Sequential Parameter Optimization Cookbook

A guide for scikit-learn and PyTorch

Thomas Bartz-Beielstein

Nov 16, 2025

Table of contents

Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

Part I

Optimization

1 Aircraft Wing Weight Example

> **Note**
>
> - This section is based on chapter 1.3 “A ten-variable weight function” in (**Forr08a?**).
> - The following Python packages are imported:

```
import math
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.axes_grid1 import ImageGrid
```

1.1 AWWE Equation

- Example from (**Forr08a?**)
- Understand the **weight** of an unpainted light aircraft wing as a function of nine design and operational parameters:

$$W = 0.036 S_W^{0.758} \times W_{fw}^{0.0035} \left(\frac{A}{\cos^2 \Lambda} \right)^{0.6} \times q^{0.006} \times \lambda^{0.04} \times \left(\frac{100 R_{tc}}{\cos \Lambda} \right)^{-0.3} \times (N_z W_{dg})^{0.49}$$

1.2 AWWE Parameters and Equations (Part 1)

Table 1.1: Aircraft Wing Weight Parameters

Symbol	Parameter	Baseline	Minimum	Maximum
S_W	Wing area (ft^2)	174	150	200
W_{fw}	Weight of fuel in wing (lb)	252	220	300

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/ftˆ2)	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 (**raym06a?**). 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.

1.3 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 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 **??**. To map values from the interval $[a, b]$ to the interval $[0, 1]$, the following formula can be employed:

$$y = f(x) = \frac{x - a}{b - a}. \tag{1.1}$$

To reverse this mapping and obtain the original values, the formula

$$g(y) = a + (b - a)y \tag{1.2}$$

can be used. The function `wingwt()` expects inputs from the unit cube, which are then transformed back to their original scales using Equation **??**. 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)
```

1.4 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))]
```

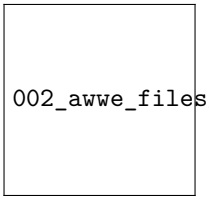
The coding used to transform inputs from natural units is largely a matter of taste, so long as it's easy to undo for reporting back on original scales

```
%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)
```

1.5 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.

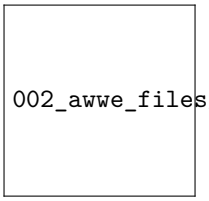
```
z = wingwt(A = X, Nz = Y)
fig = plt.figure(figsize=(7., 5.))
plt.contourf(X, Y, z, 20, cmap='jet')
plt.xlabel("A")
plt.ylabel("Nz")
plt.title("Load factor (Nz) vs. Aspect Ratio (A)")
plt.colorbar()
plt.show()
```



002_awwe_files/figure-pdf/cell-6-output-1.pdf

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)
plt.xlabel("A")
plt.ylabel("Nz")

plt.imshow(z, extent=[0, 1, 0, 1], origin='lower',
           cmap='jet', alpha=0.9)
plt.colorbar()
```



002_awwe_files/figure-pdf/cell-7-output-1.pdf

The interpretation of the AWWE plot can be summarized as follows:

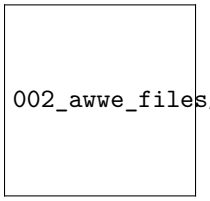
- The figure displays the weight response as a function of two variables, N_z and A, using an image-contour plot.
- The slight curvature observed in the contours suggests an interaction between these two variables.
- Notably, the range of outputs depicted in the figure, spanning from approximately 160 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.
- 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.

1.6 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:
    - 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 $\lambda \times W_{fw}$

1.7 The Big Picture: Combining all Variables

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

```
Z = []
Zlab = []
l = len(pl)
# lc = math.comb(l,2)
for i in range(l):
    for j in range(i+1, l):
    # for j in range(l):
        # print(pl[i], pl[j])
        d = {pl[i]: X, pl[j]: Y}
        Z.append(wingwt(**d))
        Zlab.append([pl[i],pl[j]])
```

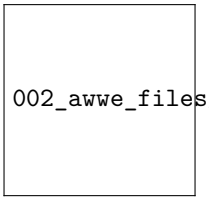
Now we can generate all 36 combinations, e.g., our first example is combination `p = 19`.

```
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 comparibility

```
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()
```



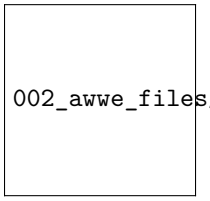
- Let's plot the second example, taper ratio λ and fuel weight W_{fw}

- This is combination 11:

```
p = 11
Zlab[p]
```

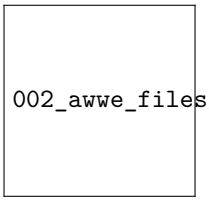
```
['Wfw', 'l']
```

```
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()
```



- 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()
```



1.8 AWWE 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 reaon why highly manoeuvrable fighter jets cannot have very efficient, glider wings)
  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 weightW_{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.

1.9 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

1.10 Exercise

1.10.1 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.

$$W = 0.036 S_W^{0.758} \times W_{fw}^{0.0035} \times \left(\frac{A}{\cos^2 \Lambda} \right)^{0.6} \times q^{0.006} \times \lambda^{0.04}$$
$$\times \left(\frac{100 R_{tc}}{\cos \Lambda} \right)^{-0.3} \times (N_z W_{dg})^{0.49} + S_w W_p$$

1.11 Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

Part II

Numerical Methods

2 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

2.1 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

2.1.1 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.

2.1.2 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

2.1.3 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**

2.1.4 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

2.1.5 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–Voltera) 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

2.1.6 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

2.1.7 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

2.2 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.

2.3 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 **??**.

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 (**Sant03a?**) and (**Forr08a?**). The former is the canonical reference in the statistics literature and the latter is perhaps more popular in engineering.

Example 2.1 (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.

2.3.1 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.

2.4 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).

- 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 2.1 (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.

3 Sampling Plans

Note

- This section is based on chapter 1 in (**Forr08a?**).
- 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
```

3.1 Ideas and Concepts

Definition 3.1 (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

$$X = \left\{ x^{(i)} \in D | i = 1, \ldots, n \right\}$$

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.

3.1.1 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 3.1 (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.

3.1.2 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.

3.1.3 Designing Preliminary Experiments (Screening)

Minimizing the number of design variables $x_1, x_2, \ldots, x_k$ 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 $\frac{\partial f}{\partial x_i} = g(x_i, x_j, \ldots), /, \forall x \in D$, where $g(x_i, x_j, \ldots)$ 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.

3.1.3.1 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,

$$x_i \in \{0, \frac{1}{p-1}, \frac{2}{p-1}, \ldots, 1\}, \quad \text{for } i = 1, \ldots, k.$$

Definition 3.2 (Elementary Effect)**.** For a given baseline value $x \in D$, let $d_i(x)$ denote the elementary effect of x_i, where:

$$d_i(x) = \frac{f(x_1, \ldots, x_i + \Delta, \ldots, x_k) - f(x_1, \ldots, x_i - \Delta, \ldots, x_k)}{2\Delta}, \quad i = 1, \ldots, k, \quad (3.1)$$

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

$$\Delta = \frac{\xi}{p-1}, \quad \xi \in \mathbb{N}^*, \quad \text{and} \quad x \in D, \text{ such that its components } x_i \leq 1 - \Delta.$$

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^*:

$$B^* = \left(1_{k+1,k} x^* + (\Delta/2)\left[(2B - 1_{k+1,k})D^* + 1_{k+1,k}\right]\right) P^*,$$

where:

- D^* is a k-dimensional diagonal matrix with diagonal elements ± 1 (equal probability),

- **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 = 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 3.2 (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}")
```

```
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 **??**.

```
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()
```

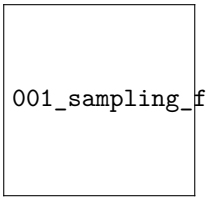


Figure 3.1: Random orientation of the sampling matrix in 2-D. The labels indicate the row index of the points.

Example 3.3 (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:
[[0.  0.  0.5]
 [0.  0.  0. ]
 [0.  0.5 0. ]
 [0.5 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 **??** 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 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.
        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()
```

3.1.4 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 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.

3.2 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 **??**.

```
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,
)
```

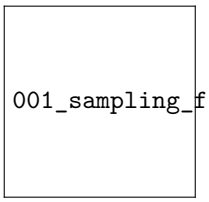


Figure 3.2: Estimated means and standard deviations of the elementary effects for the 10 design variables of the wing weight function. Example based on (**Forr08a?**).

> **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 **??** 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:

  - 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.

3.3 Designing a Sampling Plan

3.3.1 Stratification

A feature shared by all of the approximation models discussed in (**Forr08a?**) 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 $n = q_1 \times q_2 \times \ldots \times q_k$ 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]*q[1]*...*q[k-1] bins filling the un

    Returns:
        (np.ndarray): Full factorial sampling plan as an array of shape (n, k), where n is the total

    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 elemer
    # 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 ra
        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)) * val
        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 **??** shows the points in the unit hypercube for the case of 3x2 points.

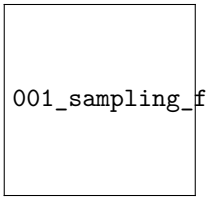


Figure 3.3: 2D Full Factorial Sampling (3x2 Points). Edges = 0

```
X = fullfactorial(q, Edges=1)
print(X)
```

```
[[0.  0. ]
 [0.  1. ]
 [0.5 0. ]
 [0.5 1. ]
 [1.  0. ]
 [1.  1. ]]
```

Figure **??** shows the points in the unit hypercube for the case of 3x2 points with edges.

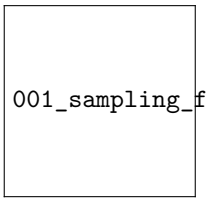


Figure 3.4: 2D Full Factorial Sampling (3x2 Points). Edges = 1

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., $n = q_1 \times q_2 \times \cdots \times q_k$.
- 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.

3.3.2 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 3.3 (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 $\{1, 2, \ldots, n\}$, ensuring each number appears only once per row and column.

Example 3.4 (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:

- 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 $\{1, 2, \ldots, n\}$.
- 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 3.5 (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)
```

```
[[0.7 0.1]
 [0.9 0.9]
 [0.5 0.5]
 [0.1 0.7]
 [0.3 0.3]]
```

Figure **??** shows the points in the unit hypercube for the case of 5 points with `edges=0`.

Example 3.6 (Random Latin Hypercube with Edges)**.** The following code can be used to generate a 2D Latin hypercube with 5 points and edges=1:

```
X = rlh(n=5, k=2, edges=1)
print(X)
```

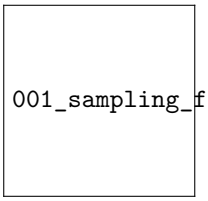


Figure 3.5: 2D Latin Hypercube Sampling (5 Points, Edges=0)

```
[[1.   0.75]
 [0.75 0.5 ]
 [0.   0.25]
 [0.25 1.  ]
 [0.5  0.  ]]
```

Figure **??** shows the points in the unit hypercube for the case of 5 points with `edges=1`.

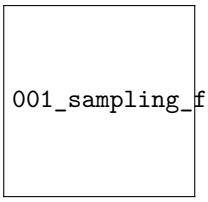


Figure 3.6: 2D Latin Hypercube Sampling (5 Points, Edges=1)

3.3.3 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 (**john90a?**). This criterion can be formally defined as follows.

Consider a sampling plan X. Let $d_1, d_2, \ldots, d_m$ represent the unique distances between all possible pairs of points within X, arranged in ascending order. Furthermore, let $J_1, J_2, \ldots, J_m$ be defined such that J_j denotes the count of point pairs in X separated by the distance d_j.

Definition 3.4 (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 **??** may identify multiple equivalent maximin designs. To address this, a more comprehensive 'tie-breaker' definition, as proposed by (**morr95a?**), is employed:

Definition 3.5 (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.

