Hyperparameter Tuning Cookbook

A guide for scikit-learn, PyTorch, river, and spotPython

Thomas Bartz-Beielstein

Dec 30, 2023

Table of contents

Preface

This document provides a comprehensive guide to hyperparameter tuning using spotPython for scikit-learn, scipy-optimize, River, and PyTorch. The first part introduces fundamental ideas from optimization. The second part discusses numerical issues and introduces spotPython's surrogate model-based optimization process. The thirs part focuses on hyperparameter tuning. Several case studies are presented, including hyperparameter tuning for sklearn models such as Support Vector Classification, Random Forests, Gradient Boosting (XGB), and K-nearest neighbors (KNN), as well as a Hoeffding Adaptive Tree Regressor from river. The integration of spotPython into the PyTorch and PyTorch Lightning training workflow is also discussed. With a hands-on approach and step-by-step explanations, this cookbook serves as a practical starting point for anyone interested in hyperparameter tuning with Python. Highlights include the interplay between Tensorboard, PyTorch Lightning, spotPython, spotRiver, and River. This publication is under development, with updates available on the corresponding webpage.

! Important: This book is still under development.

The most recent version of this book is available at https://sequential-parameter-optimization.github.io/Hyperparameter-Tuning-Cookbook/

Book Structure

This document is structured in three parts. The first part presents an introduction to optimization. The second part describes numerical methods, and the third part presents hyperparameter tuning.

Hyperparameter Tuning Reference

- The open access book Bartz et al. (2022) provides a comprehensive overview of hyperparameter tuning. It can be downloaded from https://link.springer.com/book/10.1007/978-981-19-5170-1.

> **Note**
>
> The `.ipynb` notebook (Bartz-Beielstein 2023) is updated regularly and reflects updates and changes in the `spotPython` package. It can be downloaded from https://github.com/sequential-parameter-optimization/spotPython/blob/main/notebooks/14_spot_ray_hpt_torch_cifar10.ipynb.

Software Used in this Book

scikit-learn is a Python module for machine learning built on top of SciPy and is distributed under the 3-Clause BSD license. The project was started in 2007 by David Cournapeau as a Google Summer of Code project, and since then many volunteers have contributed.

PyTorch is an optimized tensor library for deep learning using GPUs and CPUs. Lightning is a lightweight PyTorch wrapper for high-performance AI research. It allows you to decouple the research from the engineering.

River is a Python library for online machine learning. It is designed to be used in real-world environments, where not all data is available at once, but streaming in.

spotPython ("Sequential Parameter Optimization Toolbox in Python") is the Python version of the well-known hyperparameter tuner SPOT, which has been developed in the R programming environment for statistical analysis for over a decade. The related open-access book is available here: Hyperparameter Tuning for Machine and Deep Learning with R—A Practical Guide.

spotRiver provides an interface between spotPython and River.

Citation

If this document has been useful to you and you wish to cite it in a scientific publication, please refer to the following paper, which can be found on arXiv: https://arxiv.org/abs/2307.10262.

```
@ARTICLE{bart23iArXiv,
      author = {{Bartz-Beielstein}, Thomas},
      title = "{Hyperparameter Tuning Cookbook:
          A guide for scikit-learn, PyTorch, river, and spotPython}",
     journal = {arXiv e-prints},
    keywords = {Computer Science - Machine Learning,
      Computer Science - Artificial Intelligence, 90C26, I.2.6, G.1.6},
         year = 2023,
```

```
        month = jul,
          eid = {arXiv:2307.10262},
        pages = {arXiv:2307.10262},
          doi = {10.48550/arXiv.2307.10262},
archivePrefix = {arXiv},
       eprint = {2307.10262},
 primaryClass = {cs.LG},
       adsurl = {https://ui.adsabs.harvard.edu/abs/2023arXiv230710262B},
      adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}
```

Part I

Optimization

1 Introduction: Optimization

1.1 Optimization, 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

1.2 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

1.2.1 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.

1.2.2 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

1.2.3 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**

1.2.4 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

1.2.5 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

1.2.6 Example: 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

1.2.7 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

> **!** Goals
>
> - How to choose models and optimizers for solving real-world problems
> - How to use simulation to understand and improve processes

2 Aircraft Wing Weight Example

2.1 AWWE Equation

- Example from Forrester et al.
- Understand the **weight** of an unpainted light aircraft wing as a function of nine design and operational parameters:

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

2.2 AWWE Parameters and Equations (Part 1)

Table 2.1: 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
A	Aspect ratio	7.52	6	10
Λ	Quarter-chord sweep (deg)	0	-10	10
q	Dynamic pressure at cruise (lb/ft^2)	34	16	45
λ	Taper ratio	0.672	0.5	1
R_{tc}	Aerofoil thickness to chord ratio	0.12	0.08	0.18
N_z	Ultimate load factor	3.8	2.5	6
W_{dg}	Flight design gross weight (lb)	2000	1700	2500
W_p	paint weight (lb/ft^2)	0.064	0.025	0.08

The study begins with a baseline Cessna C172 Skyhawk Aircraft as its reference point. It aims to investigate the impact of wing area and fuel weight on the overall weight of the aircraft. Two crucial parameters in this analysis are the aspect ratio (A), defined as the ratio of the

wing's length to the average chord (thickness of the airfoil), and the taper ratio (λ), which represents the ratio of the maximum to the minimum thickness of the airfoil or the maximum to minimum chord.

It's important to note that the equation used in this context is not a computer simulation but will be treated as one for the purpose of illustration. This approach involves employing a true mathematical equation, even if it's considered unknown, as a useful tool for generating realistic settings to test the methodology. The functional form of this equation was derived by "calibrating" known physical relationships to curves obtained from existing aircraft data, as referenced in Raymer 2012. 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.

