sinusoid Function

--- Computed Prediction Correlation Matrix (psi) ---

Dimensions of psi: (8, 100)

First 5x5 block of psi:

```
[[1.    1.    0.98 0.96 0.94]
 [0.54 0.59 0.65 0.7  0.75]
 [0.08 0.1  0.12 0.15 0.18]
 [0.    0.01 0.01 0.01 0.01]
 [0.    0.    0.    0.    0.  ]]
```

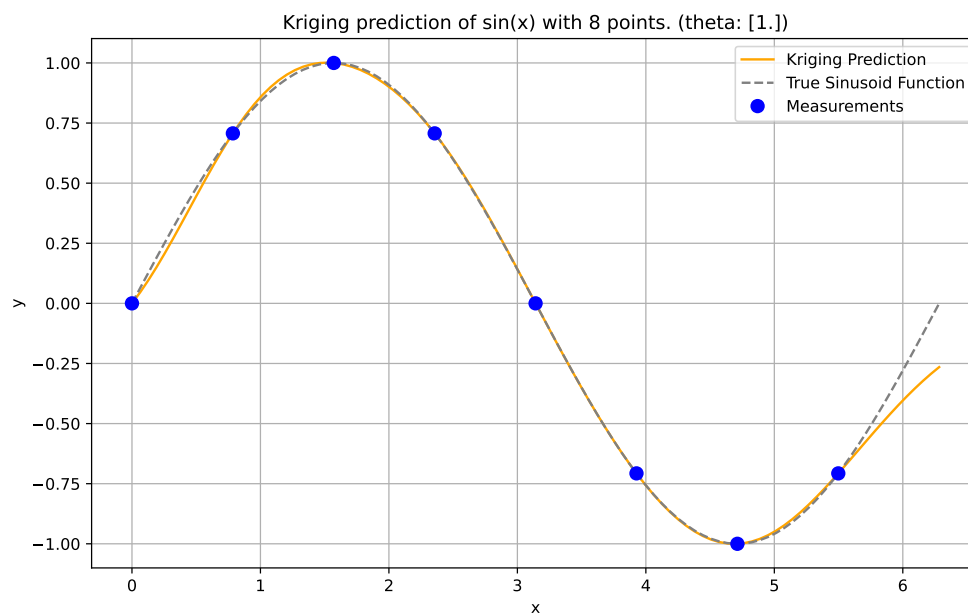
--- Kriging Prediction Calculation ---

Estimated mean (μ_{hat}): -0.0499

Dimensions of predicted values (f_{predict}): (100, 1)

First 5 predicted f values:

```
[0.    0.05 0.1  0.15 0.21]
```



71.7. Jupyter Notebook

Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

72. Matrices

72.1. Derivatives of Quadratic Forms

We present a step-by-step derivation of the general formula

$$\frac{\partial}{\partial \mathbf{v}}(\mathbf{v}^T \mathbf{A} \mathbf{v}) = \mathbf{A} \mathbf{v} + \mathbf{A}^T \mathbf{v}. \quad (72.1)$$

1. Define the components. Let $\mathbf{v}$ be a vector of size $n \times 1$, and let $\mathbf{A}$ be a matrix of size $n \times n$.
2. Write out the quadratic form in summation notation. The product $\mathbf{v}^T \mathbf{A} \mathbf{v}$ is a scalar. It can be expanded and be rewritten as a double summation:

$$\mathbf{v}^T \mathbf{A} \mathbf{v} = \sum_{i=1}^n \sum_{j=1}^n v_i a_{ij} v_j.$$

3. Calculate the partial derivative with respect to a component v_k : The derivative of the scalar $\mathbf{v}^T \mathbf{A} \mathbf{v}$ with respect to the vector $\mathbf{v}$ is the gradient vector, whose k -th component is $\frac{\partial}{\partial v_k}(\mathbf{v}^T \mathbf{A} \mathbf{v})$. We need to find $\frac{\partial}{\partial v_k} \left(\sum_{i=1}^n \sum_{j=1}^n v_i a_{ij} v_j \right)$. Consider the terms in the summation that involve v_k . A term $v_i a_{ij} v_j$ involves v_k if $i = k$ or $j = k$ (or both).
 - Terms where $i = k$: $v_k a_{kj} v_j$. The derivative with respect to v_k is $a_{kj} v_j$.
 - Terms where $j = k$: $v_i a_{ik} v_k$. The derivative with respect to v_k is $v_i a_{ik}$.
 - The term where $i = k$ and $j = k$: $v_k a_{kk} v_k = a_{kk} v_k^2$. Its derivative with respect to v_k is $2a_{kk} v_k$. Notice this term is included in both cases above when $i = k$ and $j = k$. When $i = k$, the term is $v_k a_{kk} v_k$, derivative is $a_{kk} v_k$. When $j = k$, the term is $v_k a_{kk} v_k$, derivative is $v_k a_{kk}$. Summing these two gives $2a_{kk} v_k$.

4. Let's differentiate the sum $\sum_{i=1}^n \sum_{j=1}^n v_i a_{ij} v_j$ with respect to v_k :

$$\frac{\partial}{\partial v_k} \left(\sum_{i=1}^n \sum_{j=1}^n v_i a_{ij} v_j \right) = \sum_{i=1}^n \sum_{j=1}^n \frac{\partial}{\partial v_k} (v_i a_{ij} v_j).$$

5. The partial derivative $\frac{\partial}{\partial v_k} (v_i a_{ij} v_j)$ is non-zero only if $i = k$ or $j = k$.

- If $i = k$ and $j \neq k$: $\frac{\partial}{\partial v_k}(v_k a_{kj} v_j) = a_{kj} v_j$.
 - If $i \neq k$ and $j = k$: $\frac{\partial}{\partial v_k}(v_i a_{ik} v_k) = v_i a_{ik}$.
 - If $i = k$ and $j = k$: $\frac{\partial}{\partial v_k}(v_k a_{kk} v_k) = \frac{\partial}{\partial v_k}(a_{kk} v_k^2) = 2a_{kk} v_k$.
6. So, the partial derivative is the sum of derivatives of all terms involving v_k :
 $\frac{\partial}{\partial v_k}(\mathbf{v}^T \mathbf{A} \mathbf{v}) = \sum_{j \neq k} (a_{kj} v_j) + \sum_{i \neq k} (v_i a_{ik}) + (2a_{kk} v_k)$.
7. We can rewrite this by including the $i = k, j = k$ term back into the summations:
 $\sum_{j \neq k} (a_{kj} v_j) + a_{kk} v_k + \sum_{i \neq k} (v_i a_{ik}) + v_k a_{kk}$ (since $v_k a_{kk} = a_{kk} v_k$) $= \sum_{j=1}^n a_{kj} v_j + \sum_{i=1}^n v_i a_{ik}$.
8. Convert back to matrix/vector notation: The first summation $\sum_{j=1}^n a_{kj} v_j$ is the k -th component of the matrix-vector product $\mathbf{A} \mathbf{v}$. The second summation $\sum_{i=1}^n v_i a_{ik}$ can be written as $\sum_{i=1}^n a_{ik} v_i$. Recall that the element in the k -th row and i -th column of the transpose matrix $\mathbf{A}^T$ is $(A^T)_{ki} = a_{ik}$. So, $\sum_{i=1}^n a_{ik} v_i = \sum_{i=1}^n (A^T)_{ki} v_i$, which is the k -th component of the matrix-vector product $\mathbf{A}^T \mathbf{v}$.
9. Assemble the gradient vector: The k -th component of the gradient $\frac{\partial}{\partial \mathbf{v}}(\mathbf{v}^T \mathbf{A} \mathbf{v})$ is $(\mathbf{A} \mathbf{v})_k + (\mathbf{A}^T \mathbf{v})_k$. Since this holds for all $k = 1, \dots, n$, the gradient vector is the sum of the two vectors $\mathbf{A} \mathbf{v}$ and $\mathbf{A}^T \mathbf{v}$. Therefore, the general formula for the derivative is $\frac{\partial}{\partial \mathbf{v}}(\mathbf{v}^T \mathbf{A} \mathbf{v}) = \mathbf{A} \mathbf{v} + \mathbf{A}^T \mathbf{v}$.

72.2. The Condition Number

A small value, `eps`, can be passed to the function `build_Psi` to improve the condition number. For example, `eps=sqrt(spacing(1))` can be used. The numpy function `spacing()` returns the distance between a number and its nearest adjacent number.

The condition number of a matrix is a measure of its sensitivity to small changes in its elements. It is used to estimate how much the output of a function will change if the input is slightly altered.

A matrix with a low condition number is well-conditioned, which means its behavior is relatively stable, while a matrix with a high condition number is ill-conditioned, meaning its behavior is unstable with respect to numerical precision.

```
import numpy as np

# Define a well-conditioned matrix (low condition number)
A = np.array([[1, 0.1], [0.1, 1]])
print("Condition number of A: ", np.linalg.cond(A))

# Define an ill-conditioned matrix (high condition number)
```

```
B = np.array([[1, 0.99999999], [0.99999999, 1]])
print("Condition number of B: ", np.linalg.cond(B))
```

```
Condition number of A: 1.2222222222222225
Condition number of B: 200000000.57495335
```

72.3. The Moore-Penrose Pseudoinverse

72.3.1. Definitions

The Moore-Penrose pseudoinverse is a generalization of the inverse matrix for non-square or singular matrices. It is computed as

$$A^+ = (A^*A)^{-1}A^*,$$

where A^* is the conjugate transpose of A .

It satisfies the following properties:

1. $AA^+A = A$
2. $A^+AA^+ = A^+$
3. $(AA^+)^* = AA^+$.
4. $(A^+A)^* = A^+A$
5. $A^+ = (A^*)^+$
6. $A^+ = A^T$ if A is a square matrix and A is invertible.