(**john90a?**) established that the maximin criterion (Definition **??**) is equivalent to the D-optimality criterion used in linear regression. However, the extended maximin criterion incorporating a tie-breaker (Definition **??**) is often 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 **??**) 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 3.6 (p-norm)**.** The p-norm of a vector $\vec{x} = (x_1, x_2, \ldots, x_k)$ is defined as:

$$d_p(\vec{x}^{(i_1)}, \vec{x}^{(i_2)}) = \left(\sum_{j=1}^{k} |x_j^{(i_1)} - x_j^{(i_2)}|^p \right)^{1/p}. \tag{3.2}$$

When $p = 1$, Equation **??** 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 **??**, the initial step involves constructing the vectors $d_1, d_2, \ldots, d_m$ and $J_1, J_2, \ldots, J_m$. The `jd` function facilitates this task.

3.3.3.1 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.

```python
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 3.7 (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 **??**. The distinct distances

and their occurrences can be computed using the `jd` function, as shown in the following code:

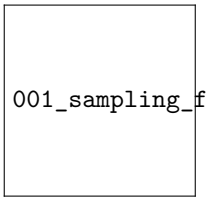


Figure 3.7: 3-Point Set in 2D Space

```
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]
```

3.3.4 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 3.7 (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 **??** 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.

3.3.4.1 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 3.8 (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 **??**) 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]])
```

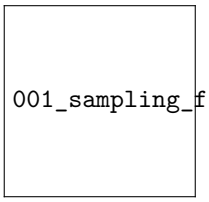


Figure 3.8: Comparison of Two Sampling Plans

We can compare which plan has better space-filling (Morris-Mitchell). The output is 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.
```

3.3.4.2 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 (**morr95a?**) 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 $(d_1, d_2, ..., d_m)$ and multiplicity $(J_1, J_2, ..., J_m)$ arrays are disregarded. These disregarded values, however, might contain potentially useful 'slope' information about the global landscape for the optimization process.

To address this, (**morr95a?**) defined the following scalar-valued criterion function, which is used to rank competing sampling plans. This function, while based on the logic of Definition **??**, incorporates the complete vectors $d_1, d_2, ..., d_m$ and $J_1, J_2, ..., J_m$.

Definition 3.8 (Morris-Mitchell Criterion)**.** The Morris-Mitchell criterion is defined as:

$$\Phi_q(X) = \left(\sum_{j=1}^{m} J_j d_j^{-q} \right)^{1/q}, \tag{3.3}$$

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 Phiq. 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.
    Phiq = np.sum(J * (d ** (-q))) ** (1.0 / q)
    return Phiq
```

Example 3.9 (The Function `mmphi`)**.** We can use the `mmphi` function to evaluate the space-filling quality of the two sampling plans from Example **??**. 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 **??**). 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 **??** and the maximin criterion of Definition **??**, sets of 50 random Latin hypercubes of varying sizes and dimensionalities were considered by (**Forr08a?**). 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 **??**.

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`.

3.3.4.3 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.

```python
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 3.10 (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 X1 and X1 from above. They are sorted using the Morris-Mitchell criterion with $p = 2.0$. For example, the output `[1, 2]` indicates that X1 is more space-filling than X2:

```
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 (**morr95a?**) 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, X3D, 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 **??**, using the rectangular distance metric. The subsequent discussion will address the methods for finding these optimized Φ_q designs.

3.3.4.4 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 3.11 (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))
```

```
[2 1]
[1 2]
```

3.3.5 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.

(**Forr08a?**) 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:

3.3.5.1 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:
            el1 = int(np.floor(np.random.rand() * n))
            el2 = int(np.floor(np.random.rand() * n))
        # Swap the two selected elements in the chosen column
        X[el1, col], X[el2, col] = X[el2, col], X[el1, col]
    return X
```

Example 3.12 (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:
[[2 3]
 [1 4]
 [3 1]
 [4 2]]
```

(**Forr08a?**) uses the term 'mutation', because this problem lends itself to nature-inspired computation. (**morr95a?**) 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 (**Forr08a?**).

3.3.6 Evolutionary Operation

As introduced by (**Box57a?**), 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 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.

3.3.7 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.

3.3.7.1 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 3.13 (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 **??** shows the initial and optimized plans in 2D. The blue points represent the initial plan, while the red points represent the optimized plan.

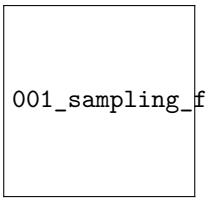


Figure 3.9: Comparison of the initial and optimized plans in 2D.

3.3.7.2 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 3.14 (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 **??** shows the best Latin hypercube sampling in 2D. The red points represent

the optimized plan.

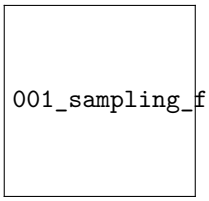


Figure 3.10: Best Latin Hypercube Sampling

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 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.

3.4 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
```

3.4.1 Evaluation of Sampling Designs

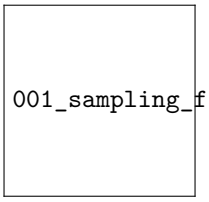
We generate various sampling designs and evaluate their space-filling properties using the Morris-Mitchell criterion.

```
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.sq
else:
    print(f"Skipping grid design as N_POINTS={N_POINTS} is not a perfect square for a

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)

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_point

results = {}

print("Calculating Morris-Mitchell metric (smaller is better):")
for name, X_design in designs.items():
    metric_val = mmphi(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: 28.8942
  Random: 46.7319
  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}\nmmphi = {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])

    plt.tight_layout()
    plt.suptitle(f"Comparison of 2D Sampling Designs ({N_POINTS} points each)", fontsize=14, y=1.02)
    plt.show()
```



3.4.2 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 dista

# 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
```

```
  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)
```

3.4.3 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.]
 [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)."
    },
```

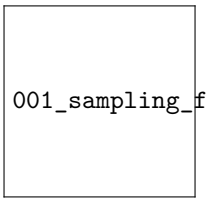
```
    "Scenario 3: Add on Edge": {
        "new_point": np.array([[0.5, 0.0]]),
        "description": "Adding a point on an edge between (0,0) and (1,0)."
    }
}

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 matrix
    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 (Phi_q): 3.6056
  Change from Initial Phi_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 (Phi_q): 7.6195
  Change from Initial Phi_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 (Phi_q): 3.8210
  Change from Initial Phi_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='
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', al
    ax.scatter(new_point[:, 0], new_point[:, 1], s=150, color='red', edgecolors='k', m

    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 suptitle
plt.suptitle(f"Impact of Adding a Point to a 2x2 Grid Design (q={q}, p={p})", fontsize=14)
plt.show()
```



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

3.5 A Sample-Size Invariant Version of the Morris-Mitchell Criterion

3.5.1 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 avaiable 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 $\{x_1, x_2, \ldots, x_n\}$ 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:

$$\Phi_q^I(X) = \left(\frac{1}{M} \sum_{l=1}^{m} J_l d_l^{-q} \right)^{1/q}$$

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.

3.5.2 Plotting the Two Morris-Mitchell Criteria for Different Sample Sizes

Figure **??** 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='m
    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
    ax2.plot(n_values, mmphi_intensive_results, color=color, marker='x', linestyle='--
    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=
    # 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...
```

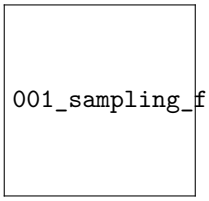


Figure 3.11: Comparison of the two Morris-Mitchell Criteria for Different Sample Sizes

3.6 Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

4 Constructing a Surrogate

> **Note**
>
> This section is based on chapter 2 in (**Forr08a?**).

Definition 4.1 (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 4.2 (Generic Solution)**.** The generic solution method is to collect the output values $y^{(1)}$, $y^{(2)}$, ..., $y^{(n)}$ that result from a set of inputs $\vec{x}^{(1)}$, $\vec{x}^{(2)}$, ..., $\vec{x}^{(n)}$ and find a best guess $\hat{f}(\vec{x})$ for the black box mapping f, based on these known observations.

4.1 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 $\vec{x} = \{x_1, x_2, \ldots, x_k\}^T$ 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

$$X = \{\vec{x}^{(1)}, \vec{x}^{(2)}, \ldots, \vec{x}^{(n)}\}^T,$$

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.

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

$$\{(\vec{x}^{(1)}, y^{(1)}), (\vec{x}^{(2)}, y^{(2)}), \ldots, (\vec{x}^{(n)}, y^{(n)})\}.$$

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 4.1 (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,

$$f(x, \vec{w}) = \vec{w}^T \vec{x} + v. \tag{4.1}$$

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.

Definition 4.3 (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 **??**, 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 4.2 (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.

4.2 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 4.4 (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

$$\{(\vec{x}^{(1)}, y^{(1)} \pm \epsilon), (\vec{x}^{(2)}, y^{(2)} \pm \epsilon), \ldots, (\vec{x}^{(n)}, y^{(n)} \pm \epsilon)\}$$

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.

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:

$$P = \frac{1}{(2\pi\sigma^2)^{n/2}} \exp\left[-\frac{1}{2\sigma^2}\sum_{i=1}^{n}\left(y^{(i)} - f(\vec{x}^{(i)}, \vec{w})\right)^2 \epsilon\right].$$

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:

$$\min_{\vec{w}} \sum_{i=1}^{n} \frac{[y^{(i)} - f(\vec{x}^{(i)}, \vec{w})]^2}{2\sigma^2} + \frac{n}{2}\ln\epsilon. \tag{4.2}$$

If we assume σ and ϵ are constants, Equation **??** 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 4.5 (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 $\theta : \{1, \ldots, n\} \to \{1, \ldots, q\}$ 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:

$$CV = \frac{1}{n}\sum_{i=1}^{n} L(y^{(i)}, f^{(-\theta(i))}(\vec{x}^{(i)})). \tag{4.3}$$

Introducing the squared error as the loss function and considering our generic model f still dependent on undetermined parameters, we write Equation **??** as:

$$CV = \frac{1}{n}\sum_{i=1}^{n}[y^{(i)} - f^{(-\theta(i))}(\vec{x}^{(i)})]^2. \tag{4.4}$$

The extent to which Equation **??** 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. (**Hast17a?**)) 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 (**arlot2010?**) and (**Koha95a?**).

4.3 Stage Three: Model Testing

If there is a sufficient amount of observational data, a random subset should be set aside initially for model testing. (**Hast17a?**) recommend setting aside approximately $0.25n$ of $\vec{x} \to y$ 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 (**Forr08a?**)), 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 4.6 (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:

$$r^2 = \left(\frac{\mathrm{cov}(y, \hat{y})}{\sqrt{\mathrm{var}(y)\mathrm{var}(\hat{y})}} \right)^2, \tag{4.5}$$

Equation **??** 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 4.3 (The Tea and Sugar Analogy)**.** (**Forr08a?**) 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.

4.4 Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

5 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

5.1 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.

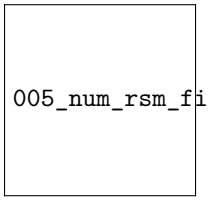
RSM is related to various fields, including Design of Experiments (DoE), quality management, reliability, and productivity. Its applications are widespread in industry and manufacturing, focusing on designing, developing, and formulating new products and improving existing ones, as well as from laboratory 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.

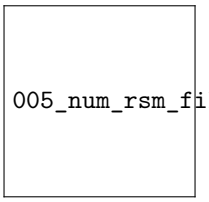
```
import numpy as np
import matplotlib.pyplot as plt

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 $(\xi_1.\xi_2)$

5.1.1 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

5.1.2 RSM: Strategies

- RSMs consist of experimental strategies for
- **exploring** the space of the process (i.e., independent/input) variables (above ξ_1 and $\xi 2$)

- 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 $\xi_1, \xi_2, \ldots, \xi_m$
- RSM: Equations of the Empirical Model
  - $Y = f(\xi_1, \xi_2, \ldots, \xi_m) + \epsilon$
  - $\mathbb{E}\{Y\} = \eta = f(\xi 1_1, \xi_2, \ldots, \xi_m)$
  - 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