2.3 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 2.1. To map values from the interval $[a, b]$ to the interval $[0, 1]$, the following formula can be employed:

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

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

$$g(y) = a + (b - a)y$$

can be used.

```
import numpy as np

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)
```

2.4 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.

```
import numpy as np
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)
```

```
[(0.0, 0.0),
 (0.5, 0.0),
 (1.0, 0.0),
 (0.0, 0.5),
 (0.5, 0.5),
 (1.0, 0.5),
 (0.0, 1.0),
 (0.5, 1.0),
 (1.0, 1.0)]
```

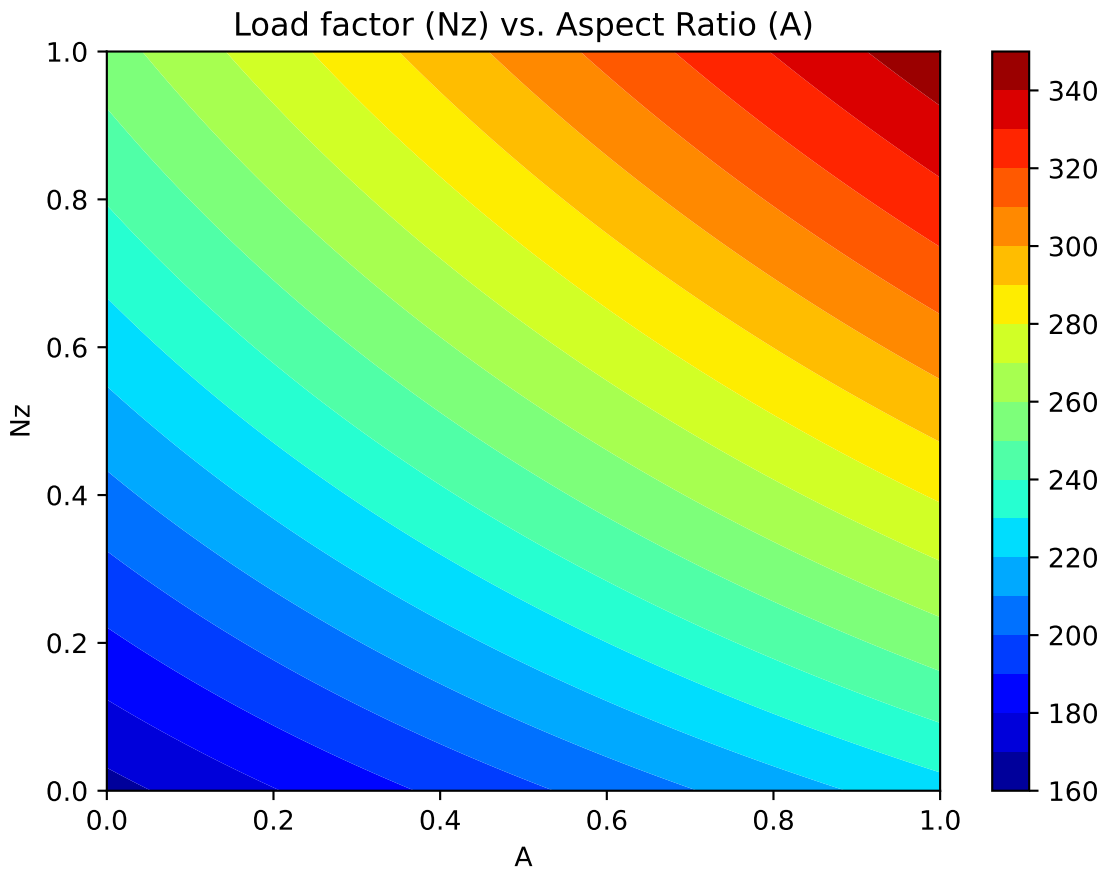
The coding used to transform inputs from natural units is largely a matter of taste, so long as it's easy to undo for reporting back on original scales

```
%matplotlib inline
import matplotlib.pyplot as plt
# plt.style.use('seaborn-white')
import numpy as np
x = np.linspace(0, 1, 100)
y = np.linspace(0, 1, 100)
X, Y = np.meshgrid(x, y)
```

2.5 Plot 1: Load Factor (N_z) and Aspect Ratio (A)

We will vary N_z and A, with other inputs fixed at their baseline values.

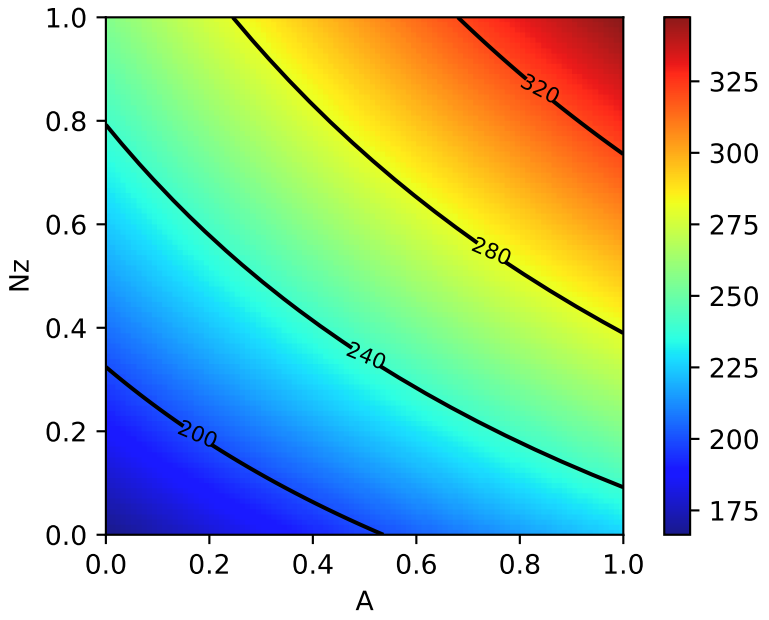
```
z = wingwt(A = X, Nz = Y)
fig = plt.figure(figsize=(7., 5.))
plt.contourf(X, Y, z, 20, cmap='jet')
plt.xlabel("A")
plt.ylabel("Nz")
plt.title("Load factor (Nz) vs. Aspect Ratio (A)")
plt.colorbar()
plt.show()
```



Contour plots can be refined, e.g., by adding explicit contour lines as shown in the following figure.

```
contours = plt.contour(X, Y, z, 4, colors='black')
plt.clabel(contours, inline=True, fontsize=8)
plt.xlabel("A")
plt.ylabel("Nz")

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