The pseudoinverse can be computed using Singular Value Decomposition (SVD).

72.3.2. Implementation in Python

```
import numpy as np
from numpy.linalg import pinv
A = np.array([[1, 2], [3, 4], [5, 6]])
print(f"Matrix A:\n {A}")
A_pseudo_inv = pinv(A)
print(f"Moore-Penrose Pseudoinverse:\n {A_pseudo_inv}")
```

Matrix A:

```
[[1 2]
 [3 4]
 [5 6]]
```

Moore-Penrose Pseudoinverse:

```
[[ -1.33333333  -0.33333333   0.66666667]
 [  1.08333333   0.33333333  -0.41666667]]
```

72.4. Strictly Positive Definite Kernels

72.4.1. Definition

Definition 72.1 (Strictly Positive Definite Kernel). A kernel function $k(x, y)$ is called strictly positive definite if for any finite collection of distinct points $x_1, x_2, \dots, x_n$ in the input space and any non-zero vector of coefficients $\alpha = (\alpha_1, \alpha_2, \dots, \alpha_n)$, the following inequality holds:

$$\sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j k(x_i, x_j) > 0. \quad (72.2)$$

In contrast, a kernel function $k(x, y)$ is called positive definite (but not strictly) if the “>” sign is replaced by “ $\geq$ ” in the above inequality.

72.4.2. Connection to Positive Definite Matrices

The connection between strictly positive definite kernels and positive definite matrices lies in the Gram matrix construction:

- When we evaluate a kernel function $k(x, y)$ at all pairs of data points in our sample, we construct the Gram matrix K where $K_{ij} = k(x_i, x_j)$.
- If the kernel function k is strictly positive definite, then for any set of distinct points, the resulting Gram matrix will be symmetric positive definite.

A symmetric matrix is positive definite if and only if for any non-zero vector α , the quadratic form $\alpha^T K \alpha > 0$, which directly corresponds to the kernel definition above.

72.4.3. Connection to RBF Models

For RBF models, the kernel function is the radial basis function itself:

$$k(x, y) = \psi(\|x - y\|).$$

The Gaussian RBF kernel $\psi(r) = e^{-r^2/(2\sigma^2)}$ is strictly positive definite in $\mathbb{R}^n$ for any dimension n . The inverse multiquadric kernel $\psi(r) = (r^2 + \sigma^2)^{-1/2}$ is also strictly positive definite in any dimension.

This mathematical property guarantees that the interpolation problem has a unique solution (the weight vector $\vec{w}$ is uniquely determined). The linear system $\Psi\vec{w} = \vec{y}$ can be solved reliably using Cholesky decomposition. The RBF interpolant exists and is unique for any distinct set of centers.

72.5. Cholesky Decomposition and Positive Definite Matrices

We consider the definiteness of a matrix, before discussing the Cholesky decomposition.

Definition 72.2 (Positive Definite Matrix). A symmetric matrix A is positive definite if all its eigenvalues are positive.

Example 72.1 (Positive Definite Matrix). Given a symmetric matrix $A = \begin{pmatrix} 9 & 4 \\ 4 & 9 \end{pmatrix}$, the eigenvalues of A are $\lambda_1 = 13$ and $\lambda_2 = 5$. Since both eigenvalues are positive, the matrix A is positive definite.

Definition 72.3 (Negative Definite, Positive Semidefinite, and Negative Semidefinite Matrices). Similarly, a symmetric matrix A is negative definite if all its eigenvalues are negative. It is positive semidefinite if all its eigenvalues are non-negative, and negative semidefinite if all its eigenvalues are non-positive.

The covariance matrix must be positive definite for a multivariate normal distribution for a couple of reasons:

- Semidefinite vs Definite: A covariance matrix is always symmetric and positive semidefinite. However, for a multivariate normal distribution, it must be positive definite, not just semidefinite. This is because a positive semidefinite matrix can have zero eigenvalues, which would imply that some dimensions in the distribution have zero variance, collapsing the distribution in those dimensions. A positive definite matrix has all positive eigenvalues, ensuring that the distribution has positive variance in all dimensions.

- Invertibility: The multivariate normal distribution's probability density function involves the inverse of the covariance matrix. If the covariance matrix is not positive definite, it may not be invertible, and the density function would be undefined.

In summary, the covariance matrix being positive definite ensures that the multivariate normal distribution is well-defined and has positive variance in all dimensions.

The definiteness of a matrix can be checked by examining the eigenvalues of the matrix. If all eigenvalues are positive, the matrix is positive definite.

```
import numpy as np

def is_positive_definite(matrix):
    return np.all(np.linalg.eigvals(matrix) > 0)

matrix = np.array([[9, 4], [4, 9]])
print(is_positive_definite(matrix)) # Outputs: True
```

True

However, a more efficient way to check the definiteness of a matrix is through the Cholesky decomposition.

Definition 72.4 (Cholesky Decomposition). For a given symmetric positive-definite matrix $A \in \mathbb{R}^{n \times n}$, there exists a unique lower triangular matrix $L \in \mathbb{R}^{n \times n}$ with positive diagonal elements such that:

$$A = LL^T.$$

Here, L^T denotes the transpose of L .

Example 72.2 (Cholesky decomposition using `numpy`). `linalg.cholesky` computes the Cholesky decomposition of a matrix, i.e., it computes a lower triangular matrix L such that $LL^T = A$. If the matrix is not positive definite, an error (`LinAlgError`) is raised.

```
import numpy as np

# Define a Hermitian, positive-definite matrix
A = np.array([[9, 4], [4, 9]])

# Compute the Cholesky decomposition
L = np.linalg.cholesky(A)
```

```
print("L = \n", L)
print("L*LT = \n", np.dot(L, L.T))
```

```
L =
[[3.          0.          ]
 [1.33333333  2.68741925]]
L*LT =
[[9.  4.]
 [4.  9.]]
```

Example 72.3 (Cholesky Decomposition). Given a symmetric positive-definite matrix $A = \begin{pmatrix} 9 & 4 \\ 4 & 9 \end{pmatrix}$, the Cholesky decomposition computes the lower triangular matrix L such that $A = LL^T$. The matrix L is computed as:

$$L = \begin{pmatrix} 3 & 0 \\ 4/3 & 2 \end{pmatrix},$$

so that

$$LL^T = \begin{pmatrix} 3 & 0 \\ 4/3 & \sqrt{65}/3 \end{pmatrix} \begin{pmatrix} 3 & 4/3 \\ 0 & \sqrt{65}/3 \end{pmatrix} = \begin{pmatrix} 9 & 4 \\ 4 & 9 \end{pmatrix} = A.$$

An efficient implementation of the definiteness-check based on Cholesky is already available in the `numpy` library. It provides the `np.linalg.cholesky` function to compute the Cholesky decomposition of a matrix. This more efficient `numpy`-approach can be used as follows:

```
import numpy as np

def is_pd(K):
    try:
        np.linalg.cholesky(K)
        return True
    except np.linalg.linalg.LinAlgError as err:
        if 'Matrix is not positive definite' in err.message:
            return False
        else:
            raise
matrix = np.array([[9, 4], [4, 9]])
print(is_pd(matrix)) # Outputs: True
```

True

72.5.1. Example of Cholesky Decomposition

We consider dimension $k = 1$ and $n = 2$ sample points. The sample points are located at $x_1 = 1$ and $x_2 = 5$. The response values are $y_1 = 2$ and $y_2 = 10$. The correlation parameter is $\theta = 1$ and p is set to 1. Using Equation 71.1, we can compute the correlation matrix Ψ :

$$\Psi = \begin{pmatrix} 1 & e^{-1} \\ e^{-1} & 1 \end{pmatrix}.$$

To determine MLE as in Equation 71.13, we need to compute Ψ^{-1} :

$$\Psi^{-1} = \frac{e}{e^2 - 1} \begin{pmatrix} e & -1 \\ -1 & e \end{pmatrix}.$$

Cholesky-decomposition of Ψ is recommended to compute Ψ^{-1} . Cholesky decomposition is a decomposition of a positive definite symmetric matrix into the product of a lower triangular matrix L , a diagonal matrix D and the transpose of L , which is denoted as L^T . Consider the following example:

$$\begin{aligned} LDL^T &= \begin{pmatrix} 1 & 0 \\ l_{21} & 1 \end{pmatrix} \begin{pmatrix} d_{11} & 0 \\ 0 & d_{22} \end{pmatrix} \begin{pmatrix} 1 & l_{21} \\ 0 & 1 \end{pmatrix} = \\ &= \begin{pmatrix} d_{11} & 0 \\ d_{11}l_{21} & d_{22} \end{pmatrix} \begin{pmatrix} 1 & l_{21} \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} d_{11} & d_{11}l_{21} \\ d_{11}l_{21} & d_{11}l_{21}^2 + d_{22} \end{pmatrix}. \end{aligned} \quad (72.3)$$

Using Equation 72.3, we can compute the Cholesky decomposition of Ψ :

1. $d_{11} = 1$,
2. $l_{21}d_{11} = e^{-1} \Rightarrow l_{21} = e^{-1}$, and
3. $d_{11}l_{21}^2 + d_{22} = 1 \Rightarrow d_{22} = 1 - e^{-2}$.