5.1.3 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

5.1.4 RSM: Natural and Coded Variables

- Inputs $\xi_1, \xi_2, \ldots, \xi_m$ called **natural variables**:
  - expressed in natural units of measurement, e.g., degrees Celsius, pounds per square inch (psi), etc.
- Transformed to **coded variables** $x_1, x_2, \ldots, x_m$:
  - to mitigate hassles and confusion that can arise when working with a multitude of scales of measurement
- Typical **Transformations** offering dimensionless inputs $x_1, x_2, \ldots, x_m$
  - in the unit cube, or

  - scaled to have a mean of zero and standard deviation of one, are common choices.
- Empirical model becomes $\eta = f(x_1, x_2, \ldots, x_m)$

5.1.5 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**

5.2 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:

$$\eta = 50 + 8x_1 + 3x_2$$

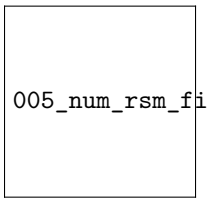
- In practice, such a surface would be obtained by fitting a model to the outcome of a designed experiment
- First-Order Model in python Evaluated on a Grid
- Evaluate model on a grid in a double-unit square centered at the origin
- Coded units are chosen arbitrarily, although one can imagine deploying 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: $50 + 8x_1 + 3x_2$')
```

```
Text(0.5, 1.0, 'First Order Model: $50 + 8x_1 + 3x_2$')
```



5.2.1 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 $\eta = 50 + 8x_1 + 3x_2 - 4x_1 x_2$

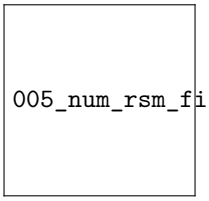
5.2.2 First-order Model with Interactions in python

- Code below facilitates evaluations for pairs (x_1, x_2)
- Responses may be observed over a mesh in the same double-unit square

```
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')
```



5.2.3 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

5.3 Second-Order Models

- Second-order model may be appropriate near local optima where f would have substantial curvature:

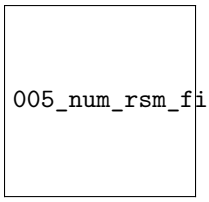
$$\eta = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_{11} x_1^2 + \beta_{22} x^2 + \beta_{12} x_1 x_2$$

- 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 $(0.6,0.2)$')
```



5.3.1 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

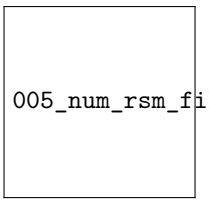
5.3.2 Example: Stationary Ridge

- 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')
```



5.3.3 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

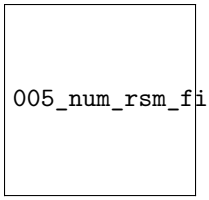
5.3.4 Example: Rising Ridge

- 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
```



5.3.5 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.

5.3.6 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.

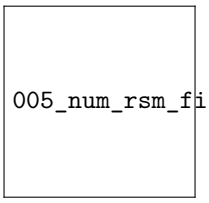
5.3.7 Saddle Point

- Finally, we can get what's called a saddle or minimax system.

```
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: $\\eta = 80 + 4x_1 + 8x_2 - 2x_2^2 - 12x_1x_2$')
```



5.3.8 Interpretation: Saddle Points

- Likely further data collection, and/or outside expertise, is needed before determining a course of action in this situation

5.3.9 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

5.4 General RSM Models

- General **first-order model** on m process variables $x_1, x_2, \cdots, x_m$ is

$$\eta = \beta_0 + \beta_1 x_1 + \cdots + \beta_m x_m$$

- General **second-order model** on m process variables

$$\eta = \beta_0 + \sum_{j=1}^{m} + \sum_{j=1}^{m} x_j^2 + \sum_{j=2}^{m} \sum_{k=1}^{j} \beta_{kj} x_k x_j.$$

5.4.1 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

5.5 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

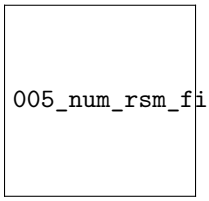
$$b = (X'X)^{-1}X'Y.$$

Example 5.1 (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 1ßß 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]]
```



5.6 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
- 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

5.6.1 Different Designs

- **Screening desings**: 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

5.7 RSM Experimentation

5.7.1 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

5.7.2 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

5.7.3 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

5.8 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

5.8.1 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

5.8.2 Status Quo

- Nowadays, field experiments and statistical models, designs and optimizations are coupled with 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

5.8.3 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

5.8.4 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

5.9 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.

5.10 Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

6 Polynomial Models

> **Note**
>
> - This section is based on chapter 2.2 in (**Forr08a?**).
> - The following Python packages are imported:

```
import numpy as np
import matplotlib.pyplot as plt
```

6.1 Fitting a Polynomial

We will consider one-variable cases, i.e., $k = 1$, first.

Let us consider the scalar-valued function $f : \mathbb{R} \to \mathbb{R}$ observed according to the sampling plan $X = \{x^{(1)}, x^{(2)} \ldots, x^{(n)}\}^T$, yielding the responses $\vec{y} = \{y^{(1)}, y^{(2)}, \ldots, y^{(n)}\}^T$.

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 $w = w_0, w_1, \ldots, w_m{}^T$ through a least squares solution of:

$$\Phi \vec{w} = \vec{y}$$

where Φ is the Vandermonde matrix:

$$\Phi = \begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^m \\ 1 & x_2 & x_2^2 & \ldots & x_2^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^m \end{bmatrix}.$$

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 **??**).

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., $m \in 1, 2, \ldots, 15$).

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.

6.2 Polynomial Fitting in Python

6.2.1 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
```

6.2.2 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`:

```python
# 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]
```

6.2.3 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:

$$C_D(x) = w_8 x^8 + w_7 x^7 + \cdots + w_1 x + w_0,$$

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
```

6.2.4 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="r
    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()
```

6.3 Example One: Aerofoil Drag

The circles in Figure **??** 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 (**Forr08a?**)). The members of the family have different shapes, as determined by the sampling plan:

$$X = x_1, x_2, \ldots, x_{101}$$

The responses are:

$$C_D = \{C_D^{(1)}, C_D^{(2)}, \ldots, C_D^{(101)}\}$$

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 untion 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
```

```
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)
```

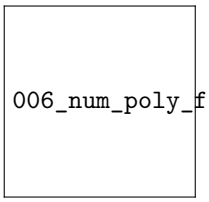


Figure 6.1: Aerofoil drag data

Figure **??** 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:

```
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))
```

6.4 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)
```

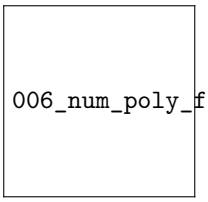


Figure 6.2: Onevar function

This function, depicted by the dotted line in Figure **??**, 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 **??**) by adding normally distributed noise to the function. Figure **??** shows a seventh-order polynomial fitted through the noisy data. This polynomial was selected as it minimizes the cross-validation metric.

6.5 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:

$$\hat{f}(\vec{x}) = \sum_{i=1}^{N} w_i \phi_i(\vec{x}),$$

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:

$$\text{Constant term: } 1 \tag{6.1}$$

$$\text{First-order terms: } x_1, x_2, x_3 \tag{6.2}$$

$$\text{Second-order terms: } x_1^2, x_2^2, x_3^2, x_1x_2, x_1x_3, x_2x_3 \tag{6.3}$$

$$\text{Third-order terms: } x_1^3, x_2^3, x_3^3, x_1^2x_2, x_1^2x_3, x_2^2x_1, x_2^2x_3, x_3^2x_1, x_3^2x_2, x_1x_2x_3 \tag{6.4}$$

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.

6.6 Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

7 Radial Basis Function Models

> **Note**
>
> - This section is based on chapter 2.3 in (**Forr08a?**).
> - The following Python packages are imported:

```
import numpy as np
import matplotlib.pyplot as plt
```

7.1 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 $x_1 x_2$. 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 (**Forr08a?**) 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 7.1 (Euclidean Distance)**.** The Euclidean distance between two points in a k-dimensional space is defined as:

$$\|\vec{x} - \vec{c}\| = \sqrt{\sum_{i=1}^{k}(x_i - c_i)^2}, \tag{7.1}$$

where:

- $\vec{x} = (x_1, x_2, \ldots, x_d)$ is the first point,
- $\vec{c} = (c_1, c_2, \ldots, c_d)$ 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 7.2 (Radial Basis Function (RBF))**.** Mathematically, a radial basis function ψ can be expressed as:

$$\psi(\vec{x}) = \psi(\|\vec{x} - \vec{c}\|), \tag{7.2}$$

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.

7.1.1 Fitting Noise-Free Data

Let us consider the scalar valued function f observed without error, according to the sampling plan $X = \{\vec{x}^{(1)}, \vec{x}^{(2)}, \ldots, \vec{x}^{(n)}\}^T$, yielding the responses $\vec{y} = \{y^{(1)}, y^{(2)}, \ldots, y^{(n)}\}^T$.

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:

$$\hat{f}(\vec{x}) = \sum_{i=1}^{n_c} w_i \psi(||\vec{x} - \vec{c}^{(i)}||), \tag{7.3}$$

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 **??**.

7.1.1.1 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 7.1 (Gaussian RBF)**.** Using the general definition of a radial basis function (Equation **??**), we can express the Gaussian RBF as:

$$\psi(\vec{x}) = \exp\left(-\frac{\|\vec{x} - \vec{c}\|^2}{2\sigma^2}\right) = \exp\left(-\frac{\sum_{j=1}^{k}(x_j - c_j)^2}{2\sigma^2}\right) \tag{7.4}$$

where:

- $\vec{x}$ 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 **??** 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.

```python
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))
```

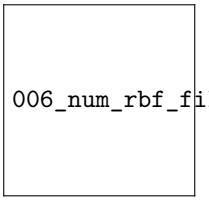


Figure 7.1: Gaussian RBF

The sum of Gaussian RBFs can be visualized by summing the individual Gaussian RBFs centered at different points as shown in Figure **??**. 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.

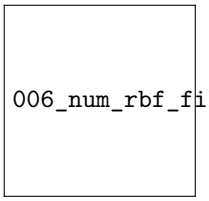


Figure 7.2: Sum of Gaussian RBFs

7.1.1.2 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:

$$\hat{f}(\vec{x}^{(i)}) = y^{(i)}, \quad i = 1, 2, \ldots, n. \tag{7.5}$$

This constraint produces one of the most powerful aspects of RBF modeling: while the system in Equation **??** is linear with respect to the weights $\vec{w}$, the resulting predictor $\hat{f}$ 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 $i = 1, 2, \ldots, n$. 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:

$$\Psi_{i,j} = \psi(||\vec{x}^{(i)} - \vec{x}^{(j)}||), \quad i, j = 1, 2, \ldots, n.$$

The solution for the weight vector becomes:

$$\vec{w} = \Psi^{-1} \vec{y}.$$

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.

7.1.2 Numerical Stability Through Positive Definite Matrices

A significant advantage of Gaussian and inverse multiquadric basis functions lies in their mathematical guarantees. (**vapn98a?**) demonstrated that these functions always produce symmetric positive definite Gram matrices when using strictly positive definite kernels (see Section **??**), 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
```

7.1.3 Ill-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 (**micc86a?**). 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 (**kean05a?**). 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).

7.1.4 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):

7.1.4.1 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)

7.1.4.2 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.

7.2 Python Implementation of the RBF Model

Section **??** 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.

7.2.1 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(re

                    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
```

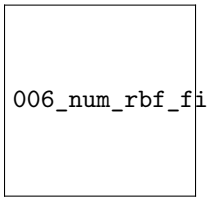
```
        d = np.zeros(len(self.X))
        for k in range(len(self.X)):
            d[k] = np.linalg.norm(x - self.X[k])

        # 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
```

7.3 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.estim_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.wei

# 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
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]
```



7.4 RBF Example: The Two-Diemnsional 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)):
        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', linestyles='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 **??** 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)
```

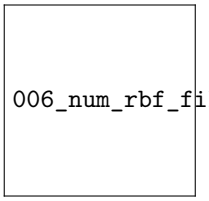


Figure 7.3: RBF Approximation.