```
<matplotlib.colorbar.Colorbar at 0x137e06b50>
```



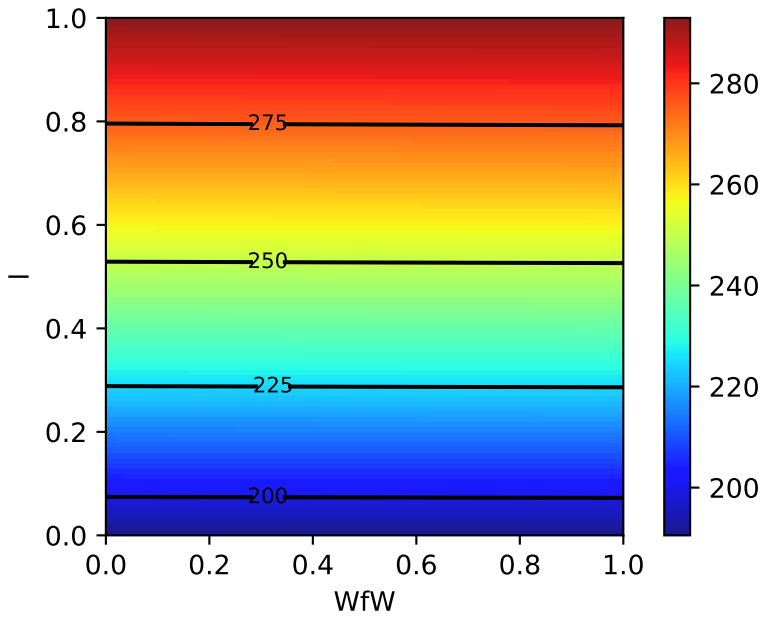
The interpretation of the AWWE plot can be summarized as follows:

- 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.

2.6 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 $\lambda \times W_{fw}$

2.7 The Big Picture: Combining all Variables

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

```
import math

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`.

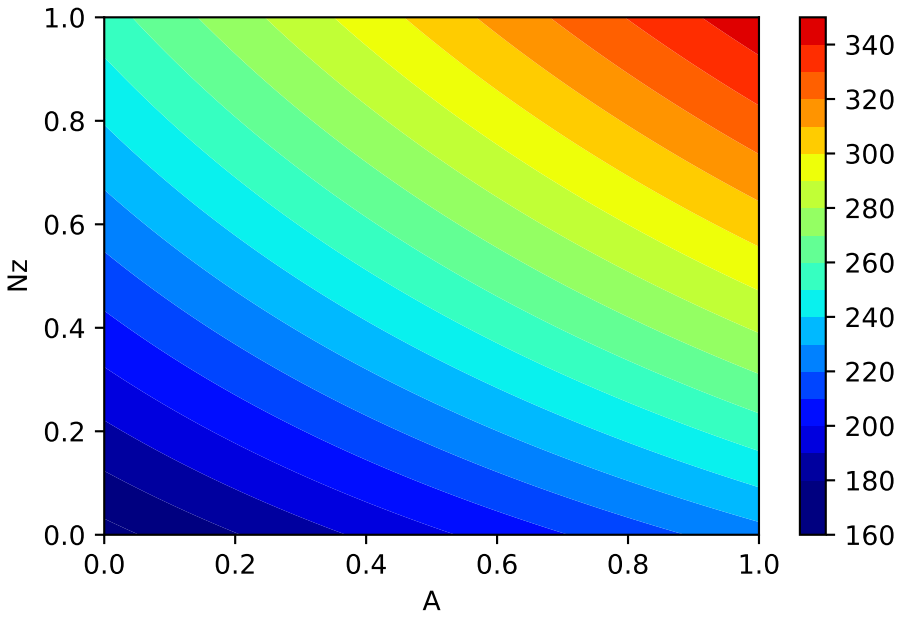
```
p = 19
Zlab[p]
```

```
['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()
```

```
<matplotlib.colorbar.Colorbar at 0x168918a90>
```



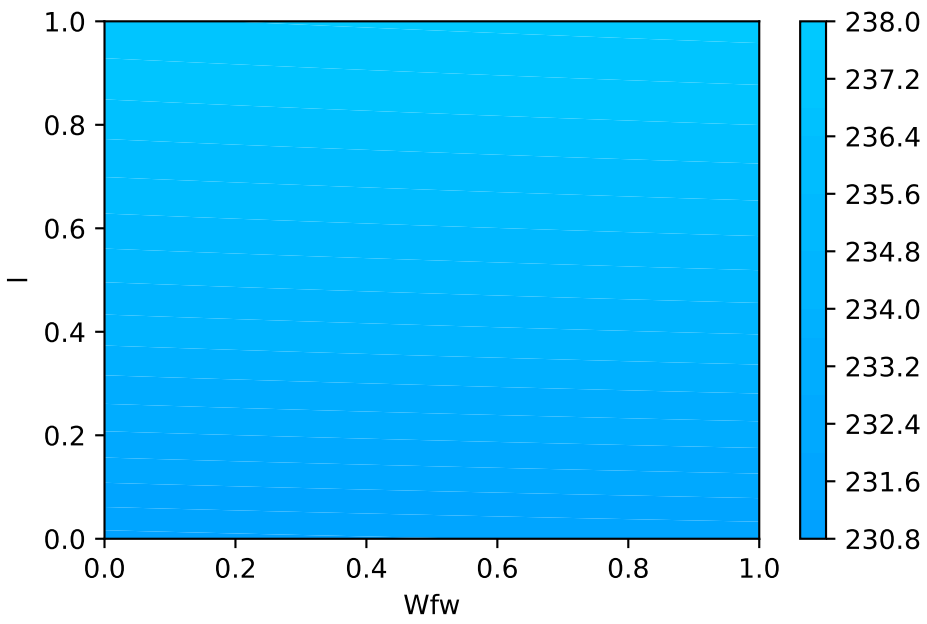
- Let's plot the second example, taper ratio λ and fuel weight W_{fw}
- This is combination `11`:

```
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()
```