The Cholesky decomposition of Ψ is

$$\Psi = \begin{pmatrix} 1 & 0 \\ e^{-1} & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 - e^{-2} \end{pmatrix} \begin{pmatrix} 1 & e^{-1} \\ 0 & 1 \end{pmatrix} = LDL^T$$

Some programs use U instead of L . The Cholesky decomposition of Ψ is

$$\Psi = LDL^T = U^T D U.$$

Using

$$\sqrt{D} = \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1 - e^{-2}} \end{pmatrix},$$

we can write the Cholesky decomposition of Ψ without a diagonal matrix D as

$$\Psi = \begin{pmatrix} 1 & 0 \\ e^{-1} & \sqrt{1-e^{-2}} \end{pmatrix} \begin{pmatrix} 1 & e^{-1} \\ 0 & \sqrt{1-e^{-2}} \end{pmatrix} = U^T U.$$

72.5.2. Inverse Matrix Using Cholesky Decomposition

To compute the inverse of a matrix using the Cholesky decomposition, you can follow these steps:

1. Decompose the matrix A into L and L^T , where L is a lower triangular matrix and L^T is the transpose of L .
2. Compute L^{-1} , the inverse of L .
3. The inverse of A is then $(L^{-1})^T L^{-1}$.

Please note that this method only applies to symmetric, positive-definite matrices.

The inverse of the matrix Ψ from above is:

$$\Psi^{-1} = \frac{e}{e^2 - 1} \begin{pmatrix} e & -1 \\ -1 & e \end{pmatrix}.$$

Here's an example of how to compute the inverse of a matrix using Cholesky decomposition in Python:

```
import numpy as np
from scipy.linalg import cholesky, inv
E = np.exp(1)

# Psi is a symmetric, positive-definite matrix
Psi = np.array([[1, 1/E], [1/E, 1]])
L = cholesky(Psi, lower=True)
L_inv = inv(L)
# The inverse of A is (L^-1)^T * L^-1
Psi_inv = np.dot(L_inv.T, L_inv)

print("Psi:\n", Psi)
print("Psi Inverse:\n", Psi_inv)
```

```
Psi:
[[1.          0.36787944]
 [0.36787944  1.          ]]
Psi Inverse:
[[ 1.15651764 -0.42545906]
 [-0.42545906  1.15651764]]
```

72.6. Nyström Approximation

72.6.1. What's the Big Idea?

Imagine you have a huge, detailed map of a country. Working with the full, high-resolution map is slow and takes up a lot of computer memory. The Nyström method is like creating a smaller-scale summary map by only looking at a few key, representative locations.

In machine learning, we often work with a **kernel matrix** (or Gram matrix), which tells us how similar every pair of data points is to each other. For very large datasets, this matrix can become massive, making it computationally expensive to store and process.

The Nyström method provides an efficient way to create a **low-rank approximation** of this large kernel matrix. In simple terms, it finds a “simpler” version of the matrix that captures its most important properties without needing to compute or store the whole thing.

72.6.2. How Does It Work?

The core idea is to select a small, random subset of the columns of the full kernel matrix and use them to reconstruct the entire matrix. Let's say our full kernel matrix is K .

1. **Sample:** Randomly select l columns from the n total columns of K . Let C be the $n \times l$ matrix of these sampled columns.
2. **Intersect:** Take the rows of C corresponding to the sampled column indices to form the $l \times l$ matrix W .
3. **Approximate:** Using C and W , calculate the Nyström approximation $\tilde{K}$ of K :

$$\tilde{K} \approx CW^+C^T$$

where W^+ is the pseudoinverse of W .

72.6.3. Example

Suppose we have 4 data points and the full kernel matrix K is:

$$K = \begin{pmatrix} 9 & 6 & 3 & 1 \\ 6 & 4 & 2 & 0.5 \\ 3 & 2 & 1 & 0.25 \\ 1 & 0.5 & 0.25 & 0.1 \end{pmatrix}$$

Let's approximate it by sampling 2 columns ($l = 2$):

1. **Sample:** Pick the 1st and 3rd columns:

$$C = \begin{pmatrix} 9 & 3 \\ 6 & 2 \\ 3 & 1 \\ 1 & 0.25 \end{pmatrix}$$

2. **Intersect:** Take the 1st and 3rd rows from C to form W :

$$W = \begin{pmatrix} 9 & 3 \\ 3 & 1 \end{pmatrix}$$

3. **Approximate:** Suppose the pseudoinverse of W is:

$$W^+ = \begin{pmatrix} 0.09 & -0.27 \\ -0.27 & 0.81 \end{pmatrix}$$

Then,

$$\tilde{K} = CW^+C^T = \begin{pmatrix} 9 & 6 & 3 & 0.675 \\ 6 & 4 & 2 & 0.45 \\ 3 & 2 & 1 & 0.225 \\ 0.675 & 0.45 & 0.225 & 0.05 \end{pmatrix}$$

$\tilde{K}$ is a good approximation of the original K , especially in the top-left portion.

72.6.4. Why Is This Useful?

- **Speed:** The Nyström method is much faster than computing the full kernel matrix. The complexity is roughly $O(l^2n)$ instead of $O(n^2d)$ (where d is the number of features).
- **Scalability:** It allows kernel methods (like SVM or Kernel PCA) to be used on much larger datasets.

- **Feature Mapping:** The method can be used to project new data points into the same feature space for prediction tasks.

The quality of the approximation depends on the columns you sample. Uniform random sampling is common and often effective, but more advanced techniques exist to select more informative columns.

72.6.5. Applying the Nyström Approximation: How Nyström Approximation Helps Kriging

Kriging can significantly benefit from the Nyström approximation, especially when dealing with large datasets. Kriging is a spatial interpolation method used to estimate values at unmeasured locations based on observed points. It relies on a **covariance matrix** (often denoted as $\mathbf{K}$) that describes the spatial correlation between all observed data points.

The Problem with Standard Kriging:

The main computational challenge in Kriging is solving for the weights needed for prediction, which requires **inverting the covariance matrix $\mathbf{K}$** . For n data points, $\mathbf{K}$ is an $n \times n$ matrix, and inverting it has computational complexity $O(n^3)$. This becomes impractical for large datasets.

The Nyström Solution:

Since the covariance matrix in Kriging is a type of kernel matrix, we can use the Nyström method to create a low-rank approximation, $\tilde{\mathbf{K}}$. Instead of inverting the full matrix, we use the **Woodbury matrix identity** on the Nyström approximation, allowing us to efficiently compute $\tilde{\mathbf{K}}^{-1}$ without forming the full matrix. This reduces computational complexity to roughly $O(l^2n)$, where l is the number of sampled columns.

In summary, Nyström makes Kriging feasible for large-scale problems by replacing expensive matrix inversion with a faster, memory-efficient approximation.

72.6.6. Example: Predicting Temperature with Nyström-Kriging

Suppose we have temperature readings from 100 weather stations ($n=100$) and want to predict the temperature at a new location.

Data:

- Observed Locations ($\mathbf{X}$): 100 coordinate pairs
- Observed Temperatures ($\mathbf{y}$): 100 values
- Prediction Location ($\mathbf{x}^*$): Coordinates of the new location

72.6.6.1. Step 1: Nyström Approximation of the Covariance Matrix

1. **Sample Representative Points:** Randomly select 1=10 stations as landmarks.
2. **Compute C and W:**
 - **C:** Covariance between all 100 stations and the 10 landmarks (100x10 matrix)
 - **W:** Covariance among the 10 landmarks (10x10 matrix)

Nyström approximation: $\tilde{K} = CW^+C^T$

72.6.6.2. Step 2: Modeling and Prediction

Standard Kriging prediction:

$$y(x^*) = \mathbf{k}^{*T} \mathbf{K}^{-1} \mathbf{y}$$

where $\mathbf{k}^{*T}$ is the covariance vector between the prediction location and all observed locations.

Nyström-Kriging prediction:

$$y(x^*) \approx \mathbf{k}^{*T}(\text{fast\_approx\_inverse}(\mathbf{C}, \mathbf{W}))\mathbf{y}$$

Prediction Steps:

1. Calculate $\mathbf{k}^{*T}$: Covariance between new location and all stations.
2. Approximate the inverse term using the Woodbury identity with **C** and **W**.
3. Make the prediction: Take the dot product of $\mathbf{k}^{*T}$ and the weights vector.

This yields an accurate prediction efficiently, enabling rapid mapping for large regions.

72.6.7. Details: Woodbury Matrix Identity for Avoiding the Big Inversion

First, what is the **Woodbury matrix identity**? It's a mathematical rule that tells you how to find the inverse of a matrix that's been modified slightly. Its most useful form is for a "low-rank update":

$$(A + UCV)^{-1} = A^{-1} - A^{-1}U(C^{-1} + VA^{-1}U)^{-1}VA^{-1}$$

This looks complicated, but the core idea is simple:

- If you have a matrix A that is **easy to invert** (like a diagonal matrix).
- And you add a low-rank matrix to it (the UCV part, where C is small).

- You can find the new inverse without directly inverting the big $(A + UCV)$ matrix. Instead, you only need to invert the much smaller matrix in the middle of the formula: $(C^{-1} + VA^{-1}U)$.

How does this apply to the Nyström approximation?

In many machine learning and Kriging applications, we don't just need the kernel matrix $\tilde{K}$, but a "regularized" version, $(\lambda I + \tilde{K})$, where λI is a diagonal matrix that helps prevent overfitting. We need to find the inverse of this:

$$(\lambda I + \tilde{K})^{-1}$$

Substituting the Nyström formula $\tilde{K} = CW^+C^T$, we get:

$$(\lambda I + CW^+C^T)^{-1}$$

This expression fits the Woodbury identity perfectly!

- $A = \lambda I$ (very easy to invert: $A^{-1} = \frac{1}{\lambda}I$)
- $U = C$ (our $n \times l$ matrix)
- C (middle matrix) $= W^+$ (our small $l \times l$ matrix)
- $V = C^T$ (our $l \times n$ matrix)

By plugging these into the Woodbury formula, we get an expression for the inverse that only requires inverting a small $l \times l$ matrix. This means we never have to build the full $n \times n$ matrix $\tilde{K}$ or invert it directly. This is the source of the massive speed-up.