7.4.1 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 7.2 (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 neral 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.

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.

7.5 Radial Basis Function Models for Noisy Data

When the responses $\vec{y} = y^{(1)}, y^{(2)}, \ldots, y^{(n)^T}$ 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:

7.5.1 Ridge Regularization Approach

The most straightforward solution involves introducing regularization through the parameter λ (**pogg90a?**). 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 $\vec{y}$ (**kean05a?**). Since this information is rarely available in practice, λ is typically estimated through cross-validation alongside other model parameters.

7.5.2 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.

7.6 Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

8 Kriging (Gaussian Process Regression)

> **Note**
>
> - This section is based on chapter 2.4 in (**Forr08a?**).
> - 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
```

8.1 From Gaussian RBF to Kriging Basis Functions

Kriging can be explained using the concept of radial basis functions (RBFs), which were introduced in Chapter **??**. 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 8.1 (The Kriging Basis Functions)**.** Kriging (also known as Gaussian Process Regression) uses k-dimensional basis functions of the form

$$\psi^{(i)}(\vec{x}) = \psi(\vec{x}^{(i)}, \vec{x}) = \exp\left(-\sum_{j=1}^{k} \theta_j |x_j^{(i)} - x_j|^{p_j}\right), \tag{8.1}$$

where $\vec{x}$ and $\vec{x}^{(i)}$ denote the k-dim vector $\vec{x} = (x_1, \ldots, x_k)^T$ and $\vec{x}^{(i)} = (x_1^{(i)}, \ldots, x_k^{(i)})^T$, respectively.

□

Kriging uses a specialized basis function that offers greater flexibility than standard RBFs. Examining Equation **??**, 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 $\vec{\theta} = (\theta_1, \theta_2, \ldots, \theta_k)^T$. 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 $\vec{p} = (p_1, p_2, \ldots, p_k)^T$ (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.

8.2 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 a 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

$$\begin{pmatrix} Y(\vec{x}^{(1)}) \\ \vdots \\ Y(\vec{x}^{(n)}) \end{pmatrix}. \tag{8.2}$$

The set of random vectors from Equation **??** (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 **??**:

$$\operatorname{cor}\left(Y(\vec{x}^{(i)}), Y(\vec{x}^{(l)})\right) = \exp\left(-\sum_{j=1}^{k} \theta_j |x_j^{(i)} - x_j^{(l)}|^{p_j}\right). \tag{8.3}$$

Using Equation **??**, we can compute the $(n \times n)$ correlation matrix Ψ of the observed sample data as shown in Equation **??**,

$$\Psi = \begin{pmatrix} \operatorname{cor}\left(Y(\vec{x}^{(1)}), Y(\vec{x}^{(1)})\right) & \dots & \operatorname{cor}\left(Y(\vec{x}^{(1)}), Y(\vec{x}^{(n)})\right) \\ \vdots & \vdots & \vdots \\ \operatorname{cor}\left(Y(\vec{x}^{(n)}), Y(\vec{x}^{(1)})\right) & \dots & \operatorname{cor}\left(Y(\vec{x}^{(n)}), Y(\vec{x}^{(n)})\right) \end{pmatrix}, \tag{8.4}$$

and a covariance matrix as shown in Equation **??**,

$$\operatorname{Cov}(Y, Y) = \sigma^2 \Psi. \tag{8.5}$$

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 8.1 (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 **??**. 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 $|x_j^{(i)} - x_j| \to 0$ and $\exp\left(-|x_j^{(i)} - x_j|^{p_j}\right) \to 1$ (these points show very close correlation and $Y(x_j^{(i)}) = Y(x_j)$).
- two points move far apart, then $|x_j^{(i)} - x_j| \to \infty$ and $\exp\left(-|x_j^{(i)} - x_j|^{p_j}\right) \to 0$ (these points show very low correlation).

Example 8.1 (Correlations for different p_j)**.** Three different correlations are shown in Figure **??**: $p_j = 0.1, 1, 2$. The smoothness parameter p_j affects the correlation:

- With $p_j = 0.1$, there is basicaly 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 **??**.

□

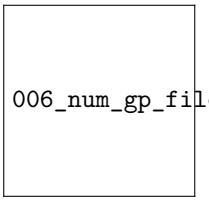


Figure 8.1: Correlations with varying p. θ set to 1.

Example 8.2 (Correlations for different θ)**.** Figure **??** 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:

- 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 **??**.

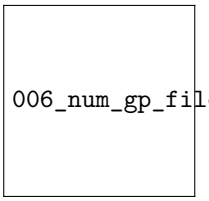


Figure 8.2: Correlations with varying θ. p set to 2.

□

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 $\vec{\theta}$, 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 8.3 (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:

- 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 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:

$$\begin{pmatrix} 0 & \theta_1(x_{11}-x_{21})^2 + \theta_2(x_{12}-x_{22})^2 + \theta_3(x_{13}-x_{23})^2 & \theta_1(x_{11}-x_{31})^2 + \theta_2(x_{12}-x_{32})^2 + \theta_3(x_{13}-x_{33}) \\ 0 & 0 & \theta_1(x_{21}-x_{31})^2 + \theta_2(x_{22}-x_{32})^2 + \theta_3(x_{23}-x_{33}) \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

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 **??**. 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.        ]])
```

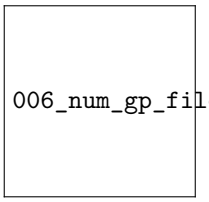


Figure 8.3: Correlation matrix Ψ.

□

Example 8.4 (Example: The Correlation Matrix (Using Existing Functions))**.** The same result as computed in Example **??** can be obtained with existing python functions, e.g., from the package `scipy`.

```
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.        , 0.04978707, 0.        , 0.        ],
       [0.04978707, 1.        , 0.        , 0.        ],
       [0.        , 0.        , 1.        , 0.36787944],
       [0.        , 0.        , 0.36787944, 1.        ]])
```

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 **??**.

```
np.linalg.cond(Psi)
```

```
np.float64(2.163953413738652)
```

□

8.3 MLE to estimate θ and p

8.3.1 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 $\vec{y}$,

$$L = L\left(Y(\vec{x}^{(1)}), \ldots, Y(\vec{x}^{(n)}) | \mu, \sigma\right) = \frac{1}{(2\pi\sigma^2)^{n/2}} \exp\left[-\frac{\sum_{i=1}^{n}(Y(\vec{x}^{(i)}) - \mu)^2}{2\sigma^2}\right], \quad (8.6)$$

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

$$L = \frac{1}{(2\pi\sigma^2)^{n/2}|\vec{\Psi}|^{1/2}} \exp\left[-\frac{(\vec{y} - \vec{1}\mu)^T \vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu)}{2\sigma^2}\right]. \quad (8.7)$$

Remark 8.2. The transition from Equation **??** to Equation **??** 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 **??** 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 $\vec{1}\mu$.

3. **Assuming Independent Observations:** Equation **??** 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 **??** represents the likelihood under a simplified assumption of independent errors, whereas Equation **??** 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 **??**, 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 **??** can be formulated as the log-likelihood:

$$\ln(L) = -\frac{n}{2}\ln(2\pi\sigma) - \frac{1}{2}\ln|\vec{\Psi}| - \frac{(\vec{y} - \vec{1}\mu)^T \vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu)}{2\sigma^2}. \tag{8.8}$$

8.3.2 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 **??**.

Remark 8.3 (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 **??**).
- 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 $\mathbf{A}\mathbf{v} + \mathbf{A}\mathbf{v} = 2\mathbf{A}\mathbf{v}$.

□

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:

$$\frac{\partial}{\partial \mu}\left(-\frac{1}{2\sigma^2}(\vec{y} - \vec{1}\mu)^T\vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu)\right) = -\frac{1}{2\sigma^2}\left(-2\vec{1}^T\vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu)\right) = \frac{1}{\sigma^2}\vec{1}^T\vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu)$$

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:

$$\vec{1}^T\vec{\Psi}^{-1}\vec{y} = \mu\vec{1}^T\vec{\Psi}^{-1}\vec{1}.$$

8.3.3 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:

$$\frac{\partial}{\partial \nu}\left(-\frac{n}{2}\ln(\nu)\right) = -\frac{n}{2}\cdot\frac{1}{\nu} = -\frac{n}{2\sigma^2}.$$

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:

$$-\frac{n}{2\sigma^2} + \frac{(\vec{y} - \vec{1}\mu)^T \vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu)}{2(\sigma^2)^2} = 0.$$

8.3.4 Results of the Optimizations

Optimization of the log-likelihood by taking derivatives with respect to μ and σ results in

$$\hat{\mu} = \frac{\vec{1}^T \vec{\Psi}^{-1} \vec{y}^T}{\vec{1}^T \vec{\Psi}^{-1} \vec{1}^T} \tag{8.9}$$

and

$$\hat{\sigma}^2 = \frac{(\vec{y} - \vec{1}\mu)^T \vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu)}{n}. \tag{8.10}$$

8.3.5 The Concentrated Log-Likelihood Function

Combining the equations, i.e., substituting Equation **??** and Equation **??** into Equation **??** leads to the concentrated log-likelihood function:

$$\ln(L) \approx -\frac{n}{2}\ln(\hat{\sigma}) - \frac{1}{2}\ln|\vec{\Psi}|. \tag{8.11}$$

Remark 8.4 (The Concentrated Log-Likelihood).

- The first term in Equation **??** 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 **??**) is not differentiable.

□

8.3.6 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 **??**, 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.

8.3.7 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.

8.4 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 **??** 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.

8.5 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 (**Forr08a?**) and (**bart21i?**).

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

$$\vec{\psi} = \begin{pmatrix} \operatorname{cor}\left(Y(\vec{x}^{(1)}), Y(\vec{x})\right) \\ \vdots \\ \operatorname{cor}\left(Y(\vec{x}^{(n)}), Y(\vec{x})\right) \end{pmatrix} = \begin{pmatrix} \vec{\psi}^{(1)} \\ \vdots \\ \vec{\psi}^{(n)} \end{pmatrix}.$$

Definition 8.2 (The Augmented Correlation Matrix)**.** The augmented correlation matrix is constructed as

$$\vec{\tilde{\Psi}} = \begin{pmatrix} \vec{\Psi} & \vec{\psi} \\ \vec{\psi}^T & 1 \end{pmatrix}.$$

□

The log-likelihood of the augmented data is

$$\ln(L) = -\frac{n}{2}\ln(2\pi) - \frac{n}{2}\ln(\hat{\sigma}^2) - \frac{1}{2}\ln|\vec{\tilde{\Psi}}| - \frac{(\vec{\tilde{y}} - \vec{1}\hat{\mu})^T \vec{\tilde{\Psi}}^{-1}(\vec{\tilde{y}} - \vec{1}\hat{\mu})}{2\hat{\sigma}^2}, \tag{8.12}$$

where $\vec{1}$ is a vector of ones and $\hat{\mu}$ and $\hat{\sigma}^2$ are the MLEs from Equation **??** and Equation **??**. Only the last term in Equation **??** depends on $\hat{y}$, so we need only consider this term in the maximization. Details can be found in (**Forr08a?**). Finally, the MLE for $\hat{y}$ can be calculated as

$$\hat{y}(\vec{x}) = \hat{\mu} + \vec{\psi}^T \vec{\tilde{\Psi}}^{-1}(\vec{y} - \vec{1}\hat{\mu}). \tag{8.13}$$

Equation **??** 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

$$\vec{w} = \vec{\vec{\Psi}}^{(-1)}(\vec{y} - \vec{1}\hat{\mu}).$$

- Interpolation: The predictions interpolate the sample data. When calculating the prediction at the ith sample point, $\vec{x}^{(i)}$, the ith column of $\vec{\vec{\Psi}}^{-1}$ is $\vec{\psi}$, and $\vec{\psi}\vec{\vec{\Psi}}^{-1}$ is the ith unit vector. Hence,

$$\hat{y}(\vec{x}^{(i)}) = y^{(i)}.$$

8.6 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.

8.6.1 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 **??**.