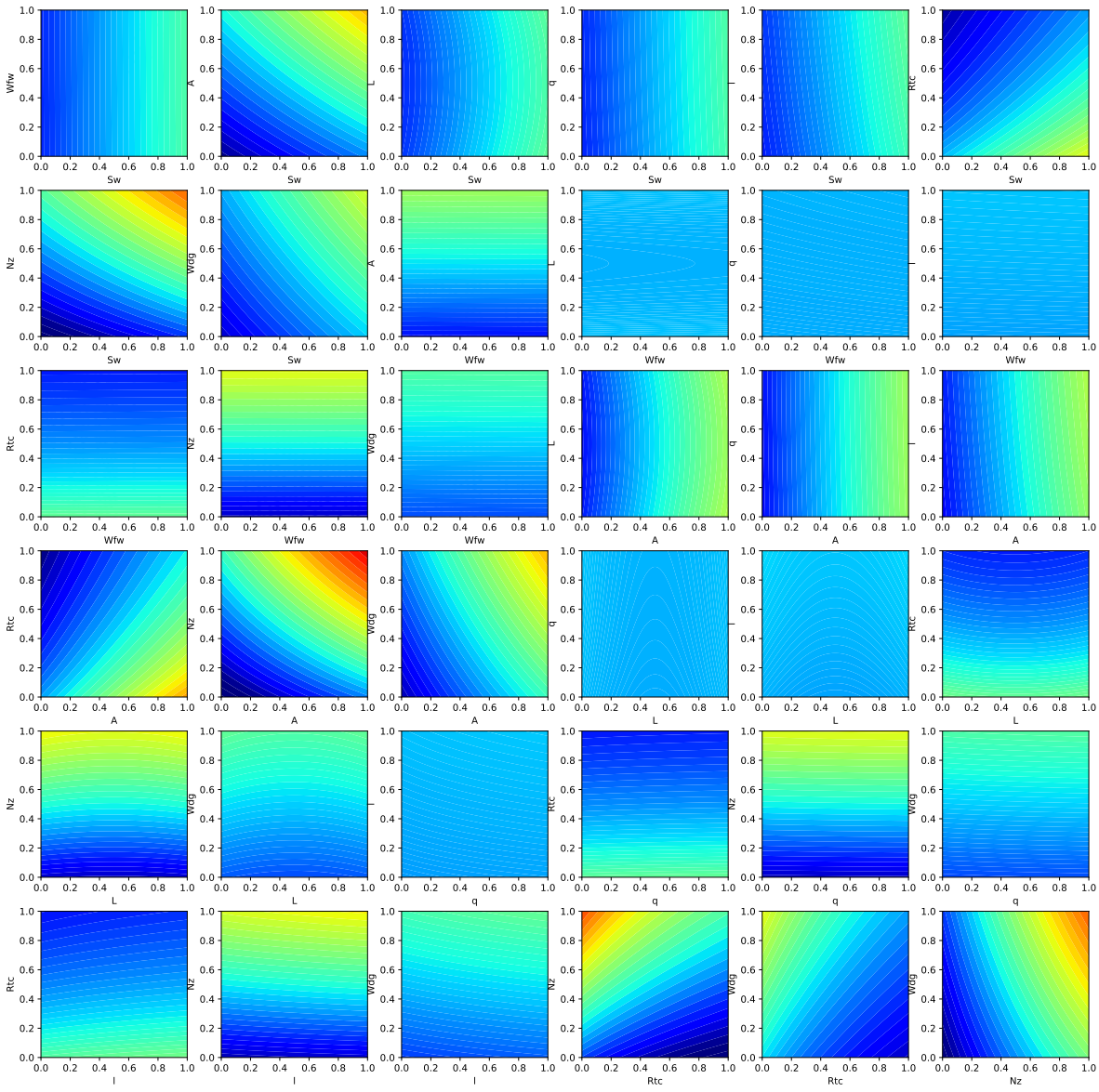
```
<matplotlib.colorbar.Colorbar at 0x168a58ad0>
```



- 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.

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

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="0",
                 )
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()
```



2.8 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.

2.9 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

2.10 Exercise

2.10.1 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.

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

3 Introduction to `scipy.optimize`

SciPy provides algorithms for optimization, integration, interpolation, eigenvalue problems, algebraic equations, differential equations, statistics and many other classes of problems. SciPy is a collection of mathematical algorithms and convenience functions built on NumPy. It adds significant power to Python by providing the user with high-level commands and classes for manipulating and visualizing data.

SciPy optimize provides functions for minimizing (or maximizing) objective functions, possibly subject to constraints. It includes solvers for nonlinear problems (with support for both local and global optimization algorithms), linear programing, constrained and nonlinear least-squares, root finding, and curve fitting.

In this notebook, we will learn how to use the `scipy.optimize` module to solve optimization problems. See: https://docs.scipy.org/doc/scipy/tutorial/optimize.html

> **Note**
>
> - This content is based on information from the scipy.optimize package.
> - The `scipy.optimize` package provides several commonly used optimization algorithms. A detailed listing is available in `scipy.optimize` (can also be found by `help(scipy.optimize)`).

Common functions and objects, shared across different SciPy optimize solvers, are shown in Table 3.1.

Table 3.1: Common functions and objects, shared across different SciPy optimize solvers

Function or Object	Description
show_options([solver, method, disp])	Show documentation for additional options of optimization solvers.
OptimizeResult	Represents the optimization result.
OptimizeWarning	Warning issued by solvers.

We will introduce unconstrained minimization of multivariate scalar functions in this chapter. The minimize function provides a common interface to unconstrained and constrained minimization algorithms for multivariate scalar functions in `scipy.optimize`. To demonstrate

the minimization function, consider the problem of minimizing the Rosenbrock function of N variables:

$$f(J) = \sum_{i=1}^{N-1} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2$$

The minimum value of this function is 0, which is achieved when (x_i = 1).

Note that the Rosenbrock function and its derivatives are included in `scipy.optimize`. The implementations shown in the following sections provide examples of how to define an objective function as well as its Jacobian and Hessian functions. Objective functions in `scipy.optimize` expect a numpy array as their first parameter, which is to be optimized and must return a float value. The exact calling signature must be `f(x, *args)`, where `x` represents a numpy array, and `args` is a tuple of additional arguments supplied to the objective function.

3.1 Derivative-free Optimization Algorithms

Section 3.1.1 and Section 3.1.2 present two approaches that do not need gradient information to find the minimum. They use function evaluations to find the minimum.

3.1.1 Nelder-Mead Simplex Algorithm

`method='Nelder-Mead'`: In the example below, the `minimize` routine is used with the *Nelder-Mead* simplex algorithm (selected through the `method` parameter):