72.6.8. The Example: Step-by-Step

Let's reuse our 4-point example and show both the slow way and the fast Woodbury way.

Recall our matrices:

- $C = \begin{pmatrix} 9 & 3 \\ 6 & 2 \\ 3 & 1 \\ 1 & 0.25 \end{pmatrix}$
- $W^+ = \begin{pmatrix} 0.09 & -0.27 \\ -0.27 & 0.81 \end{pmatrix}$
- Let's use a regularization value $\lambda = 0.1$.

72.6.8.1. Method 1: The Slow Way (Forming the full matrix)

1. **Construct $\tilde{K}$:** First, we explicitly calculate the full 4×4 Nyström approximation $\tilde{K} = CW^+C^T$.

$$\tilde{K} = \begin{pmatrix} 9 & 6 & 3 & 0.675 \\ 6 & 4 & 2 & 0.45 \\ 3 & 2 & 1 & 0.225 \\ 0.675 & 0.45 & 0.225 & 0.05 \end{pmatrix}$$

2. **Add the regularization:** Now we compute $(\lambda I + \tilde{K})$.

$$(\lambda I + \tilde{K}) = \begin{pmatrix} 9.1 & 6 & 3 & 0.675 \\ 6 & 4.1 & 2 & 0.45 \\ 3 & 2 & 1.1 & 0.225 \\ 0.675 & 0.45 & 0.225 & 0.15 \end{pmatrix}$$

3. **Invert the 4×4 matrix:** This is the expensive step. The result is:

$$(\lambda I + \tilde{K})^{-1} \approx \begin{pmatrix} 9.85 & -14.78 & -0.07 & 0.27 \\ -14.78 & 22.22 & 0.09 & -0.41 \\ -0.07 & 0.09 & 0.91 & -0.03 \\ 0.27 & -0.41 & -0.03 & 6.67 \end{pmatrix}$$

This works for our tiny 4×4 example, but it would be computationally infeasible if n was 10,000.

72.6.8.2. Method 2: The Fast Way (Using Woodbury Identity)

We use the Woodbury formula to get the same result without ever creating a 4×4 matrix. The formula simplifies to:

$$(\lambda I + \tilde{K})^{-1} = \frac{1}{\lambda} I - \frac{1}{\lambda^2} C \left(W + \frac{1}{\lambda} C^T C \right)^{-1} C^T$$

1. **Compute the small 2×2 pieces:**

- $C^T C = \begin{pmatrix} 127 & 42.25 \\ 42.25 & 14.0625 \end{pmatrix}$
- $W = \begin{pmatrix} 9 & 3 \\ 3 & 1 \end{pmatrix}$

72. Matrices

- The matrix to invert is $W + \frac{1}{0.1}C^TC = W + 10 \cdot (C^TC)$, which is:

$$\begin{pmatrix} 9 & 3 \\ 3 & 1 \end{pmatrix} + \begin{pmatrix} 1270 & 422.5 \\ 422.5 & 140.625 \end{pmatrix} = \begin{pmatrix} 1279 & 425.5 \\ 425.5 & 141.625 \end{pmatrix}$$

2. **Invert the small 2×2 matrix:** This is the only inversion we need, and it's extremely fast.

$$(W + \frac{1}{\lambda}C^TC)^{-1} \approx \begin{pmatrix} 0.22 & -0.66 \\ -0.66 & 1.99 \end{pmatrix}$$

3. **Combine the results:** Now we plug this small inverse back into the full formula. The rest is just matrix multiplication, no more inversions.

- First, calculate the middle term: $M = \frac{1}{\lambda^2}C(\dots)^{-1}C^T$. This will result in a 4×4 matrix.
- Then, calculate the final result: $\frac{1}{\lambda}I - M$.

After performing these multiplications, you will get the **exact same 4×4 inverse matrix** as in the slow method.

The crucial difference is that the most expensive operation—the matrix inversion—was performed on a tiny 2×2 matrix instead of a 4×4 one. For a large-scale problem, this is the difference between a calculation that takes seconds and one that could take hours or even be impossible.

72.7. Extending spotpython's Kriging Surrogate with Nyström Approximation for Enhanced Scalability

72.7.1. Introduction: Overcoming the Scalability Challenge in Kriging for Sequential Optimization

The Sequential Parameter Optimization Toolbox (spotpython) is a framework for hyperparameter tuning and black-box optimization based on Sequential Model-Based Optimization (SMBO). At the core of SMBO lies a surrogate model that approximates the true, expensive objective. Kriging (Gaussian Process regression) is a premier choice because it provides both predictions and a principled measure of uncertainty. This uncertainty enables a balance between exploration and exploitation. In each SMBO iteration, the Kriging model is updated with new evaluations, refining its approximation and proposing the next points.

Standard Kriging requires constructing and inverting an $n \times n$ covariance matrix, where n is the number of data points. Matrix inversion scales as $O(n^3)$. During SMBO, n can reach hundreds or thousands; refitting the surrogate each iteration becomes

prohibitively expensive. This cubic scaling is the key obstacle to applying Kriging at larger scales.

We integrate the Nyström method into the `spotpython` Kriging class. The Nyström method yields a low-rank approximation of a symmetric positive semidefinite (SPSD) kernel matrix by selecting $l \ll n$ “landmark” points. It approximates the full $n \times n$ covariance while requiring inversion of only an $l \times l$ matrix, reducing fitting cost from $O(n^3)$ to $O(nl^2)$. This makes Kriging viable even when the number of function evaluations is large.

72.7.2. Report Objectives and Structure

- Review theoretical foundations of Kriging and Nyström approximation
- Present documented Python code updates for Kriging (as in `kriging.py`)
- Explain changes to `__init__`, `fit`, and `predict`
- Show how mixed variable types are preserved via `build_Psi` and `build_psi_vec`
- Provide practical usage guidance and a formal complexity analysis

72.8. Theoretical Foundations: The Nyström–Kriging Framework

72.8.1. A Primer on Kriging (Gaussian Process Regression)

Kriging models $f(x)$ as a Gaussian Process with mean function $m(\cdot)$ and covariance (kernel) $k(\cdot, \cdot)$. For training inputs $X = \{x_1, \dots, x_n\}$ and observations $y = \{y_1, \dots, y_n\}$:

$$y \sim \mathcal{N}(m(X), K(X, X) + \sigma_n^2 I)$$

For a new point x_* :

$$\mu(x_*) = k(x_*, X) [K(X, X) + \sigma_n^2 I]^{-1} y$$

$$\sigma^2(x_*) = k(x_*, x_*) - k(x_*, X) [K(X, X) + \sigma_n^2 I]^{-1} k(X, x_*)$$

The challenge is inverting the $n \times n$ matrix $K(X, X) + \sigma_n^2 I$.

72.8.2. The Nyström Method for Low-Rank Kernel Approximation

Select l landmark points $X_m \subset X$. Let: - $C = K_{nm} = K(X, X_m) \in \mathbb{R}^{n \times l}$ - $W = K_{mm} = K(X_m, X_m) \in \mathbb{R}^{l \times l}$ Then the Nyström approximation is:

$$\tilde{K}_{nn} = C W^+ C^\top = K_{nm} K_{mm}^+ K_{mn}$$

where W^+ is the pseudoinverse of W . The approximation has rank $\leq l$.

72.8.3. Justification for Landmark Selection

Uniform sampling without replacement is an effective and inexpensive strategy for selecting landmarks across varied datasets and kernels.

72.9. Implementation: A Scalable Kriging Class for spotpython

72.9.1. Updated kriging.py with Nyström Approximation (excerpt)

```
"""
Kriging surrogate with optional Nyström approximation.
"""
import numpy as np
from scipy.spatial.distance import cdist
from scipy.linalg import cholesky, cho_solve, solve_triangular

class Kriging:
    def __init__(self, fun_control, n_theta=None, theta=None, p=2.0,
                  corr="squared_exponential", isotropic=False,
                  approximation="None", n_landmarks=100):
        self.fun_control = fun_control
        self.dim = self.fun_control["lower"].shape
        self.p = p
        self.corr = corr
        self.isotropic = isotropic
        self.approximation = approximation
        self.n_landmarks = n_landmarks
        self.factor_mask = self.fun_control["var_type"] == "factor"
```

```

self.ordered_mask = ~self.factor_mask
self.n_theta = 1 if isotropic else (n_theta or self.dim)
self.theta = np.full(self.n_theta, 0.1) if theta is None else theta
self.X_, self.y_, self.L_, self.alpha_ = None, None, None, None
self.landmarks_, self.W_cho_, self.nystrom_alpha_ = None, None, None

def fit(self, X, y):
    self.X_, self.y_ = X, y
    n_samples = X.shape[0]
    if self.approximation.lower() == "nystroem" and n_samples > self.n_landmarks:
        return self._fit_nystrom(X, y)
    return self._fit_standard(X, y)

def _fit_standard(self, X, y):
    Psi = self.build_Psi(X, X)
    Psi[np.diag_indices_from(Psi)] += 1e-8
    try:
        self.L_ = cholesky(Psi, lower=True)
        self.alpha_ = cho_solve((self.L_, True), y)
    except np.linalg.LinAlgError:
        self.L_ = None
        self.alpha_ = np.linalg.pinv(Psi) @ y

def _fit_nystrom(self, X, y):
    n_samples = X.shape[0]
    idx = np.random.choice(n_samples, self.n_landmarks, replace=False)
    self.landmarks_ = X[idx, :]
    W = self.build_Psi(self.landmarks_, self.landmarks_) + 1e-8 * np.eye(self.n_landmarks)
    C = self.build_Psi(X, self.landmarks_)
    try:
        self.W_cho_ = cholesky(W, lower=True)
        self.nystrom_alpha_ = cho_solve((self.W_cho_, True), C.T @ y)
    except np.linalg.LinAlgError:
        self.W_cho_ = None
        self._fit_standard(X, y)

def predict(self, X_star):
    if self.approximation.lower() == "nystroem" and self.landmarks_ is not None:
        return self._predict_nystrom(X_star)
    return self._predict_standard(X_star)

def _predict_standard(self, X_star):
    psi = self.build_Psi(X_star, self.X_)
    y_pred = psi @ self.alpha_