Table 8.1: 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 **??**.

```
plt.plot(X, y, "bo")
plt.title(f"Sin(x) evaluated at {n} points")
plt.grid()
plt.show()
```

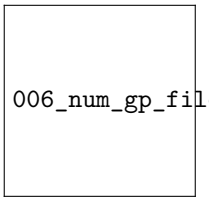


Figure 8.4: Sin(x) evaluated at 8 points.

8.6.2 Computing the Ψ Matrix

We will use the `build_Psi` function from Example **??** 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 **??** visualizes the $(8, 8)$ correlation matrix Ψ.

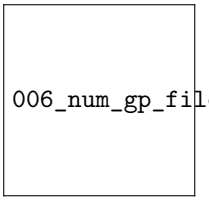


Figure 8.5: Correlation matrix Ψ for the sinusoid function.

8.6.3 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 `x`.

```
m = 100
x = np.linspace(0, 2*np.pi, m, endpoint=False).reshape(-1,1)
```

8.6.4 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 ?? visualizes the $(8, 100)$ prediction matrix ψ.

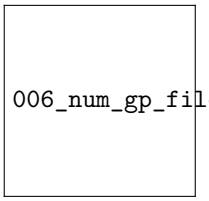


Figure 8.6: Visualization of the predition matrix ψ

8.6.5 Predicting at New Locations

Computation of the predictive equations.

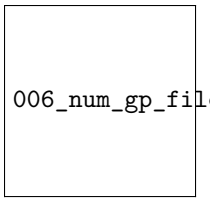
```
U = cholesky(Psi).T
one = np.ones(n).reshape(-1,1)
mu = (one.T.dot(solve(U, solve(U.T, y)))) / one.T.dot(solve(U, solve(U.T, one)))
f = mu * ones(m).reshape(-1,1) + psi.T.dot(solve(U, solve(U.T, y - one * mu)))
print(f"Dimensions of f: {f.shape}")
```

```
Dimensions of f: (100, 1)
```

To compute f, Equation **??** is used.

8.6.6 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()
```



006_num_gp_files/figure-pdf/cell-21-output-1.pdf

8.6.7 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_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_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).
```

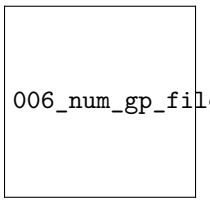
```
    """
    # Calculate pairwise squared Euclidean distances (D) between points in X
    D = squareform(pdist(X, metric='sqeuclidean', out=None, w=theta))
    # Compute Psi = exp(-D)
    Psi = exp(-D)
    # Add a small value to the diagonal for numerical stability (nugget)
    # This is often done 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)
```

```
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]
```

```
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()
```

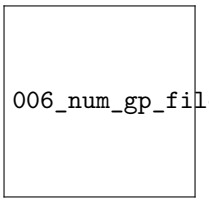
```
--- Training Data (X_train, y_train) ---
x values:
 [[0.  ]
 [0.79]
 [1.57]
 [2.36]
 [3.14]
 [3.93]
 [4.71]
 [5.5 ]]
y values:
 [[ 0.  ]
 [ 0.71]
 [ 1.  ]
 [ 0.71]
```

```
 [ 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]
----------------------------------------

--- 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 (mu_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]
----------------------------------------
```



8.7 Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

9 Matrices

9.1 Derivatives of Quadratic Forms

We present a step-by-step derivation of the general formula

$$\frac{\partial}{\partial \mathbf{v}}(\mathbf{v}^T\mathbf{A}\mathbf{v}) = \mathbf{A}\mathbf{v} + \mathbf{A}^T\mathbf{v}. \tag{9.1}$$

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 $k = 1, \ldots, n$, 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 $\frac{\partial}{\partial \mathbf{v}}(\mathbf{v}^T \mathbf{A}\mathbf{v}) = \mathbf{A}\mathbf{v} + \mathbf{A}^T \mathbf{v}$.

9.2 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
```

9.3 The Moore-Penrose Pseudoinverse

9.3.1 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).

9.3.2 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]]
```

9.4 Strictly Positive Definite Kernels

9.4.1 Definition

Definition 9.1 (Strictly Positive Definite Kernel)**.** A kernel function $k(x, y)$ is called strictly positive definite if for any finite collection of distinct points $x_1, x_2, \ldots, x_n$ in the input space and any non-zero vector of coefficients $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$, the following inequality holds:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j k(x_i, x_j) > 0. \tag{9.2}$$

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.

9.4.2 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.

9.4.3 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.

9.5 Cholesky Decomposition and Positive Definite Matrices

We consider the definiteness of a matrix, before discussing the Cholesky decomposition.

Definition 9.2 (Positive Definite Matrix)**.** A symmetric matrix A is positive definite if all its eigenvalues are positive.

Example 9.1 (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 9.3 (Negative Definite, Positive Semidefinite, and Negative Semidefinite Matrices)**.** Similarily, 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 9.4 (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 9.2 (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 9.3 (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
```

9.5.1 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 **??**, we can compute the correlation matrix Ψ:

$$\Psi = \begin{pmatrix} 1 & e^{-1} \\ e^{-1} & 1 \end{pmatrix}.$$

To determine MLE as in Equation **??**, 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:

$$LDL^T = \begin{pmatrix} 1 & 0 \\ l_{21} & 1 \end{pmatrix} \begin{pmatrix} d_{11} & 0 \\ 0 & d_{22} \end{pmatrix} \begin{pmatrix} 1 & l_{21} \\ 0 & 1 \end{pmatrix} =$$

$$\begin{pmatrix} d_{11} & 0 \\ d_{11}l_{21} & d_{22} \end{pmatrix} \begin{pmatrix} 1 & l_{21} \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} d_{11} & d_{11}l_{21} \\ d_{11}l_{21} & d_{11}l_{21}^2 + d_{22} \end{pmatrix}. \tag{9.3}$$

Using Equation **??**, 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

$$\Psi = LDL^T = U^T DU.$$

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

$$\Psi = \begin{pmatrix} 1 & 0 \\ e^{-1} & \sqrt{1-e^{-2}} \end{pmatrix} \begin{pmatrix} 1 & e^{-1} \\ 0 & \sqrt{1-e^{-2}} \end{pmatrix} = U^T U.$$

9.5.2 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]]
```

9.6 Nyström Approximation

9.6.1 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.

9.6.2 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.

9.6.3 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.

9.6.4 Why Is This Useful?

- **Speed:** The Nyström method is much faster than computing the full kernel matrix. The complexity is roughly $O(l^2 n)$ instead of $O(n^2 d)$ (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.

9.6.5 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 **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 K**. For `n` data points, **K** is an `n x 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, $\tilde{K}$. Instead of inverting the full matrix, we use the **Woodbury matrix identity** on the Nyström approximation, allowing us to efficiently compute $\tilde{K}^{-1}$ without forming the full matrix. This reduces computational complexity to roughly $O(l^2 n)$, 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.

9.6.6 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 (X): 100 coordinate pairs
- Observed Temperatures (y): 100 values
- Prediction Location (x*): Coordinates of the new location

9.6.6.1 Step 1: Nyström Approximation of the Covariance Matrix

1. **Sample Representative Points:** Randomly select `l=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: $\tilde{K} = CW^{+}C^{T}$

9.6.6.2 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 $\mathbf{C}$ and $\mathbf{W}$.
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.

9.6.7 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.

9.6.8 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 $\lambda = 0.1$.

9.6.8.1 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.

9.6.8.2 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:**

- $C^T C = \begin{pmatrix} 127 & 42.25 \\ 42.25 & 14.0625 \end{pmatrix}$
- $W = \begin{pmatrix} 9 & 3 \\ 3 & 1 \end{pmatrix}$

- The matrix to invert is $W + \frac{1}{0.1} C^T C = W + 10 \cdot (C^T C)$, 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.

$$(W + \frac{1}{\lambda} C^T C)^{-1} \approx \begin{pmatrix} 0.22 & -0.66 \\ -0.66 & 1.99 \end{pmatrix}$$

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.

9.7 Extending spotpython's Kriging Surrogate with Nyström Approximation for Enhanced Scalability

9.7.1 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 $O(n\,l^2)$. This makes Kriging viable even when the number of function evaluations is large.

9.7.2 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

---

9.8 Theoretical Foundations: The Nyström–Kriging Framework

9.8.1 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 $X = \{x_1, \ldots, x_n\}$ and observations $y = \{y_1, \ldots, y_n\}$:

$$y \sim \mathcal{N}\big(m(X),\, K(X, X) + \sigma_n^2 I\big)$$

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 $K(X, X) + \sigma_n^2 I$.

9.8.2 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 $\leq l$.

9.8.3 Justification for Landmark Selection

Uniform sampling without replacement is an effective and inexpensive strategy for selecting landmarks across varied datasets and kernels.

9.9 Implementation: A Scalable Kriging Class for spotpython

9.9.1 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.th
        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
```

9.10 Implementation Details

9.10.1 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

9.10.2 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

9.10.3 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

9.10.4 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` → `build_psi_vec`, preserving mixed-type handling

9.10.5 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.

9.11 Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

10 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.

10.1 Balancing Exploitation and Exploration

A crucial aspect of selecting infill points is navigating the inherent trade-off between exploitation and exploration.

Definition 10.1 (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 10.2 (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.

(**Forr08a?**) emphasizes that effective infill criteria are designed to combine both exploitation and exploration.

10.2 Expected Improvement (EI)

Expected Improvement (EI) is one of the most influential and widely-used infill criteria. Formalized by (**Jones1998?**) and building upon the work of (**mockus1978toward?**), 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:

$$EI(x) = \sigma(x)\left[Z\Phi(Z) + \phi(Z)\right]$$

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.
- $\Phi(Z)$ 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.

10.3 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 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 (**Forr08a?**) 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.

10.4 Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

Part III

Sequential Parameter Optimization Toolbox (SPOT)

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,
```

```
    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) = 0.001607
Iteration 2: f(x) = 0.007183
Iteration 3: f(x) = 0.009504
Iteration 4: New best f(x) = 0.000000
Iteration 5: f(x) = 0.000003
Iteration 6: f(x) = 0.000002
Iteration 7: f(x) = 0.000001
Iteration 8: f(x) = 0.000001
Iteration 9: f(x) = 0.000002
Iteration 10: f(x) = 0.000000
Iteration 11: f(x) = 0.000001
Iteration 12: f(x) = 0.000003
Iteration 13: f(x) = 0.000000
Iteration 14: f(x) = 0.000000
Iteration 15: f(x) = 0.000001
Best solution: [-0.0003343  -0.00013835]
Best value: 1.3089731064068852e-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
)
```

```
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): 0.036226
Config 2 (fewer initial): 0.015384
```

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
```

```
TensorBoard logging disabled
Initial best: f(x) = 20.392774
Iteration 1: f(x) = 21.363476
Iteration 2: f(x) = 47.301823
Iteration 3: f(x) = 54.722105
Iteration 4: f(x) = 26.938125
Iteration 5: f(x) = 47.976435
Iteration 6: New best f(x) = 11.815352
Iteration 7: f(x) = 42.986117
Iteration 8: f(x) = 18.405281
Iteration 9: New best f(x) = 9.416725
Iteration 10: New best f(x) = 8.466069
Iteration 11: f(x) = 30.100210
Iteration 12: New best f(x) = 2.903089
Iteration 13: f(x) = 14.397238
Iteration 14: f(x) = 6.076849
Iteration 15: f(x) = 3.075001
Iteration 16: f(x) = 8.238939
Iteration 17: f(x) = 32.184621
Iteration 18: f(x) = 24.864179
Iteration 19: f(x) = 47.118258
Iteration 20: f(x) = 39.419666
Iteration 21: f(x) = 53.085310
Iteration 22: f(x) = 9.975504
Iteration 23: f(x) = 35.001393
Iteration 24: f(x) = 38.805263
Iteration 25: f(x) = 53.761437
Iteration 26: f(x) = 22.263620
Iteration 27: f(x) = 36.329232
Iteration 28: f(x) = 43.768628
Iteration 29: f(x) = 48.136701
Iteration 30: f(x) = 33.012990
Iteration 31: f(x) = 39.630292
Iteration 32: f(x) = 56.076800
Iteration 33: f(x) = 56.395240
Iteration 34: f(x) = 38.348341
Iteration 35: f(x) = 49.557146
Iteration 36: f(x) = 59.083493
Iteration 37: f(x) = 30.011270
Iteration 38: f(x) = 41.465815
Iteration 39: f(x) = 27.727924
Iteration 40: f(x) = 42.088395
Iteration 41: f(x) = 28.204654
Iteration 42: f(x) = 18.178644
Iteration 43: f(x) = 35.087961
```