```
import numpy as np
from scipy.optimize import minimize

def rosen(x):
    """The Rosenbrock function"""
    return sum(100.0 * (x[1:] - x[:-1]**2.0)**2.0 + (1 - x[:-1])**2.0)

x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])
res = minimize(rosen, x0, method='nelder-mead',
               options={'xatol': 1e-8, 'disp': True})

print(res.x)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 339
         Function evaluations: 571
[1. 1. 1. 1. 1.]
```

The simplex algorithm is probably the simplest way to minimize a well-behaved function. It requires only function evaluations and is a good choice for simple minimization problems. However, because it does not use any gradient evaluations, it may take longer to find the minimum.

3.1.2 Powell's Method

Another optimization algorithm that needs only function calls to find the minimum is *Powell*'s method, which can be selected by setting the `method` parameter to `'powell'` in the `minimize` function.

To demonstrate how to supply additional arguments to an objective function, let's consider minimizing the Rosenbrock function with an additional scaling factor a and an offset b:

$$f(J, a, b) = \sum_{i=1}^{N-1} a(x_{i+1} - x_i^2)^2 + (1 - x_i)^2 + b$$

You can achieve this using the `minimize` routine with the example parameters $a = 0.5$ and $b = 1$:

```
def rosen_with_args(x, a, b):
    """The Rosenbrock function with additional arguments"""
    return sum(a * (x[1:] - x[:-1]**2.0)**2.0 + (1 - x[:-1])**2.0) + b

x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])
res = minimize(rosen_with_args, x0, method='nelder-mead',
               args=(0.5, 1.), options={'xatol': 1e-8, 'disp': True})

print(res.x)
```

```
Optimization terminated successfully.
         Current function value: 1.000000
         Iterations: 319
         Function evaluations: 525
[1.         1.         1.         1.         0.99999999]
```

As an alternative to using the `args` parameter of `minimize`, you can wrap the objective function in a new function that accepts only `x`. This approach is also useful when it is necessary to pass additional parameters to the objective function as keyword arguments.

```
def rosen_with_args(x, a, *, b):  # b is a keyword-only argument
    return sum(a * (x[1:] - x[:-1]**2.0)**2.0 + (1 - x[:-1])**2.0) + b

def wrapped_rosen_without_args(x):
    return rosen_with_args(x, 0.5, b=1.)  # pass in `a` and `b`

x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])
res = minimize(wrapped_rosen_without_args, x0, method='nelder-mead',
               options={'xatol': 1e-8,})

print(res.x)
```

```
[1.         1.         1.         1.         0.99999999]
```

Another alternative is to use `functools.partial`.

```
from functools import partial

partial_rosen = partial(rosen_with_args, a=0.5, b=1.)
res = minimize(partial_rosen, x0, method='nelder-mead',
               options={'xatol': 1e-8,})

print(res.x)
```

```
[1.         1.         1.         1.         0.99999999]
```

3.2 Gradient-based optimization algorithms

3.2.1 An Introductory Example: Broyden-Fletcher-Goldfarb-Shanno Algorithm (BFGS)

This section introduces an optimization algorithm that uses gradient information to find the minimum. The Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm (selected by setting `method='BFGS'`) is an optimization algorithm that aims to converge quickly to the solution. This algorithm uses the gradient of the objective function. If the gradient is not provided by the

user, it is estimated using first-differences. The BFGS method typically requires fewer function calls compared to the simplex algorithm, even when the gradient needs to be estimated.

Example: BFGS

To demonstrate the BFGS algorithm, let's use the Rosenbrock function again. The gradient of the Rosenbrock function is a vector described by the following mathematical expression:

$$\frac{\partial f}{\partial x_j} = \sum_{i=1}^{N} 200(x_i - x_{i-1}^2)(\delta_{i,j} - 2x_{i-1}\delta_{i-1,j}) - 2(1 - x_{i-1})\delta_{i-1,j} \tag{3.1}$$

$$= 200(x_j - x_{j-1}^2) - 400x_j(x_{j+1} - x_j^2) - 2(1 - x_j) \tag{3.2}$$

This expression is valid for interior derivatives, but special cases are:

$$\frac{\partial f}{\partial x_0} = -400x_0(x_1 - x_0^2) - 2(1 - x_0)$$

$$\frac{\partial f}{\partial x_{N-1}} = 200(x_{N-1} - x_{N-2}^2)$$

Here's a Python function that computes this gradient:

```
def rosen_der(x):
    xm = x[1:-1]
    xm_m1 = x[:-2]
    xm_p1 = x[2:]
    der = np.zeros_like(x)
    der[1:-1] = 200*(xm-xm_m1**2) - 400*(xm_p1 - xm**2)*xm - 2*(1-xm)
    der[0] = -400*x[0]*(x[1]-x[0]**2) - 2*(1-x[0])
    der[-1] = 200*(x[-1]-x[-2]**2)
    return der
```

You can specify this gradient information in the minimize function using the jac parameter as illustrated below:

```
res = minimize(rosen, x0, method='BFGS', jac=rosen_der,
               options={'disp': True})