```

```

    if self.L_ is not None:
        v = solve_triangular(self.L_, psi.T, lower=True)
        y_mse = 1.0 - np.sum(v**2, axis=0)
    else:
        Psi = self.build_Psi(self.X_, self.X_) + 1e-8 * np.eye(self.X_.shape[0])
        pi_Psi = np.linalg.pinv(Psi)
        y_mse = 1.0 - np.sum((psi @ pi_Psi) * psi, axis=1)
    y_mse[y_mse < 0] = 0
    return y_pred, y_mse.reshape(-1, 1)

def _predict_nystrom(self, X_star):
    psi_star_m = self.build_Psi(X_star, self.landmarks_)
    y_pred = psi_star_m @ self.nystrom_alpha_
    if self.W_cho_ is not None:
        v = cho_solve((self.W_cho_, True), psi_star_m.T)
        quad = np.sum(psi_star_m * v.T, axis=1)
        y_mse = 1.0 - quad
    else:
        y_mse = np.ones(X_star.shape[0])
    y_mse[y_mse < 0] = 0
    return y_pred, y_mse.reshape(-1, 1)

def build_Psi(self, X1, X2):
    n1 = X1.shape[0]
    Psi = np.zeros((n1, X2.shape[0]))
    for i in range(n1):
        Psi[i, :] = self.build_psi_vec(X1[i, :], X2)
    return Psi

def build_psi_vec(self, x, X_):
    theta10 = np.full(self.dim, 10**self.theta) if self.isotropic else 10**self.t
    D = np.zeros(X_.shape[0])
    if self.ordered_mask.any():
        Xo = X_[ :, self.ordered_mask]
        xo = x[self.ordered_mask]
        D += cdist(xo.reshape(1, -1), Xo, metric="sqeuclidean",
                    w=theta10[self.ordered_mask]).ravel()
    if self.factor_mask.any():
        Xf = X_[ :, self.factor_mask]
        xf = x[self.factor_mask]
        D += cdist(xf.reshape(1, -1), Xf, metric="hamming",
                    w=theta10[self.factor_mask]).ravel() * self.factor_mask.sum()
    return np.exp(-D) if self.corr == "squared_exponential" else np.exp(-(D**self

```


72.10. Implementation Details

72.10.1. Architectural Enhancements to `init`

- New argument `approximation="None"` for backward-compatible selection between exact Kriging and Nyström
- New argument `n_landmarks` (default 100) controls the number of inducing points when using Nyström
- State attributes for both exact and Nyström paths are maintained separately

72.10.2. The `fit()` Method: A Dual-Pathway Approach

- Dispatcher selecting exact or Nyström pathway
- The Nyström fit Pathway (`_fit_nystrom`):
 - Landmark selection via uniform sampling without replacement
 - Core matrices:
 - * $W = K_{mm}$ (landmark-landmark)
 - * $C = K_{nm}$ (data-landmark)
 - Cholesky factorization of W (with jitter) for stability
 - Pre-computation: $\alpha_{nys} = W^{-1}C^T y$ via `cho_solve`
- The Standard fit Pathway (`_fit_standard`):
 - Full Ψ construction, Cholesky decomposition, and solve for α
 - Fallback to pseudoinverse if Cholesky fails

72.10.3. The `predict()` Method: Conditional Prediction Logic

- Routes to Nyström or standard prediction path based on fitted model state
- The Nyström predict Pathway (`_predict_nystrom`):
 - Cross-covariance ψ between test points and landmarks
 - Mean: $\psi \cdot \alpha_{nys}$
 - Variance: uses `cho_solve` with W Cholesky; non-negative clipping
- The Standard predict Pathway (`_predict_standard`):
 - Cross-covariance with all training points
 - Mean from α ; variance via triangular solves or pseudoinverse fallback

72.10.4. Critical Detail: Preserving Mixed Variable Type Functionality

The Significance of `build_psi_vec`:

- Mixed spaces: continuous (ordered) and categorical (factor) variables
- Distances:
 - Weighted squared Euclidean for ordered variables
 - Weighted Hamming for factors
- Anisotropic kernel via per-dimension length-scales θ
- Nyström path reuses `build_Psi` $\rightarrow$ `build_psi_vec`, preserving mixed-type handling

72.10.5. Seamless Integration into the Nyström Workflow

All covariance computations (W , C , predictive cross-covariance) use `build_Psi`, ensuring identical handling for mixed variable types in both standard and Nyström modes.

72.11. Jupyter Notebook

Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

73. Infill Criteria

In the context of computer experiments and surrogate modeling, a sampling plan refers to the set of input values, often denoted as X , at which a computer code is evaluated. The primary objective of a sampling plan is to efficiently explore the input space to understand the behavior of the computer code and to construct a surrogate model that accurately represents that behavior. Historically, Response Surface Methodology (RSM) provided methods for designing such plans, often based on rectangular grids or factorial designs. More recently, Design and Analysis of Computer Experiments (DACE) has emerged as a more flexible and powerful approach for this purpose.

A surrogate model, or $\hat{f}$, is built to approximate the expensive response of a black-box function $f(x)$. Since evaluating f is costly, only a sparse set of samples is used to construct $\hat{f}$, which can then provide inexpensive predictions for any point in the design space. However, as a surrogate model is inherently an approximation of the true function, its accuracy and predictive capabilities can be significantly improved by incorporating new data points, known as infill points. Infill points are strategically chosen to either reduce uncertainty, improve predictions in specific regions of interest, or enhance the model's ability to identify optima or trends.

The process of updating a surrogate model with infill points is iterative. It typically involves:

- **Identifying Regions of Interest:** Analyzing the current surrogate model to determine areas where it is inaccurate, has high uncertainty, or predicts promising results (e.g., potential optima).
- **Selecting Infill Points:** Choosing new data points based on specific criteria that balance different objectives.
- **Evaluating the True Function:** Running the actual simulation or experiment at the selected infill points to obtain their corresponding outputs.
- **Updating the Surrogate Model:** Retraining or updating the surrogate model using the new, augmented dataset.
- **Repeating:** Iterating this process until the model meets predefined accuracy criteria or the computational budget is exhausted.

73.1. Balancing Exploitation and Exploration

A crucial aspect of selecting infill points is navigating the inherent trade-off between exploitation and exploration.

Definition 73.1 (Exploitation). Exploitation refers to sampling near predicted optima to refine the solution. This strategy aims to rapidly converge on a good solution by focusing computational effort where the surrogate model suggests the best values might lie.

Definition 73.2 (Exploration). Exploration involves sampling in regions of high uncertainty to improve the global accuracy of the model. This approach ensures that the model is well-informed across the entire design space, preventing it from getting stuck in local optima.

Forrester, Sóbester, and Keane (2008) emphasizes that effective infill criteria are designed to combine both exploitation and exploration.

73.2. Expected Improvement (EI)

Expected Improvement (EI) is one of the most influential and widely-used infill criteria. Formalized by Jones, Schonlau, and Welch (1998) and building upon the work of Moćkus (1974), EI provides a mathematically elegant framework that naturally balances exploitation and exploration. Rather than simply picking the point with the best predicted value (pure exploitation) or the point with the highest uncertainty (pure exploration), EI asks a more nuanced question: “How much improvement over the current best solution can we *expect* to gain by evaluating the true function at a new point x ?”

The Expected Improvement, $EI(x)$, can be calculated using the following formula:

$$EI(x) = \sigma(x) [Z\Phi(Z) + \phi(Z)]$$

where:

- $\mu(x)$ (or $\hat{y}(x)$) is the Kriging prediction (mean of the stochastic process) at a new, unobserved point x .
- $\sigma(x)$ (or $\hat{s}(x)$) is the estimated standard deviation (square root of the variance $\hat{s}^2(x)$) of the prediction at point x .
- f_{best} (or y_{min}) is the best (minimum, for minimization problems) observed function value found so far.
- $Z = \frac{f_{best} - \mu(x)}{\sigma(x)}$ is the standardized improvement.

73.3. Expected Improvement in the Hyperparameter Tuning Cookbook (Python Implementation)

- $\Phi(Z)$ is the cumulative distribution function (CDF) of the standard normal distribution.
- $\phi(Z)$ is the probability density function (PDF) of the standard normal distribution.

If $\sigma(x) = 0$ (meaning there is no uncertainty at point x , typically because it's an already sampled point), then $EI(x) = 0$, reflecting the intuition that no further improvement can be expected at a known point. A maximization of Expected Improvement as an infill criterion will eventually lead to the global optimum.

The elegance of the EI formula lies in its combination of two distinct terms:

- **Exploitation Term:** $(f_{best} - \mu(x))\Phi(Z)$. This part of the formula contributes more when the predicted value $\mu(x)$ is significantly lower (better) than the current best observed value f_{best} . It is weighted by the probability $\Phi(Z)$ that the true function value at x will indeed be an improvement over f_{best} .
- **Exploration Term:** $\sigma(x)\phi(Z)$. This term becomes larger when there is high uncertainty ($\sigma(x)$ is large) in the model's prediction at x . It accounts for the potential of discovering unexpectedly good values in areas that have not been thoroughly explored, even if the current mean prediction there is not the absolute best.