```
Iteration 44: f(x) = 57.612698
Iteration 45: f(x) = 38.620443
Iteration 46: f(x) = 54.041182
Iteration 47: f(x) = 28.335777
Iteration 48: f(x) = 31.404859
Iteration 49: f(x) = 68.751895
Iteration 50: f(x) = 62.442382
Iteration 51: f(x) = 53.043276
Iteration 52: f(x) = 49.678065
Iteration 53: f(x) = 51.523703
Iteration 54: f(x) = 45.533620
Iteration 55: New best f(x) = 2.181277
Iteration 56: New best f(x) = 1.996482
Iteration 57: f(x) = 51.513483
Iteration 58: f(x) = 43.295834
Iteration 59: f(x) = 3.690653
Iteration 60: f(x) = 33.624017
Iteration 61: f(x) = 37.625215
Iteration 62: f(x) = 67.144123
Iteration 63: f(x) = 25.844100
Iteration 64: f(x) = 54.673898
Iteration 65: f(x) = 11.388202
Iteration 66: f(x) = 39.385158
Iteration 67: f(x) = 4.226770
Iteration 68: f(x) = 67.022923
Iteration 69: f(x) = 28.240244
Iteration 70: f(x) = 36.207929

Experiment Results (seed=12345):
Best solution: [ 0.99166413 -0.99787425 -0.0037072 ]
Best value: 1.9964821694753638
Iterations: 70
Function evaluations: 100
```

11.3.3 Example 3: Multiple Independent Runs

To test robustness, run the same optimization with different seeds:

```python
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) = 0.145422
Run with seed  123: f(x) = 0.023381
Run with seed  456: f(x) = 0.022664
Run with seed  789: f(x) = 0.035378
Run with seed 1011: f(x) = 0.115351

Best result: 0.022664
Worst result: 0.145422
Mean: 0.068439
Std dev: 0.051664
```

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)
```

```
print("Results are identical:", np.allclose(result1.x, result2.x))
print(f"Best value: {result1.fun:.6f}")
```

```
Results are identical: True
Best value: 1.024940
```

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: 0.066466
Variance at best: 0.008717
```

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) = 0.796656
Seed 11: f(x) = 0.042354
Seed 12: f(x) = 1.156801
Seed 13: f(x) = 0.720165
Seed 14: f(x) = 1.193134
Seed 15: f(x) = 1.617529
Seed 16: f(x) = 0.345078
Seed 17: f(x) = 0.677278
Seed 18: f(x) = 1.458146
```

```
Seed 19: f(x) = 1.349480

Best result with seed 11: 0.042354
```

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

12 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.

12.1 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.

12.2 Fallback Strategies

SpotOptim supports two fallback strategies, controlled by the `acquisition_failure_strategy` parameter:

12.2.1 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

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()
```

12.2.2 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)],
    max_iter=100,
    n_initial=20,
    acquisition_failure_strategy="mm",  # Morris-Mitchell
    verbose=True
)

result = optimizer.optimize()
```

12.3 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

12.4 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

12.5 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)
    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()

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")
```

12.6 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:

```
# Strict tolerance (smaller value) - fewer fallbacks
optimizer_strict = SpotOptim(
    fun=objective,
    bounds=bounds,
    tolerance_x=1e-6,  # Very small - almost never triggers fallback
    acquisition_failure_strategy="mm"
)

# Relaxed tolerance (larger value) - more fallbacks
optimizer_relaxed = SpotOptim(
    fun=objective,
    bounds=bounds,
    tolerance_x=0.5,  # Larger - triggers fallback more often
```

```
    acquisition_failure_strategy="mm"
)
```

12.7 Best Practices

12.7.1 1. Use Random for Most Problems

The random strategy (default) is sufficient for most optimization problems:

```
optimizer = SpotOptim(
    fun=objective,
    bounds=bounds,
    acquisition_failure_strategy="random"  # Good default choice
)
```

12.7.2 2. Use Morris-Mitchell for Intensive Sampling

When you have a large budget and want maximum exploration:

```
optimizer = SpotOptim(
    fun=expensive_objective,
    bounds=bounds,
    max_iter=200,  # Large budget
    acquisition_failure_strategy="mm"  # Maximize space coverage
)
```

12.7.3 3. Monitor Fallback Activations

Enable verbose mode to see when fallbacks are triggered:

```
optimizer = SpotOptim(
    fun=objective,
    bounds=bounds,
    acquisition_failure_strategy="mm",
    verbose=True  # Shows fallback messages
)
```

12.7.4 4. Adjust Tolerance Based on Problem Scale

For problems with small search spaces, use smaller tolerance:

```
# Small search space
optimizer_small = SpotOptim(
    fun=objective,
    bounds=[(-1, 1), (-1, 1)],
    tolerance_x=0.01,  # Small tolerance for small space
    acquisition_failure_strategy="random"
)

# Large search space
optimizer_large = SpotOptim(
    fun=objective,
    bounds=[(-100, 100), (-100, 100)],
    tolerance_x=1.0,  # Larger tolerance for large space
    acquisition_failure_strategy="mm"
)
```

12.8 Technical Details

12.8.1 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.

12.8.2 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.

12.9 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 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.

13.1 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

13.2 Quick Start

13.2.1 Basic Usage

```
from spotoptim.data import get_diabetes_dataloaders

# 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
```

13.2.2 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}")
```

13.3 Function Reference

13.3.1 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
)
```

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_[:3]}")  # First 3 features
```

13.4 DiabetesDataset Class

PyTorch Dataset implementation for the diabetes dataset.

Signature:

```
DiabetesDataset(X, y, transform=None, target_transform=None)
```

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

13.4.1 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,)
```

13.5 Advanced Usage

13.5.1 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]
```

13.5.2 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
```

13.5.3 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
```

13.5.4 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
```

13.5.5 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 ...
```

13.6 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()
```

13.7 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=["num", "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}")
```

13.8 Best Practices

13.8.1 1. Always Use Feature Scaling

```
# Good: Features are standardized
train_loader, test_loader, scaler = get_diabetes_dataloaders(
    scale_features=True
)
```

Neural networks typically perform better with normalized inputs.

13.8.2 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)
```

13.8.3 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.

13.8.4 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

13.8.5 5. Save the Scaler for Production

```
import pickle
from spotoptim.data import get_diabetes_dataloaders

# Train with scaling
train_loader, test_loader, scaler = get_diabetes_dataloaders(
    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:
    scaler = pickle.load(f)

new_data_scaled = scaler.transform(new_data)
```

13.9 Troubleshooting

13.9.1 Issue: Out of Memory

Solution: Reduce batch size or disable pin_memory

```
train_loader, test_loader, scaler = get_diabetes_dataloaders(
    batch_size=16,      # Smaller batches
    pin_memory=False    # Disable if not using GPU
)
```

13.9.2 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
)
```

13.9.3 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
)
```

13.9.4 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.

13.10 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

For more examples, see: - `examples/diabetes_dataset_example.py` - `notebooks/demos.ipynb` - Test suite: `tests/test_diabetes_dataset.py`

14 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.

14.1 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 integerstring conversion
- Seamless integration with neural network hyperparameters
- Mix factor variables with numeric/integer variables

14.2 Quick Start

14.2.1 Basic Factor Variable Usage

```
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}")
```

14.2.2 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!
        )

        # 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}")
```

14.3 Mixed Variable Types

14.3.1 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()

        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=["num", "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}")
```

14.4 Multiple Factor Variables

14.4.1 Optimizing Both Activation and Optimizer

```
from spotoptim import SpotOptim
from spotoptim.data import get_diabetes_dataloaders
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_dataloaders()

        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", "num"],
    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}")
```

14.5 Advanced Usage

14.5.1 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}")
```

14.5.2 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

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}")
```

14.6 How It Works

14.6.1 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 → strings

```
# [0, 2, 2] → ["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
```

14.6.2 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=["num", "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"]
# )

# 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=["num", "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]}")
```

14.7 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


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_dataloaders(
            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()
                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=["num", "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(
        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()
```

14.8 Best Practices

14.8.1 Do's

Use descriptive string values

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

Explicitly specify var_type for clarity

```
var_type=["num", "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)
```

Mix factor variables with numeric/integer variables

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

14.8.2 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
```

Don't use empty tuples

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

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

14.9 Troubleshooting

14.9.1 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.

14.10 Summary

Factor variables in SpotOptim enable:

- **Categorical optimization**: Optimize over discrete string choices
- **Automatic conversion**: Seamless integerstring 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!

14.11 See Also

- LinearRegressor Documentation - Neural network class supporting string-based activation functions
- Diabetes Dataset Utilities - Data loading utilities used in examples
- Variable Types - Overview of all variable types in SpotOptim
- Save and Load - Saving and loading optimization results with factor variables

15 Kriging Surrogate Integration Summary

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)
- `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

kriging = Kriging(seed=42)
optimizer = SpotOptim(
    fun=objective,
    bounds=bounds,
    surrogate=kriging,  # Just pass the Kriging instance
    seed=42
)
```

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

kriging = Kriging(noise=1e-6, seed=42)
optimizer = SpotOptim(fun=objective, bounds=bounds, surrogate=kriging)
result = optimizer.optimize()
```

15.6.2 Custom Parameters

```
kriging = Kriging(
    noise=1e-4,
    min_theta=-2.0,
    max_theta=3.0,
    seed=123
)
```

15.6.3 Prediction with Uncertainty

```
model = Kriging(seed=42)
model.fit(X_train, y_train)
y_pred, y_std = model.predict(X_test, return_std=True)
```

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 = _k _k(x_ik - x_jk)²
   - Apply kernel: R_ij = exp(-D_ij)
   - Add nugget: R_ii += noise
2. **Maximum Likelihood:**
   - Optimize via differential evolution
   - Minimize: (n/2)log(²) + (1/2)log|R|
   - Concentrated likelihood (profiled out)
3. **Prediction:**
   - Mean: f(x) = + (x)R¹r
   - Variance: s²(x) = ²[1 + - (x)R¹(x)]
   - Uses Cholesky decomposition for efficiency

15.7.3 Key Arguments Passed from SpotOptim

SpotOptim passes these to the surrogate via the standard interface:

During fit:

```
surrogate.fit(X, y)
```

- `X`: Training points (n_initial or accumulated evaluations)
- `y`: Function values

During predict:

```
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 Conclusion

Implementation of a Kriging surrogate into SpotOptim with:

- Full scikit-learn compatibility
- Comprehensive test coverage (9 new tests)
- Complete documentation
- Example notebook
- Zero breaking changes
- All 25 tests passing

16 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.

16.1 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

16.2 Quick Start

16.2.1 Basic Usage

```
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}")
```

16.2.2 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}")
```

16.2.3 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']}")
```

16.3 Function Reference

16.3.1 `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: "Adadelta", "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)
```

16.4 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
Adadelta	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

16.5 Use Cases

16.5.1 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}")
```

16.5.2 Hyperparameter Optimization with SpotOptim

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

def train_and_evaluate(X):
    """Objective function for hyperparameter optimization."""
    results = []

    # Load data once
    train_loader, test_loader, _ = get_diabetes_dataloaders(
        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=[
        (-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=["num", "factor", "int", "int"],
    max_iter=30,
    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}")
```

16.5.3 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}")
```

Output:

```
log_lr     unified_lr   Adam         SGD          RMSprop
------------------------------------------------------------
-4.00      0.000100     0.00000010   0.00000100   0.00000100
```