print(res.x)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
```

```
         Iterations: 25
         Function evaluations: 30
         Gradient evaluations: 30
[1.00000004 1.0000001  1.00000021 1.00000044 1.00000092]
```

3.2.2 Background and Basics for Gradient-based Optimization

3.2.3 Gradient

The gradient $\nabla f(J)$ for a scalar function $f(J)$ with n different variables is defined by its partial derivatives:

$$\nabla f(J) = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n}\right]$$

3.2.4 Jacobian Matrix

The Jacobian matrix $J(J)$ for a vector-valued function $F(J) = [f_1(J), f_2(J), \dots, f_m(J)]$ is defined as:

$$J(J) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

It consists of the first order partial derivatives and gives therefore an overview about the gradients of a vector valued function.

i Example: Jacobian matrix

Consider a vector-valued function $f : \mathbb{R}^2 \to \mathbb{R}^3$ defined as follows:

$$f(J) = \begin{bmatrix} x_1^2 + 2x_2 \\ 3x_1 - \sin(x_2) \\ e^{x_1 + x_2} \end{bmatrix}$$

Let's compute the partial derivatives and construct the Jacobian matrix:
$\frac{\partial f_1}{\partial x_1} = 2x_1, \quad \frac{\partial f_1}{\partial x_2} = 2$
$\frac{\partial f_2}{\partial x_1} = 3, \quad \frac{\partial f_2}{\partial x_2} = -\cos(x_2)$
$\frac{\partial f_3}{\partial x_1} = e^{x_1+x_2}, \quad \frac{\partial f_3}{\partial x_2} = e^{x_1+x_2}$
So, the Jacobian matrix is:

$$J(J) = \begin{bmatrix} 2x_1 & 2 \\ 3 & -\cos(x_2) \\ e^{x_1+x_2} & e^{x_1+x_2} \end{bmatrix}$$

This Jacobian matrix provides information about how small changes in the input variables x_1 and x_2 affect the corresponding changes in each component of the output vector.

3.2.5 Hessian Matrix

The Hessian matrix $H(J)$ for a scalar function $f(J)$ is defined as:

$$H(J) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

So, the Hessian matrix consists of the second order dervatives of the function. It provides information about the local curvature of the function with respect to changes in the input variables.

Example: Hessian matrix

Consider a scalar-valued function:

$$f(J) = x_1^2 + 2x_2^2 + \sin(x_1 x_2)$$

The Hessian matrix of this scalar-valued function is the matrix of its second-order partial derivatives with respect to the input variables:

$$H(J) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} \end{bmatrix}$$

Let's compute the second-order partial derivatives and construct the Hessian matrix:

$$\frac{\partial^2 f}{\partial x_1^2} = 2 + \cos(x_1 x_2) x_2^2 \tag{3.3}$$

$$\frac{\partial^2 f}{\partial x_1 \partial x_2} = 2x_1 x_2 \cos(x_1 x_2) - \sin(x_1 x_2) \tag{3.4}$$

$$\frac{\partial^2 f}{\partial x_2 \partial x_1} = 2x_1 x_2 \cos(x_1 x_2) - \sin(x_1 x_2) \tag{3.5}$$

$$\frac{\partial^2 f}{\partial x_2^2} = 4x_2^2 + \cos(x_1 x_2) x_1^2 \tag{3.6}$$

So, the Hessian matrix is:

$$H(J) = \begin{bmatrix} 2 + \cos(x_1 x_2) x_2^2 & 2x_1 x_2 \cos(x_1 x_2) - \sin(x_1 x_2) \\ 2x_1 x_2 \cos(x_1 x_2) - \sin(x_1 x_2) & 4x_2^2 + \cos(x_1 x_2) x_1^2 \end{bmatrix}$$

3.2.6 Gradient for Optimization

In optimization, the goal is to find the minimum or maximum of a function. Gradient-based optimization methods utilize information about the gradient (or derivative) of the function to guide the search for the optimal solution. This is particularly useful when dealing with complex, high-dimensional functions where an exhaustive search is impractical.

The gradient descent method can be divided in the following steps:

- **Initialize:** start with an initial guess for the parameters of the function to be optimized.
- **Compute Gradient:** Calculate the gradient (partial derivatives) of the function with respect to each parameter at the current point. The gradient indicates the direction of the steepest increase in the function.
- **Update Parameters:** Adjust the parameters in the opposite direction of the gradient, scaled by a learning rate. This step aims to move towards the minimum of the function:
  - $x_{k+1} = x_k - \alpha \times \nabla f(x_k)$
  - x_x is current parameter vector or point in the parameter space.
  - α is the learning rate, a positive scalar that determines the step size in each iteration.
  - $\nabla f(x)$ is the gradient of the objective function.
- **Iterate:** Repeat the above steps until convergence or a predefined number of iterations. Convergence is typically determined when the change in the function value or parameters becomes negligible.

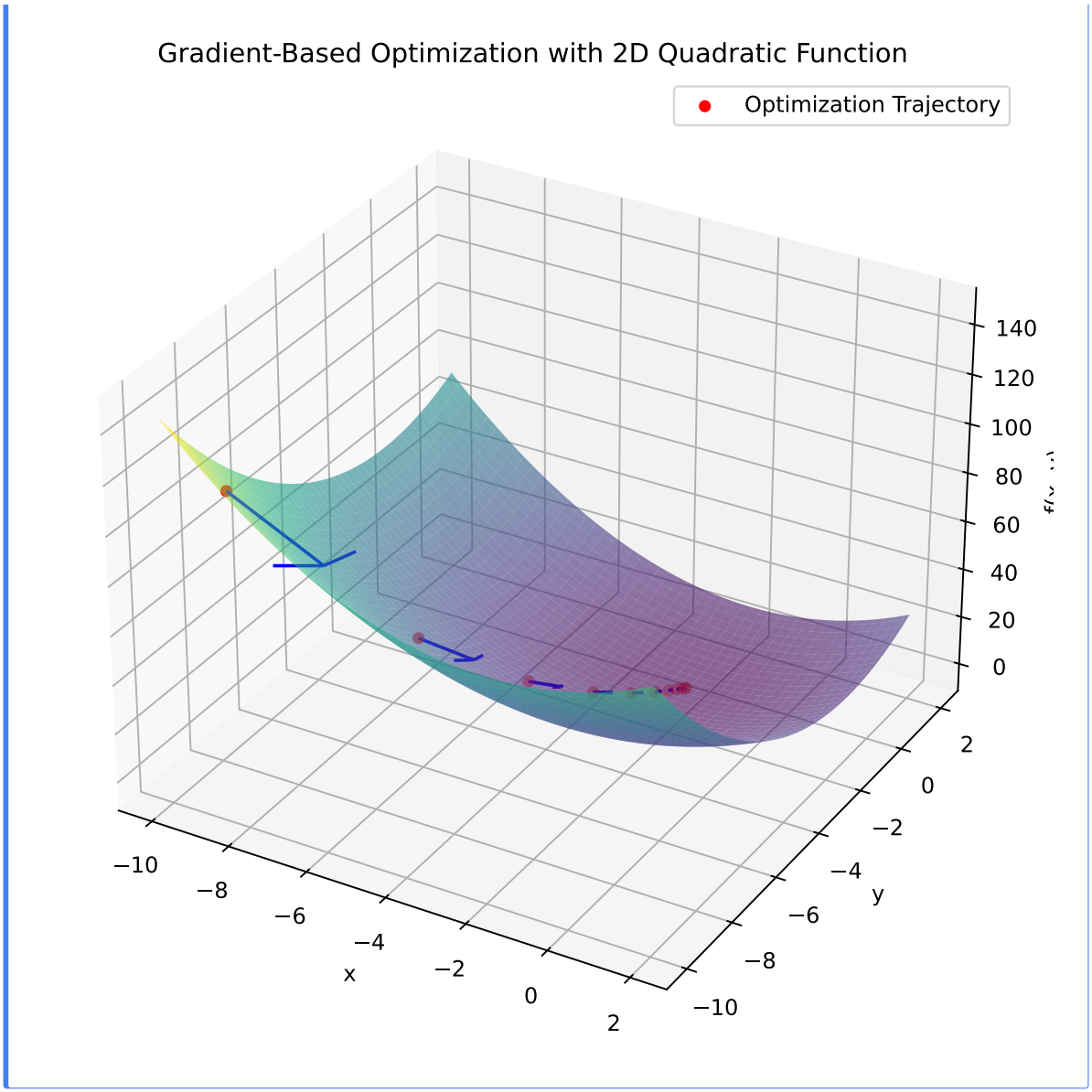
i Example: Gradient Descent

Let's consider a simple quadratic function as an example:

$$f(x) = x^2 + 4x + y^2 + 2y + 4.$$

We'll use gradient descent to find the minimum of this function.

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Define the quadratic function
def quadratic_function(x, y):
    return x**2 + 4*x + y**2 + 2*y + 4

# Define the gradient of the quadratic function
def gradient_quadratic_function(x, y):
    grad_x = 2*x + 4
    grad_y = 2*y + 2
    return np.array([grad_x, grad_y])

# Gradient Descent for optimization in 2D
def gradient_descent(initial_point, learning_rate, num_iterations):
    points = [np.array(initial_point)]

    for _ in range(num_iterations):
        current_point = points[-1]
        gradient = gradient_quadratic_function(*current_point)
        new_point = current_point - learning_rate * gradient

        points.append(new_point)

    return points

# Visualization of optimization process with 3D surface and consistent arrow sizes
def plot_optimization_process_3d_consistent_arrows(points):
    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')

    x_vals = np.linspace(-10, 2, 100)
    y_vals = np.linspace(-10, 2, 100)
    X, Y = np.meshgrid(x_vals, y_vals)
    Z = quadratic_function(X, Y)

    ax.plot_surface(X, Y, Z, cmap='viridis', alpha=0.6)
    ax.scatter(*zip(*points), [quadratic_function(*p) for p in points], c='red', label='

    for i in range(len(points) - 1):
        x, y = points[i]
        dx, dy = points[i + 1] - points[i]
        dz = quadratic_function(*(points[i + 1])) - quadratic_function(*points[i])
        gradient_length = 0.5

        ax.quiver(x, y, quadratic_function(*points[i]), dx, dy, dz, color='blue', length

    ax.set_title('Gradient-Based Optimization with 2D Quadratic Function')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('f(x, y)')
```