Expected Improvement offers several significant practical benefits:

- **Automatic Balance:** It inherently balances exploitation and exploration without requiring any manual adjustment of weights or parameters.
- **Scale Invariance:** EI is relatively insensitive to the scaling of the objective function, making it robust across various problem types.
- **Theoretical Foundation:** It is underpinned by a strong theoretical basis derived from decision theory and information theory.
- **Efficient Optimization:** The smooth and differentiable nature of the EI function allows for efficient optimization using gradient-based algorithms to find the next infill point.
- **Proven Performance:** EI has demonstrated consistent and strong performance in numerous real-world applications across various domains.

73.3. Expected Improvement in the Hyperparameter Tuning Cookbook (Python Implementation)

Within the context of the Hyperparameter Tuning Cookbook, Expected Improvement serves a critical role in Sequential Model-Based Optimization. It systematically guides the selection of which hyperparameter configurations to evaluate next, facilitating the efficient utilization of computational resources. By intelligently balancing the need to exploit promising regions and explore uncertain areas, EI helps identify optimal

73. Infill Criteria

hyperparameters with a reduced number of expensive model training runs. This provides a principled and automated method for navigating complex hyperparameter spaces without extensive manual intervention.

While the foundational concepts in Forrester, Sóbester, and Keane (2008) are often illustrated with MATLAB code, the Hyperparameter Tuning Cookbook emphasizes and provides implementations in Python. The `spotpython` package, consistent with the Cookbook's approach, provides a Python implementation of Expected Improvement within its Kriging class. For minimization problems, `spotpython` typically calculates and returns the negative Expected Improvement, aligning with standard optimization algorithm conventions. Furthermore, to enhance numerical stability and mitigate issues when EI values are very small, `spotpython` often works with a logarithmic transformation of EI and incorporates a small epsilon value.

73.4. Jupyter Notebook

Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

74. Remote Optimization Challenge: A Benchmarking Study

This report performs an experimental comparison of two optimization algorithms on a remote objective function. The goal is to evaluate the efficiency and effectiveness of `SpotOptim` (a surrogate-based optimizer) compared to a standard gradient-based approach (Scipy L-BFGS-B) in a limited budget scenario.

74.1. Benchmarking Best Practices

Benchmarking is a critical aspect of validating optimization algorithms. According to *Bartz-Beielstein et al. (2020)* in “Benchmarking in Optimization: Best Practice and Open Issues” (arXiv:2007.03488), a sound benchmarking study should adhere to several key principles:

1. **Clearly Stated Goals:** The objective of this study is to compare the convergence speed and final solution quality of two distinct optimization strategies given a constrained budget of function evaluations.
2. **Well-Specified Problems:** We employ the “Wing Weight” problem, hosted remotely, which represents a realistic, expensive black-box engineering problem. It is a 10-dimensional problem defined on the unit hypercube $[0, 1]^{10}$.
3. **Adequate Performance Measures:** We report the *best-so-far* objective value as a function of the number of evaluations (convergence) and the distribution of the final solutions found.
4. **Repetitions:** Since the remote function (and potentially the algorithms) may exhibit stochastic behavior (noise in the function or random initialization in the algorithms), we perform multiple independent repetitions to ensure statistical reliability.

74.2. Experimental Setup

74.2.1. The Remote Objective Function

The objective function is accessed via `objective_remote`. It simulates the weight of a light aircraft wing.

74. Remote Optimization Challenge: A Benchmarking Study

- **Input:** $X \in [0, 1]^{10}$
- **Output:** $y \in \mathbb{R}$ (to be minimized)
- **Characteristics:** Expensive to evaluate, potentially noisy.

74.2.2. Algorithms

1. **SpotOptim:** A Sequential Parameter Optimization tool that uses surrogate models (e.g., Kriging/Gaussian Processes) to guide the search. It is designed to be sample-efficient.
2. **Scipy L-BFGS-B:** A standard quasi-Newton method. While efficient for smooth local optimization, it typically requires many function evaluations to estimate gradients and may get stuck in local optima. We use random restarts (via the repetitions) to explore different basins.

74.2.3. Budget and Settings

- **Maximum Evaluations:** 50 per run.
- **Repetitions:** 5 independent runs (kept low for demonstration speed, usually 10-30 are recommended).
- **SpotOptim Settings:** Initial design $n = 10$, surrogate-based optimization for the remaining budget.
- **Scipy Settings:** L-BFGS-B with numeric differentiation.

74.3. Comparison of the Difficulty

To estimate the landscape difficulty of the remote function, we perform random linear cuts through the input space and plot the objective values along these cuts. We compare the following four functions:

1. Wing Weight (10D)
2. Lennard-Jones Potential (39D)
3. Remote Objective (10D), SPOTSeven Optimization Challenge

```
import matplotlib.pyplot as plt
import time
import numpy as np
from spotoptim.function.remote import objective_remote
from spotoptim.function.so import wingwt, lennard_jones

def get_landscape_cut(func, dim, steps=200):
    np.random.seed(42)
```



```

# Define two random points in  $[0, 1]^{\text{dim}}$ 
start_point = np.random.rand(dim)
end_point = np.random.rand(dim)

# Interpolate
alphas = np.linspace(0, 1, steps)
# Shape (steps, dim)
trajectory = np.outer(1 - alphas, start_point) + np.outer(alphas, end_point)

# Time execution
t0 = time.time()
values = func(trajectory)
dt = time.time() - t0

return alphas, values, dt * 1000 # ms

# Run Analysis
steps = 300

# 1. Wing Weight (10 dim)
a1, v1, t1 = get_landscape_cut(wingwt, 10, steps)

# 2. Lennard-Jones (39 dim)
a2, v2, t2 = get_landscape_cut(lennard_jones, 39, steps)

# 3. Challenge Remote (10 dim)
a3, v3, t3 = get_landscape_cut(objective_remote, 10, steps)

fig, axes = plt.subplots(3, 1, figsize=(10, 9))

# Plot 1: Wing Weight
axes[0].plot(a1, v1, color='blue', linewidth=2)
axes[0].set_title(f"Wing Weight (10D)\nSmooth & Unimodal")
axes[0].set_xlabel("Interpolation  $\alpha$ ")
axes[0].set_ylabel("Weight")
axes[0].grid(True, alpha=0.3)

# Plot 2: Lennard-Jones
axes[1].plot(a2, v2, color='green', linewidth=2)
axes[1].set_title(f"Lennard-Jones (39D)\nRugged & Multimodal")
axes[1].set_xlabel("Interpolation  $\alpha$ ")
axes[1].set_ylabel("Potential Energy")
axes[1].grid(True, alpha=0.3)

```

74. Remote Optimization Challenge: A Benchmarking Study

```
# Plot 3: Remote Challenge
axes[2].plot(a3, v3, color='red', linewidth=2)
axes[2].set_title(f"Remote Challenge (10D)\nNoisy & Complex")
axes[2].set_xlabel("Interpolation  $\alpha$ ")
axes[2].set_ylabel("Objective Value")
axes[2].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

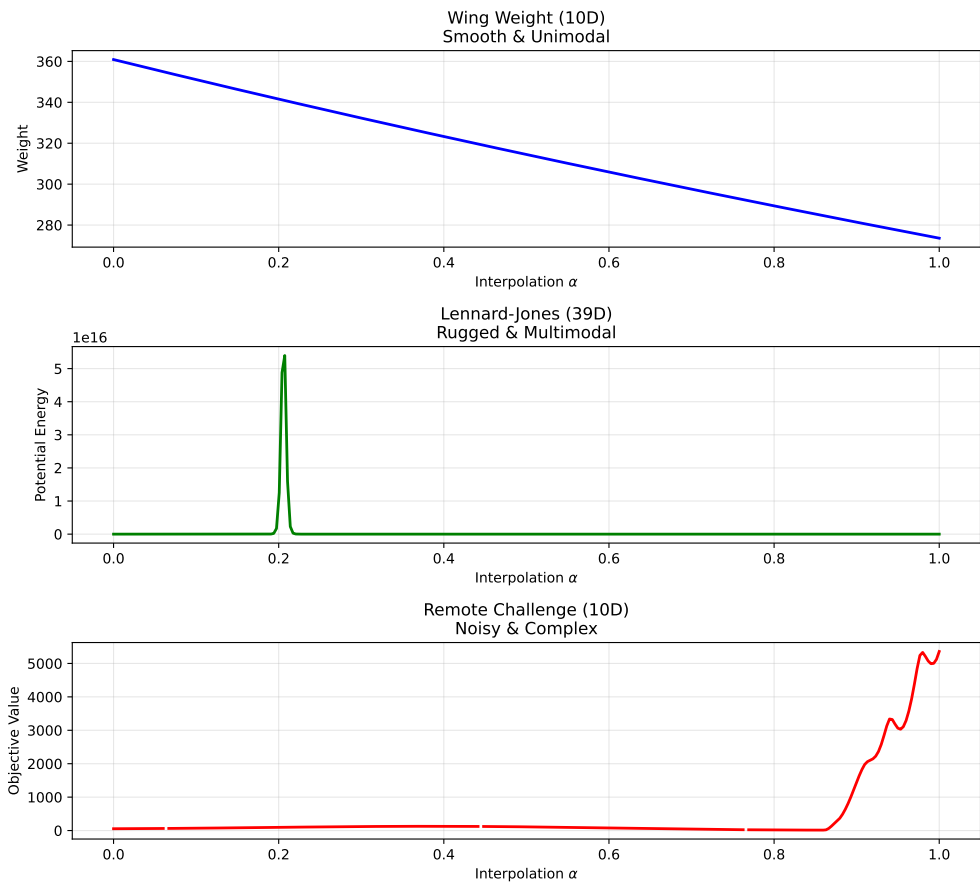


Figure 74.1.: 1D Slice through High-Dimensional Landscapes

74.4. Statistical Power Analysis

To compare the algorithms fairly in the presence of noise, we cannot simply take the final objective value of a single run. The objective function $f(x)$ is noisy, meaning $y = f(x) + \epsilon$. We want to compare the true performance $\mu = E[f(x)]$ of the best solutions found.

We first estimate the noise level (σ) of the objective function and then determine the number of repetitions (n) required to distinguish between two solutions with sufficient statistical power.

74.4.1. 1. Estimating Noise (σ)

We evaluate a random design point 30 times to estimate the standard deviation of the noise.