```
-3.56       0.000275      0.00000028    0.00000275    0.00000275
-3.11       0.000759      0.00000076    0.00000759    0.00000759
-2.67       0.002089      0.00000209    0.00002089    0.00002089
-2.22       0.005754      0.00000575    0.00005754    0.00005754
-1.78       0.015849      0.00001585    0.00015849    0.00015849
-1.33       0.043652      0.00004365    0.00043652    0.00043652
-0.89       0.120226      0.00012023    0.00120226    0.00120226
-0.44       0.331131      0.00033113    0.00331131    0.00331131
0.00        1.000000      0.00100000    0.01000000    0.01000000
```

16.5.4 Custom Learning Rate Schedules

```python
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
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}")
```

16.5.5 Direct Usage Without LinearRegressor

```python
import torch
import torch.nn as nn
```

```python
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}")
```

16.6 Best Practices

16.6.1 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

16.6.2 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
- 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 (Adadelta, Adagrad, etc.):

- Specialized use cases
- Start with `lr=1.0` and adjust

16.6.3 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)
```

16.7 Troubleshooting

16.7.1 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
```

16.7.2 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
```

16.7.3 Issue: Different results across optimizer runs

Solution: Set random seed for reproducibility:

```
import torch
torch.manual_seed(42)
```

16.7.4 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
```

16.7.5 Issue: Optimizer not supported

Solution: Check supported optimizers:

```
from spotoptim.utils.mapping import OPTIMIZER_DEFAULT_LR
print("Supported optimizers:", list(OPTIMIZER_DEFAULT_LR.keys()))
```

16.8 Technical Details

16.8.1 How It Works

The mapping is simple but effective:

```
actual_lr = lr_unified * default_lr[optimizer_name]
```

For example:

- `map_lr(1.0, "Adam")` → `1.0 * 0.001 = 0.001`
- `map_lr(0.5, "SGD")` → `0.5 * 0.01 = 0.005`
- `map_lr(2.0, "RMSprop")` → `2.0 * 0.01 = 0.02`

This ensures that the same unified learning rate gives optimizer-specific learning rates in their typical working ranges.

16.8.2 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`

16.8.3 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,
}
```

16.9 Examples

16.9.1 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]}, "
      f"test_MSE={best_overall_loss:.4f}")
print(f"Actual learning rate used: {best_overall[2]:.6f}")
```

16.10 See Also

- LinearRegressor Documentation - Neural network class with lr parameter
- Diabetes Dataset Utilities - Data loading for examples
- Hyperparameter Optimization - Using map_lr with SpotOptim
- PyTorch Optimizer Documentation - Official PyTorch reference

16.11 References

- Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv:1412.6980.
- Loshchilov, I., & Hutter, F. (2017). Decoupled weight decay regularization. arXiv:1711.05101.
- PyTorch Team. (2023). PyTorch Optimizer Documentation. https://pytorch.org/docs/stable/optim.html

17 Unified Learning Rate Interface

This module provides a sophisticated unified learning rate interface for PyTorch optimizers through the `map_lr()` function and integration with `LinearRegressor`.

17.1 Overview

Different PyTorch optimizers operate on vastly different learning rate scales:

- **Adam** typically uses lr ~ 0.0001-0.001
- **SGD** typically uses lr ~ 0.01-0.1
- **RMSprop** typically uses lr ~ 0.001-0.01

This makes it difficult to:

1. Compare optimizer performance fairly
2. Optimize learning rate as a hyperparameter across different optimizers
3. Switch between optimizers without retuning learning rates

The `map_lr()` function solves this by providing a unified learning rate scale where **lr=1.0 corresponds to each optimizer's PyTorch default**.

17.2 Key Features

- **Unified Interface**: Single learning rate parameter works across all optimizers
- **Fair Comparison**: Same unified lr gives optimizer-specific optimal ranges
- **Hyperparameter Optimization**: Optimize one learning rate for multiple optimizers
- **Backward Compatible**: Existing code continues to work
- **Well-tested**: 36 comprehensive tests covering all use cases
- **Documented**: Extensive docstrings and examples

17.3 Usage

17.3.1 Basic Usage with LinearRegressor

```
from spotoptim.nn.linear_regressor import LinearRegressor

# Create model with unified lr=1.0 (gives each optimizer its default)
model = LinearRegressor(input_dim=10, output_dim=1, lr=1.0)

# Adam gets 0.001 (its default)
optimizer_adam = model.get_optimizer("Adam")

# SGD gets 0.01 (its default)
optimizer_sgd = model.get_optimizer("SGD")

# RMSprop gets 0.01 (its default)
optimizer_rmsprop = model.get_optimizer("RMSprop")
```

17.3.2 Using Custom Unified Learning Rate

```
# Using lr=0.5 scales all optimizers by 0.5
model = LinearRegressor(input_dim=10, output_dim=1, lr=0.5)

optimizer_adam = model.get_optimizer("Adam")     # Gets 0.5 * 0.001 = 0.0005
optimizer_sgd = model.get_optimizer("SGD")       # Gets 0.5 * 0.01 = 0.005
```

17.3.3 Direct Use of map_lr()

```
from spotoptim.utils.mapping import map_lr

# Map unified lr to optimizer-specific lr
lr_adam = map_lr(1.0, "Adam")      # Returns 0.001
lr_sgd = map_lr(1.0, "SGD")        # Returns 0.01
lr_rmsprop = map_lr(1.0, "RMSprop")  # Returns 0.01

# Scale by 2x
lr_adam = map_lr(2.0, "Adam")      # Returns 0.002
lr_sgd = map_lr(2.0, "SGD")        # Returns 0.02
```

17.3.4 Hyperparameter Optimization

```
from spotoptim import SpotOptim
import numpy as np

def train_model(X):
    results = []
    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
        model = LinearRegressor(input_dim=10, output_dim=1, lr=lr_unified)
        optimizer = model.get_optimizer(optimizer_name)

        # Train and evaluate
        # ... training code ...
        results.append(test_loss)
    return np.array(results)

# Optimize unified lr across different optimizers
optimizer = SpotOptim(
    fun=train_model,
    bounds=[(-4, 0), ("Adam", "SGD", "RMSprop")],
    var_type=["num", "factor"],
    max_iter=30
)
result = optimizer.optimize()
```

17.4 Supported Optimizers

All major PyTorch optimizers are supported with their default learning rates:

Optimizer	Default LR	Typical Range
Adam	0.001	0.0001-0.01
AdamW	0.001	0.0001-0.01
Adamax	0.002	0.0001-0.01
NAdam	0.002	0.0001-0.01
RAdam	0.001	0.0001-0.01
SGD	0.01	0.001-0.1
RMSprop	0.01	0.001-0.1

Optimizer	Default LR	Typical Range
Adagrad	0.01	0.001-0.1
Adadelta	1.0	0.1-10.0
ASGD	0.01	0.001-0.1
LBFGS	1.0	0.1-10.0
Rprop	0.01	0.001-0.1

17.5 API Reference

17.5.1 `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. Typical range: [0.001, 100.0]
- `optimizer_name` (str): Name of the PyTorch optimizer
- `use_default_scale` (bool): Whether to scale by optimizer's default (default: True)

Returns:

- `float`: The optimizer-specific learning rate

Example:

```
lr = map_lr(1.0, "Adam")  # Returns 0.001 (Adam's default)
lr = map_lr(0.5, "SGD")   # Returns 0.005 (0.5 * SGD's default)
```

17.5.2 `LinearRegressor(..., lr=1.0)`

Parameter:

- `lr` (float): Unified learning rate multiplier. Default: 1.0

New Behavior in `get_optimizer()`:

- If `lr` is not specified, uses `self.lr`
- Automatically maps unified lr to optimizer-specific lr
- Can override model's lr by passing `lr` parameter

17.6 Design Rationale

17.6.1 Why Unified Learning Rates?

The approach is based on spotPython's `optimizer_handler()` but improved:

1. **Separation of Concerns**: Mapping logic in separate, testable module
2. **Flexibility**: Can be used independently or integrated with models
3. **Transparency**: Clear mapping based on PyTorch defaults
4. **Extensibility**: Easy to add new optimizers
5. **Type Safety**: Comprehensive error handling and validation

17.6.2 Comparison with spotPython

Feature	spotPython	spotoptim
Approach	`lr_mult * default_lr`	`map_lr(lr_unified, optimizer)`
Module	`optimizer_handler()`	`map_lr()` + integration
Testing	Minimal	36 comprehensive tests
Documentation	Basic	Extensive with examples
Reusability	Coupled	Standalone function
Error Handling	Basic	Comprehensive validation

17.6.3 Log-scale Optimization

For hyperparameter optimization, use log-scale for unified lr:

```
# Sample from log10 scale [-4, 0]
log_lr = -2.5  # Sampled value
lr_unified = 10 ** log_lr  # 0.00316

# Map to optimizer-specific
lr_adam = map_lr(lr_unified, "Adam")  # 0.00316 * 0.001 = 0.00000316
lr_sgd = map_lr(lr_unified, "SGD")    # 0.00316 * 0.01 = 0.0000316
```

This gives a reasonable search range across all optimizers.

17.7 Examples

See `examples/unified_learning_rate_demo.py` for comprehensive examples including: 1. Basic unified interface usage 2. Custom unified learning rates 3. Training with different optimizers 4. Direct use of map_lr() 5. Log-scale hyperparameter optimization 6. Complete hyperparameter optimization scenario

17.8 References

- PyTorch Optimizer Documentation
- spotPython's `optimizer_handler()` function (inspiration)
- Hyperparameter Optimization Best Practices

17.9 Contributing

When adding new optimizers:

1. Add default lr to `OPTIMIZER_DEFAULT_LR` dict in `mapping.py`
2. Verify the default against PyTorch documentation
3. Add tests in `test_mapping.py`
4. Update this README

17.10 License

Same as spotoptim package (see main LICENSE file).

18 Multi-Objective Optimization Support in SpotOptim

18.1 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.

18.2 What Was Implemented

18.2.1 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.

18.3 Usage Examples

18.3.1 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"MO values stored: {optimizer.y_mo.shape}")  # (30, 2)
```

18.3.2 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]
```

18.3.3 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
```

18.3.4 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()
```

18.3.5 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
    repeats_surrogate=2,
    seed=42
)

result = optimizer.optimize()
# Works seamlessly with noise handling
```

18.4 Common Scalarization Strategies

18.4.1 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

18.4.2 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

18.4.3 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

18.4.4 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

18.4.5 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

18.5 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

18.6 Implementation Details

18.6.1 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

18.6.2 Data Flow

```
User Function → y_mo (raw) → _mo2so() → y_ (single-objective)
                      ↓
                y_mo (stored)
```

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`

18.6.3 Backward Compatibility

Fully backward compatible:

- Single-objective functions work unchanged
- `fun_mo2so` defaults to `None`
- `y_mo` is `None` for single-objective problems
- No breaking changes to existing code

18.7 Limitations and Notes

18.7.1 What This Is

- Scalarization approach to multi-objective optimization
- Single solution found per optimization run
- Different scalarizations → different Pareto solutions
- Suitable for preference-based multi-objective optimization

18.7.2 What This Is Not

- 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

18.7.3 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

18.8 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

18.9 Summary

SpotOptim provides flexible multi-objective optimization support through:

- Automatic detection of multi-objective functions
- Customizable scalarization strategies via `fun_mo2so`
- Complete storage of multi-objective values in `y_mo`
- Full integration with existing features (noise, OCBA, TensorBoard, etc.)
- 100% backward compatible with existing code

This implementation mirrors the approach used in spotPython's Spot class, providing consistency across the ecosystem.

19 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.

19.1 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.