3.2.7 Newton Method

Initialization: Start with an initial guess for the optimal solution: x_0.

Iteration: Repeat the following three steps until convergence or a predefined stopping criterion is met:

1) Calculate the gradient (∇) and the Hessian matrix (∇^2) of the objective function at the

current point:

$$\nabla f(x_k) \quad \text{and} \quad \nabla^2 f(x_k)$$

2) Update the current solution using the Newton-Raphson update formula

$$x_{k+1} = x_k - [\nabla^2 f(x_k)]^{-1} \nabla f(x_k),$$

where

- $\nabla f(x_k)$ is the gradient (first derivative) of the objective function with respect to the variable x, evaluated at the current solution x_k.
- $\nabla^2 f(x_k)$: The Hessian matrix (second derivative) of the objective function with respect to x, evaluated at the current solution x_k.
- x_k: The current solution or point in the optimization process.
- $\nabla^2 f(x_k)]^{-1}$: The inverse of the Hessian matrix at the current point, representing the approximation of the curvature of the objective function.
- x_{k+1}: The updated solution or point after applying the Newton-Raphson update.

3) Check for convergence.

i Example: Newton Method

We want to optimize the Rosenbrock function and use the Hessian and the Jacobian (which is equal to the gradient vector for scalar objective function) to the `minimize` function.

```python
def rosenbrock(x):
    return 100 * (x[1] - x[0]**2)**2 + (1 - x[0])**2

def rosenbrock_gradient(x):
    dfdx0 = -400 * x[0] * (x[1] - x[0]**2) - 2 * (1 - x[0])
    dfdx1 = 200 * (x[1] - x[0]**2)
    return np.array([dfdx0, dfdx1])

def rosenbrock_hessian(x):
    d2fdx0 = 1200 * x[0]**2 - 400 * x[1] + 2
    d2fdx1 = -400 * x[0]
    return np.array([[d2fdx0, d2fdx1], [d2fdx1, 200]])

def classical_newton_optimization_2d(initial_guess, tol=1e-6, max_iter=100):
    x = initial_guess.copy()

    for i in range(max_iter):
        gradient = rosenbrock_gradient(x)
        hessian = rosenbrock_hessian(x)

        # Solve the linear system H * d = -g for d
        d = np.linalg.solve(hessian, -gradient)

        # Update x
        x += d

        # Check for convergence
        if np.linalg.norm(gradient, ord=np.inf) < tol:
            break

    return x

# Initial guess
initial_guess_2d = np.array([0.0, 0.0])

# Run classical Newton optimization for the 2D Rosenbrock function
result_2d = classical_newton_optimization_2d(initial_guess_2d)

# Print the result
print("Optimal solution:", result_2d)
print("Objective value:", rosenbrock(result_2d))
```

```
Optimal solution: [1. 1.]
Objective value: 0.0
```

3.2.8 BFGS-Algorithm

BFGS is an optimization algorithm designed for unconstrained optimization problems. It belongs to the class of quasi-Newton methods and is known for its efficiency in finding the minimum of a smooth, unconstrained objective function.

3.2.9 Procedure:

1. **Initialization:**
   - Start with an initial guess for the parameters of the objective function.
   - Initialize an approximation of the Hessian matrix (inverse) denoted by H.

2. **Iterative Update:**
   - At each iteration, compute the gradient vector at the current point.
   - Update the parameters using the BFGS update formula, which involves the inverse Hessian matrix approximation, the gradient, and the difference in parameter vectors between successive iterations:

$$x_{k+1} = x_k - H_k^{-1} \nabla f(x_k).$$

   - Update the inverse Hessian approximation using the BFGS update formula for the inverse Hessian.

$$H_{k+1} = H_k + \frac{\Delta x_k \Delta x_k^T}{\Delta x_k^T \Delta g_k} - \frac{H_k g_k g_k^T H_k}{g_k^T H_k g_k},$$

   where:
   - x_k and x_{k+1} are the parameter vectors at the current and updated iterations, respectively.
   - $\nabla f(x_k)$ is the gradient vector at the current iteration.
   - $\Delta x_k = x_{k+1} - x_k$ is the change in parameter vectors.
   - $\Delta g_k = \nabla f(x_{k+1}) - \nabla f(x_k)$ is the change in gradient vectors.