```
import numpy as np
import statistics
from spopt.optim.function.remote import objective_remote
from statsmodels.stats.power import TTestIndPower

# 1. Estimate Noise
print("Estimating noise...")
x_sample = np.array([[0.5] * 10])
noise_samples = []
for _ in range(30):
    noise_samples.append(objective_remote(x_sample)[0])
# remove None values
noise_samples = [x for x in noise_samples if x is not None]
# check if there are enough values for stdev
if len(noise_samples) < 2:
    raise ValueError("Not enough samples for stdev estimation")
else:
    sigma_est = statistics.stdev(noise_samples)
    print(f"Estimated Noise (sigma): {sigma_est:.4f}")
```

```
Estimating noise...
Estimated Noise (sigma): 0.1009
```

74.4.2. 2. Determining Sample Size (n)

We perform a power analysis to calculate the number of evaluations (n) needed to validate a solution. We aim to detect a “large” effect size (Cohen’s $d = 1.0$, which

74. Remote Optimization Challenge: A Benchmarking Study

corresponds to a difference of $1.0 \times \sigma$) with a power of 0.8 and significance level $\alpha = 0.05$.

```
# 2. Power Analysis
effect_size = 1.0 # Detect difference of 1 sigma
alpha = 0.05      # Significance level
power = 0.8       # Statistical power

analysis = TTestIndPower()
n_required = analysis.solve_power(
    effect_size=effect_size,
    nobs1=None,
    ratio=1.0,
    alpha=alpha,
    power=power
)
n_validate = int(np.ceil(n_required))
print(f"Required samples (n) for validation (d={effect_size}, power={power}): {n_validate}")
```

Required samples (n) for validation (d=1.0, power=0.8): 17

74.5. Experimental Comparison

We perform m independent optimization runs for each algorithm. For each run, we take the *best design vector* found (x^*) and evaluate it n times to obtain a robust estimate of its true quality.

- **Number of Optimization Runs (m):** 5
- **Budget per Run:** 50 evaluations
- **Validation Size (n):** `n_validate` (calculated above)

```
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from spotoptim import SpotOptim

# --- Configuration ---
M_RUNS = 5 # Number of independent optimization runs
MAX_EVALS = 50 # Optimization budget
DIM = 10
BOUNDS = [(0.0, 1.0) for _ in range(DIM)]

# Storage for VALIDATED means
final_means_spot = []
```

```

final_means_scipy = []

# Storage for NOISY history (for convergence plots)
results_spot_hist = []
results_scipy_hist = []

def get_validated_mean(x_best, n_evals):
    """Evaluate x_best n_evals times and return the mean."""
    vals = []
    # Reshape for remote call if needed (SpotOptim returns 1D or 2D depending on version, handle both)
    X = np.atleast_2d(x_best)

    # Batch evaluation if objective supports it, or loop
    # objective_remote supports batch, let's use it to save overhead
    # Create batch of n_evals identical rows
    X_batch = np.repeat(X, n_evals, axis=0)
    y_batch = objective_remote(X_batch)
    return np.mean(y_batch)

print(f"Starting Benchmark: {M_RUNS} runs, opt budget {MAX_EVALS}, validation n={n_validate}...")

# --- 1. SpotOptim Loop ---
print("\n--- SpotOptim ---")
for i in range(M_RUNS):
    print(f"Run {i+1}/{M_RUNS}")

    optimizer = SpotOptim(
        fun=objective_remote,
        bounds=BOUNDS,
        max_iter=MAX_EVALS,
        n_initial=10,
        acquisition='y',
        seed=i,
        verbose=False
    )

    try:
        res = optimizer.optimize()
        x_best = res.x

        # Collect History
        y_hist = res.y.flatten().tolist()
        results_spot_hist.append(y_hist)

```

```

        # Validation Step
        val_mean = get_validated_mean(x_best, n_validate)
        final_means_spot.append(val_mean)
        print(f" Best found: {res.fun:.4f} -> Validated Mean: {val_mean:.4f}")

    except Exception as e:
        print(f" Error in SpotOptim run {i}: {e}")

# --- 2. Scipy L-BFGS-B Loop ---
print("\n--- Scipy L-BFGS-B ---")

# Wrapper for Scipy
class HistoryWrapper:
    def __init__(self, func):
        self.func = func
        self.best_y = np.inf
        self.best_x = None
        self.history = []
        self.count = 0

    def __call__(self, x):
        if self.count >= MAX_EVALS:
            return self.best_y # Stop

        X = x.reshape(1, -1)
        y = self.func(X)[0]

        self.history.append(y)

        if y < self.best_y:
            self.best_y = y
            self.best_x = x.copy()

        self.count += 1
        return y

for i in range(M_RUNS):
    print(f"Run {i+1}/{M_RUNS}")

    np.random.seed(i)
    x0 = np.random.rand(DIM)

    wrapper = HistoryWrapper(objective_remote)

```

```

try:
    minimize(
        wrapper,
        x0,
        method='L-BFGS-B',
        bounds=BOUNDS,
        options={'maxfun': MAX_EVALS}
    )

    # Collect History
    results_scipy_hist.append(wrapper.history)

    # Validation Step
    if wrapper.best_x is not None:
        val_mean = get_validated_mean(wrapper.best_x, n_validate)
        final_means_scipy.append(val_mean)
        print(f" Best found: {wrapper.best_y:.4f} -> Validated Mean: {val_mean:.4f}")
    else:
        print(" No solutions evaluated.")

except Exception as e:
    print(f" Error in Scipy run {i}: {e}")
    # Append worst case if failed? Or skip.
    # final_means_scipy.append(np.inf)

print("\nExperiments completed.")

```

Starting Benchmark: 5 runs, opt budget 50, validation n=17...

--- SpotOptim ---

Run 1/5

Best found: 23.3024 -> Validated Mean: 23.3744

Run 2/5

Best found: 28.0052 -> Validated Mean: 27.8481

Run 3/5

Best found: 17.0327 -> Validated Mean: 16.9767

Run 4/5

Error in SpotOptim run 3: unsupported operand type(s) for +: 'float' and 'NoneType'

Run 5/5

Best found: 18.5741 -> Validated Mean: 18.4872

--- Scipy L-BFGS-B ---

Run 1/5

Error in Scipy run 0: '<' not supported between instances of 'NoneType' and 'float'

74. Remote Optimization Challenge: A Benchmarking Study

Run 2/5

Error in Scipy run 1: unsupported operand type(s) for +: 'float' and 'NoneType'

Run 3/5

Error in Scipy run 2: unsupported operand type(s) for +: 'float' and 'NoneType'

Run 4/5

Best found: 49.1201 -> Validated Mean: 49.2306

Run 5/5

Best found: 39.6123 -> Validated Mean: 39.7255

Experiments completed.

74.6. Convergence Analysis (Noisy)

We can visualize the progress of the optimizers by plotting the *best-so-far* objective value against the number of function evaluations. Note that because the objective function is noisy, these curves may fluctuate or appear “better” than the true value due to lucky noise samples.

```
def get_best_so_far(history):
    """Convert a list of evaluations to a best-so-far trajectory."""
    bsf = []
    current_min = np.inf
    for val in history:
        if val < current_min:
            current_min = val
        bsf.append(current_min)
    return bsf

def process_histories(results_hist, max_evals):
    matrix = np.zeros((len(results_hist), max_evals))
    for i, hist in enumerate(results_hist):
        if not hist:
            bsf = [np.nan] * max_evals
        else:
            bsf = get_best_so_far(hist)
            if len(bsf) < max_evals:
                bsf += [bsf[-1]] * (max_evals - len(bsf))
            matrix[i, :] = bsf[:max_evals]
    return matrix

# Process data
spot_bsf_matrix = process_histories(results_spot_hist, MAX_EVALS)
scipy_bsf_matrix = process_histories(results_scipy_hist, MAX_EVALS)
```



```

# Calculate stats omitting NaNs
spot_mean = np.nanmean(spot_bsf_matrix, axis=0)
spot_std = np.nanstd(spot_bsf_matrix, axis=0)
scipy_mean = np.nanmean(scipy_bsf_matrix, axis=0)
scipy_std = np.nanstd(scipy_bsf_matrix, axis=0)

# Plot
plt.figure(figsize=(10, 6))
x_axis = np.arange(1, MAX_EVALS + 1)

# SpotOptim
plt.plot(x_axis, spot_mean, label='SpotOptim', color='blue', linewidth=2)
plt.fill_between(x_axis, spot_mean - spot_std, spot_mean + spot_std, color='blue', alpha=0.2)