19.2 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

19.3 Usage

19.3.1 Basic Usage

```
import numpy as np
from spotoptim import SpotOptim

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

```
# 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)
```

19.3.2 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
)
```

19.3.3 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:

```
# 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'])
```

19.4 Plot Interpretation

The visualization consists of 4 panels:

19.4.1 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

19.4.2 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
- Lower values (blue) indicate high confidence, higher values (red) indicate uncertainty

19.4.3 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

19.4.4 Bottom Right: Uncertainty Contour with Points

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

19.5 Parameters

19.5.1 Dimension Selection

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

19.5.2 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)

19.5.3 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

19.5.4 Color Scaling

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

19.5.5 Display

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

19.6 Examples

19.6.1 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
)
```

19.6.2 Example 2: Using Kriging Surrogate

```
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'])
```

19.6.3 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
)
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'])
```

19.7 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

19.8 Comparison with spotpython's plotkd()

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

19.8.1 Similarities

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

19.8.2 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)

19.9 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
```

19.10 See Also

- `notebooks/demos.ipynb`: Example 4 demonstrates `plot_surrogate()`
- `examples/plot_surrogate_demo.py`: Standalone example script
- `tests/test_plot_surrogate.py`: Comprehensive test suite

20 Point Selection Implementation

20.1 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.

20.2 Implementation Details

20.2.1 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')

20.2.2 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
    - 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`

20.3 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.

20.4 Example Usage

```
from spotoptim import SpotOptim

# Without point selection (default behavior)
optimizer1 = SpotOptim(
    fun=expensive_function,
    bounds=bounds,
    max_iter=100,
    n_initial=20
)

# With point selection using distant method
optimizer2 = SpotOptim(
    fun=expensive_function,
    bounds=bounds,
    max_iter=100,
    n_initial=20,
    max_surrogate_points=50,
    selection_method='distant'
)

# With point selection using best cluster method
optimizer3 = SpotOptim(
```

```
    fun=expensive_function,
    bounds=bounds,
    max_iter=100,
    n_initial=20,
    max_surrogate_points=50,
    selection_method='best'
)
```

20.5 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

20.6 Comparison with spotpython

Feature	spotpython	SpotOptim
Point selection via clustering		
'distant' method		
'best' method		
Selection dispatcher		
Nyström approximation		
Modular design	(utils.aggregate)	(class methods)

20.7 References

- spotpython implementation: `src/spotpython/spot/spot.py` lines 1646-1778
- spotpython utilities: `src/spotpython/utils/aggregate.py` lines 262-336

21 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.

21.1 Key Concepts

21.1.1 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.

21.1.2 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.

21.2 Quick Start

21.2.1 Basic Save and Load

```python
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=50,
    n_initial=15,
    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}")
```

21.3 Distributed Workflow

The save/load functionality enables a powerful workflow for distributed optimization:

21.3.1 Step 1: Define Experiment Locally

```
import numpy as np
from spotoptim import SpotOptim

# Define configuration locally (no need to run optimization yet)
optimizer = SpotOptim(
    bounds=[(-10, 10), (-10, 10), (-10, 10)],
    max_iter=200,
    n_initial=30,
    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.")
```

21.3.2 Step 2: Execute on Remote Machine

```
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.")
```

21.3.3 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}")
```

21.4 Advanced Usage

21.4.1 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
```

21.4.2 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
optimizer.save_result(prefix="my_result", overwrite=True)
print("File overwritten successfully")
```

21.4.3 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=30,
    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 = 50  # Increase budget

# Continue optimization
```

```
result2 = opt2.optimize()
print(f"After continuation: {result2.nfev} evaluations, best={result2.fun:.6f}")
```

21.5 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=40,
    n_initial=15,
    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:
    print(f"Mean values available: {len(loaded_opt.mean_y)}")
    print(f"Variance values available: {len(loaded_opt.var_y)}")
```

21.6 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=["num", "int", "factor", "num"],
    var_name=["continuous", "integer", "categorical", "another_cont"],
    max_iter=50,
    n_initial=20,
    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
```

21.7 Best Practices

21.7.1 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()
```

21.7.2 2. Use Meaningful Prefixes

Organize your experiments with descriptive prefixes:

```
# 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
```

21.7.3 3. Save Experiments Before Remote Execution

```
# Define locally
optimizer = SpotOptim(bounds=bounds, max_iter=500, seed=42)
optimizer.save_experiment(prefix="remote_job")

# Transfer file to remote machine
# Execute remotely
# Transfer results back
# Analyze locally
```

21.7.4 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
```

21.7.5 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)
```

21.8 Complete Example: Multi-Machine Workflow

Here's a complete example demonstrating the entire workflow:

21.8.1 Local Machine (Setup)

```
# setup_experiment.py
import numpy as np
from spotoptim import SpotOptim
```

```
import os

# Create experiments directory
os.makedirs("experiments", exist_ok=True)

# Define multiple experiments
experiments = [
    {"seed": 42, "max_iter": 100, "prefix": "exp_seed42"},
    {"seed": 123, "max_iter": 100, "prefix": "exp_seed123"},
    {"seed": 999, "max_iter": 100, "prefix": "exp_seed999"},
]

for exp_config in experiments:
    optimizer = SpotOptim(
        bounds=[(-10, 10), (-10, 10), (-10, 10)],
        max_iter=exp_config["max_iter"],
        n_initial=30,
        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.")
```

21.8.2 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")

# 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.")
```

21.8.3 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)
    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")
```

21.9 Technical Details

21.9.1 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

21.9.2 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.

21.9.3 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

21.9.4 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)
```

21.10 Troubleshooting

21.10.1 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()
```

21.10.2 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}")
```

21.10.3 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)
```

21.10.4 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
```

21.11 See Also

- Reproducibility Manual: Learn about using seeds for reproducible results
- TensorBoard Manual: Monitor optimization progress in real-time

22 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.

22.1 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

22.2 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

# Run optimization with periodic success rate checks
optimizer = SpotOptim(
    fun=rosenbrock,
    bounds=[(-2, 2), (-2, 2)],
    max_iter=200,
    n_initial=20,
    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: 0.000000
Total evaluations: 200
Final success rate: 4.00%
→ Low success rate: Optimization has likely converged
```

22.3 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=100,
    n_initial=20,
    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}")
```

22.4 Accessing Success Rate

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

```
optimizer = SpotOptim(fun=objective, bounds=bounds, max_iter=50)
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()
```

22.5 Interpreting Success Rate

22.5.1 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!

22.5.2 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.

22.5.3 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.

22.6 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=100,
    n_initial=20,
    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

22.7 Example: Monitoring Optimization Progress

```
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

# Run optimization with periodic success rate checks
optimizer = SpotOptim(
    fun=rosenbrock,
    bounds=[(-2, 2), (-2, 2)],
    max_iter=100,
    n_initial=20,
    tensorboard_log=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")
```

22.8 Example: Comparing Multiple Runs

```
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": 50, "name": "Small initial"},
    {"n_initial": 30, "max_iter": 50, "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%}")
```

22.9 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=50,
    n_initial=15,
    repeats_initial=3,    # 3 evaluations per initial point
    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]}")
```

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

22.10 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:

```
optimizer = SpotOptim(
    fun=objective,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=50,
    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)
```

22.11 Best Practices

22.11.1 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
```

22.11.2 2. Combine with TensorBoard

Always enable TensorBoard logging for visual monitoring:

```
optimizer = SpotOptim(
    fun=expensive_function,
    bounds=bounds,
    max_iter=1000,
    tensorboard_log=True,  # Track success_rate visually
    tensorboard_path="runs/long_optimization"
)
```

22.11.3 3. Use as Stopping Criterion

Consider stopping when success rate drops very low:

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

22.11.4 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=50)
    result = opt.optimize()
    print(f"{acq}: success_rate={opt.success_rate:.2%}, best={result.fun:.6f}")
```

22.12 Technical Details

22.12.1 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.

22.12.2 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

22.12.3 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)

22.13 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

23 TensorBoard Log Cleaning Feature

23.1 Summary

Automatic cleaning of old TensorBoard log directories with the `tensorboard_clean` parameter.

23.2 Usage

23.2.1 Basic Usage

```
from spotoptim import SpotOptim

# Remove old logs and create new log directory
optimizer = SpotOptim(
    fun=objective,
    bounds=[(-5, 5), (-5, 5)],
    tensorboard_log=True,
    tensorboard_clean=True,  # Removes all subdirectories in 'runs'
    verbose=True
)

result = optimizer.optimize()
```

23.2.2 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)

23.3 Implementation Details

23.3.1 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}")
```

23.3.2 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)
5. If `tensorboard_log=True`, a new log directory is created
6. Optimization proceeds normally

23.4 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

23.5 Warning

IMPORTANT: Setting `tensorboard_clean=True` permanently deletes all subdirectories in the 'runs' folder. Make sure to save important logs elsewhere before enabling this feature.

24 TensorBoard Logging in SpotOptim

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

24.1 Quick Start

24.1.1 1. Enable TensorBoard Logging

```
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=50,
    n_initial=15,
    tensorboard_log=True,  # Enable logging
    verbose=True
)

result = optimizer.optimize()
```

24.1.2 2. View Logs in TensorBoard

In a separate terminal, run:

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

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

24.2 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.

24.2.1 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
   ```

24.3 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",
    ...
)
```

24.4 What Gets Logged

24.4.1 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

24.4.2 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.

24.5 Examples

24.5.1 Basic Usage

```
optimizer = SpotOptim(
    fun=lambda X: np.sum(X**2, axis=1),
    bounds=[(-5, 5), (-5, 5)],
    max_iter=30,
    tensorboard_log=True,
    verbose=True
)
result = optimizer.optimize()
```

24.5.2 Noisy Optimization

```
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=50,
    repeats_initial=3,
    repeats_surrogate=2,
    tensorboard_log=True,
    tensorboard_path="runs/noisy_exp",
    seed=42
)
result = optimizer.optimize()
```

24.5.3 With OCBA

```
optimizer = SpotOptim(
    fun=noisy_objective,
    bounds=[(-5, 5), (-5, 5)],
    max_iter=50,
    repeats_initial=2,
    ocba_delta=3,  # Re-evaluate 3 promising points per iteration
    tensorboard_log=True,
    seed=42
)
result = optimizer.optimize()
```

24.6 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
```

24.7 TensorBoard Features

24.7.1 SCALARS Tab

- View convergence curves
- Compare optimization progress across runs
- Track how metrics change over iterations

24.7.2 HPARAMS Tab

- Explore hyperparameter space
- See which parameter combinations work best
- Identify patterns in successful configurations

24.7.3 Text Tab

- View configuration details
- Check run metadata

24.8 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
```

24.9 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

This demonstrates:

- Deterministic optimization (Rosenbrock function)
- Noisy optimization with repeated evaluations
- OCBA for intelligent re-evaluation

24.10 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.

24.11 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)

24.12 Performance Notes

TensorBoard logging has minimal overhead:

- $< 1\%$ slowdown for typical optimizations
- Event files are efficiently buffered and written
- Writer is properly closed after optimization completes

25 Variable Type (var_type) Implementation

25.1 Overview

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

25.2 Supported Variable Types

SpotOptim supports three main data types:

25.2.1 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)

25.2.2 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)

25.2.3 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)

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

25.3 Implementation Details

25.3.1 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

25.3.2 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

25.4 Usage Examples

25.5 5. Example Usage

```
import numpy as np
from spotoptim import SpotOptim

# Example 1: All float variables (default)
opt1 = SpotOptim(
    fun=lambda x: np.sum(x**2),
    lower=np.array([0, 0, 0]),
    upper=np.array([10, 10, 10])
    # var_type defaults to ["float", "float", "float"]
)
```

25.5.1 Example 2: Pure Integer Optimization

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

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

opt = SpotOptim(
    fun=discrete_func,
    bounds=bounds,
    var_type=var_type,
    max_iter=20,
    seed=42
)

result = opt.optimize()
# result.x will have integer values like [1.0, -2.0]
```

25.5.2 Example 3: Categorical (Factor) Variables

```
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,
    seed=42
)

result = opt.optimize()
# result.x will be integers like [0.0, 1.0] representing categories
```

25.5.3 Example 4: Mixed Variable Types

```
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"]

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

result = opt.optimize()
```

```
# result.x[0]: continuous float like 0.123
# result.x[1]: integer like 5.0
# result.x[2]: integer category like 0.0
```

25.6 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.

25.7 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

25.8 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

References