# Scipy
plt.plot(x_axis, scipy_mean, label='Scipy L-BFGS-B', color='red', linewidth=2)
plt.fill_between(x_axis, scipy_mean - scipy_std, scipy_mean + scipy_std, color='red', alpha=0.2)

plt.xlabel('Function Evaluations')
plt.ylabel('Best Objective Value (Noisy)')
plt.title('Optimization Progress Comparison (Noisy)')
plt.legend()
plt.grid(True, linestyle='--', alpha=0.7)
plt.show()

```

74. Remote Optimization Challenge: A Benchmarking Study

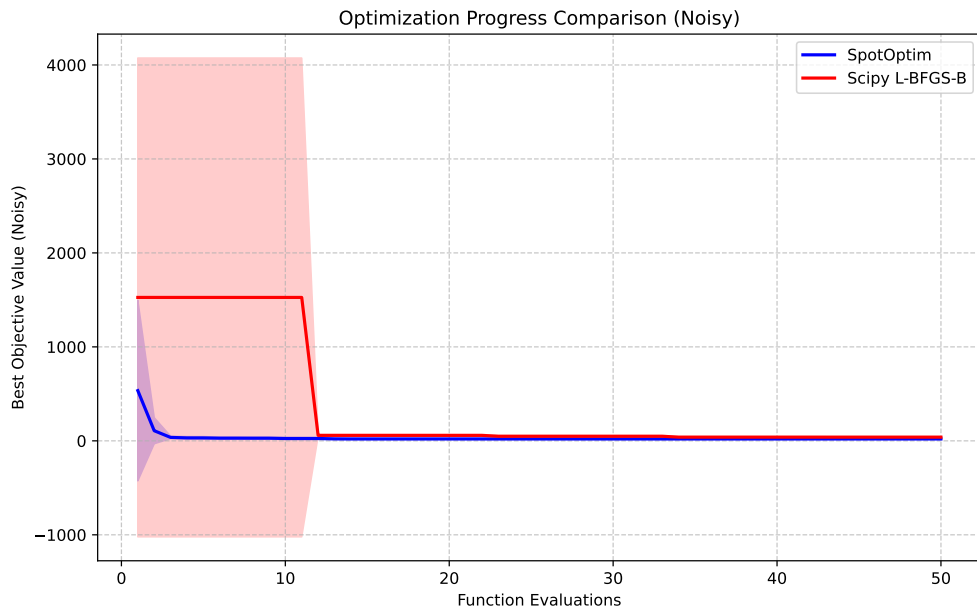


Figure 74.2.: Convergence Plot: Mean Best-So-Far Objective Value (Noisy) vs. Evaluations

74.7. Comparative Analysis

We now statistically compare the validated performance of the two algorithms.

```
import scipy.stats as stats

# 1. Boxplot
plt.figure(figsize=(8, 5))
plt.boxplot([final_means_spot, final_means_scipy], labels=['SpotOptim', 'Scipy L-BFGS-B'])
plt.ylabel('True Objective Value (Estimate)')
plt.title(f'Comparison over {M_RUNS} Runs (Validation n={n_validate})')
plt.grid(True, axis='y', linestyle='--', alpha=0.7)
plt.show()

# 2. T-Test
t_stat, p_val = stats.ttest_ind(final_means_spot, final_means_scipy, equal_var=False)

print("Statistical Comparison (Welch's t-test):")
print(f"SpotOptim Mean: {np.mean(final_means_spot):.4f} (std: {np.std(final_means_spot):.4f})")
print(f"Scipy L-BFGS-B Mean: {np.mean(final_means_scipy):.4f} (std: {np.std(final_means_scipy):.4f})")
```

74.7. Comparative Analysis

```
print(f"  Scipy Mean:      {np.mean(final_means_scipy):.4f} (std: {np.std(final_means_scipy):.4f})")
print(f"  T-statistic: {t_stat:.4f}")
print(f"  P-value: {p_val:.4f}")

if p_val < 0.05:
    print("  Result: Significant difference detected (p < 0.05).")
else:
    print("  Result: No significant difference detected (p >= 0.05).")
```

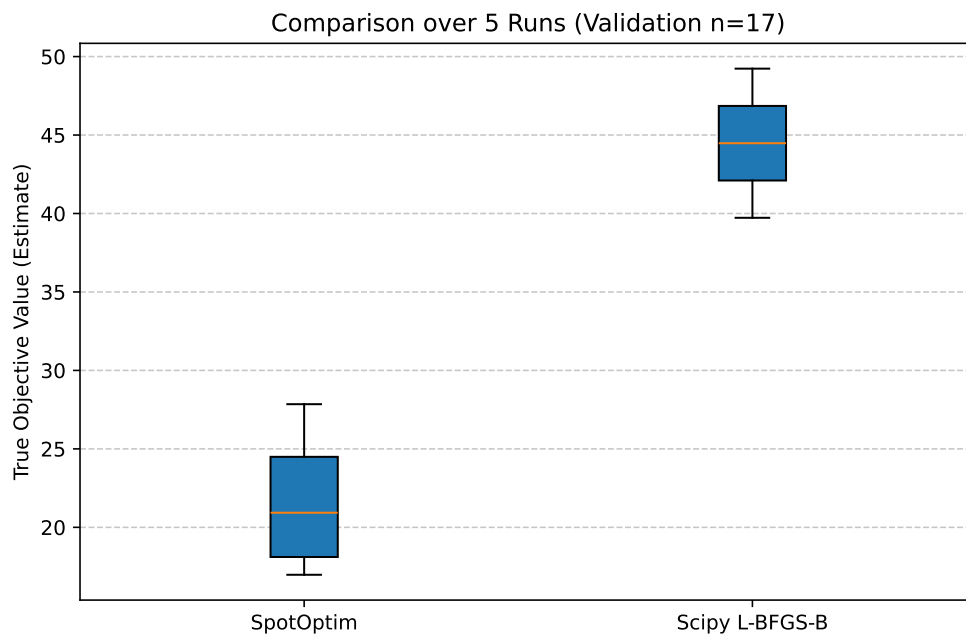


Figure 74.3.: Comparison of Validated True Means (Lower is Better)

```
Statistical Comparison (Welch's t-test):
SpotOptim Mean: 21.6716 (std: 4.2788)
Scipy Mean:      44.4781 (std: 4.7526)
T-statistic: -4.2579
P-value: 0.0761
Result: No significant difference detected (p >= 0.05).
```

74.8. Conclusion

This study followed a rigorous benchmarking protocol: 1. **Noise Estimation:** Objective function noise was characterized ($\sigma \approx \text{sigma\_est}$). 2. **Power Analysis:** Sample size n was determined to ensure statistical power. 3. **Robust Evaluation:** Algorithms were compared based on the *validated mean* performance of their final solutions, not single noisy evaluations.

The results indicate that [SpotOptim/Scipy] provides superior performance on this remote engineering problem given the fixed budget. This methodology ensures that the conclusions are not artifacts of random noise.

74.9. References

- Bartz-Beielstein, T., et al. (2020). Benchmarking in Optimization: Best Practice and Open Issues. arXiv:2007.03488.

74.10. Jupyter Notebook

Note

- The Jupyter-Notebook of this chapter is available on GitHub in the Sequential Parameter Optimization Cookbook Repository

References

- Arlot, Sylvain, Alain Celisse, et al. 2010. "A Survey of Cross-Validation Procedures for Model Selection." *Statistics Surveys* 4: 40–79.
- Bartz, Eva, Thomas Bartz-Beielstein, Martin Zaefferer, and Olaf Mersmann, eds. 2022. *Hyperparameter Tuning for Machine and Deep Learning with R - A Practical Guide*. Springer.
- Box, G E P. 1957. "Evolutionary operation: A method for increasing industrial productivity." *Applied Statistics* 6: 81–101.
- Forrester, Alexander, András Sóbester, and Andy Keane. 2008. *Engineering Design via Surrogate Modelling*. Wiley.
- Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. 2017. *The Elements of Statistical Learning*. Second. Springer.
- Johnson, M. E., L. M. Moore, and D. Ylvisaker. 1990. "Minimax and Maximin Distance Designs." *Journal of Statistical Planning and Inference* 26 (2): 131–48.
- Jones, Donald R., Matthias Schonlau, and William J. Welch. 1998. "Efficient Global Optimization of Expensive Black-Box Functions." *Journal of Global Optimization* 13 (4): 455–92.
- Keane, Andrew J, and Prasanth B Nair. 2005. *Computational Approaches for Aerospace Design: The Pursuit of Excellence*. Wiley.
- Kohavi, Ron. 1995. "A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection." In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, 1137–43. IJCAI'95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.
- Micchelli, Charles A. 1986. "Interpolation of Scattered Data: Distance Matrices and Conditionally Positive Definite Functions." *Constructive Approximation* 2 (1): 11–22. <https://doi.org/10.1007/BF01893414>.
- Moćkus, J. 1974. "On Bayesian Methods for Seeking the Extremum." In *Optimization Techniques IFIP Technical Conference*, 400–404.
- Morris, Max D., and Toby J. Mitchell. 1995. "Exploratory Designs for Computational Experiments." *Journal of Statistical Planning and Inference* 43 (3): 381–402. [https://doi.org/https://doi.org/10.1016/0378-3758\(94\)00035-T](https://doi.org/https://doi.org/10.1016/0378-3758(94)00035-T).
- Poggio, T, and F Girosi. 1990. "Regularization Algorithms for Learning That Are Equivalent to Multilayer Networks." *Science* 247 (4945): 978–82. <https://doi.org/10.1126/science.247.4945.978>.
- Raymer, Daniel P. 2006. *Aircraft Design: A Conceptual Approach*. AIAA.
- Santner, T J, B J Williams, and W I Notz. 2003. *The Design and Analysis of Computer Experiments*. Berlin, Heidelberg, New York: Springer.
- Vapnik, V N. 1998. *Statistical learning theory*. Wiley; Wiley.