3. **Convergence:**
   - Repeat the iterative update until the optimization converges. Convergence is typically determined by reaching a sufficiently low gradient or parameter change.

Example: BFGS for Rosenbrock

```python
import numpy as np
from scipy.optimize import minimize

# Define the 2D Rosenbrock function
def rosenbrock(x):
    return (1 - x[0])**2 + 100 * (x[1] - x[0]**2)**2

# Initial guess
initial_guess = np.array([0.0, 0.0])

# Minimize the Rosenbrock function using BFGS
minimize(rosenbrock, initial_guess, method='BFGS')
```

```
 message: Optimization terminated successfully.
 success: True
  status: 0
     fun: 2.843987518235081e-11
       x: [ 1.000e+00  1.000e+00]
     nit: 19
     jac: [ 3.987e-06 -2.844e-06]
hess_inv: [[ 4.948e-01  9.896e-01]
           [ 9.896e-01  1.984e+00]]
    nfev: 72
    njev: 24
```

3.2.10 Visualization BFGS for Rosenbrock

A visualization of the BFGS search process on Rosenbrock's function can be found here: https://upload.wikimedia.org/wikipedia/de/f/ff/Rosenbrock-bfgs-animation.gif

Tasks

- In which situations is it possible to use algorithms like BFGS, but not the classical Newton method?
- Investigate the Newton-CG method
- Use an objective function of your choice and apply Newton-CG
- Compare the Newton-CG method with the BFGS. What are the similarities and differences between the two algorithms?

3.3 Gradient- and Hessian-based optimization algorithms

Section 3.3.1 presents an optimization algorithm that uses gradient and Hessian information to find the minimum. Section 3.3.2 presents an optimization algorithm that uses gradient and Hessian information to find the minimum. Section 3.3.3 presents an optimization algorithm that uses gradient and Hessian information to find the minimum.

The methods Newton-CG, trust-ncg and trust-krylov are suitable for dealing with large-scale problems (problems with thousands of variables). That is because the conjugate gradient algorithm approximately solve the trust-region subproblem (or invert the Hessian) by iterations without the explicit Hessian factorization. Since only the product of the Hessian with an arbitrary vector is needed, the algorithm is specially suited for dealing with sparse Hessians, allowing low storage requirements and significant time savings for those sparse problems.

3.3.1 Newton-Conjugate-Gradient Algorithm

Newton-Conjugate Gradient algorithm is a modified Newton's method and uses a conjugate gradient algorithm to (approximately) invert the local Hessian.

3.3.2 Trust-Region Newton-Conjugate-Gradient Algorithm

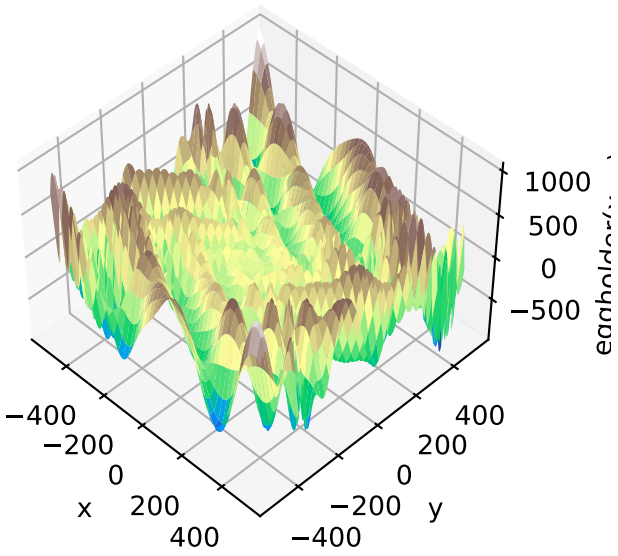
3.3.3 Trust-Region Truncated Generalized Lanczos / Conjugate Gradient Algorithm

3.4 Global Optimization

Global optimization aims to find the global minimum of a function within given bounds, in the presence of potentially many local minima. Typically, global minimizers efficiently search the parameter space, while using a local minimizer (e.g., minimize) under the hood. SciPy contains a number of good global optimizers. Here, we'll use those on the same objective function, namely the (aptly named) eggholder function:

```
def eggholder(x):
    return (-(x[1] + 47) * np.sin(np.sqrt(abs(x[0]/2 + (x[1]  + 47))))
            -x[0] * np.sin(np.sqrt(abs(x[0] - (x[1]  + 47)))))

bounds = [(-512, 512), (-512, 512)]
```

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

x = np.arange(-512, 513)
y = np.arange(-512, 513)
xgrid, ygrid = np.meshgrid(x, y)
xy = np.stack([xgrid, ygrid])

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.view_init(45, -45)
ax.plot_surface(xgrid, ygrid, eggholder(xy), cmap='terrain')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('eggholder(x, y)')
plt.show()
```



We now use the global optimizers to obtain the minimum and the function value at the minimum. We'll store the results in a dictionary so we can compare different optimization results later.

```
from scipy import optimize
results = dict()
```

```
results['shgo'] = optimize.shgo(eggholder, bounds)
results['shgo']
```

```
message: Optimization terminated successfully.
success: True
    fun: -935.3379515605789
   funl: [-9.353e+02]
      x: [ 4.395e+02  4.540e+02]
     xl: [[ 4.395e+02  4.540e+02]]
    nit: 1
   nfev: 45
  nlfev: 40
  nljev: 10
  nlhev: 0
```

```
results['DA'] = optimize.dual_annealing(eggholder, bounds)
results['DA']
```

```
message: ['Maximum number of iteration reached']
success: True
 status: 0
    fun: -956.9182316238251
      x: [ 4.824e+02  4.329e+02]
    nit: 1000
   nfev: 4112
   njev: 37
   nhev: 0
```

All optimizers return an `OptimizeResult`, which in addition to the solution contains information on the number of function evaluations, whether the optimization was successful, and more. For brevity, we won't show the full output of the other optimizers:

```
results['DE'] = optimize.differential_evolution(eggholder, bounds)
results['DE']
```

```
message: Optimization terminated successfully.
success: True
    fun: -935.3379515605704
      x: [ 4.395e+02  4.540e+02]
```

```
     nit: 24
    nfev: 771
     jac: [-1.137e-05 -1.137e-05]
```

`shgo` has a second method, which returns all local minima rather than only what it thinks is the global minimum:

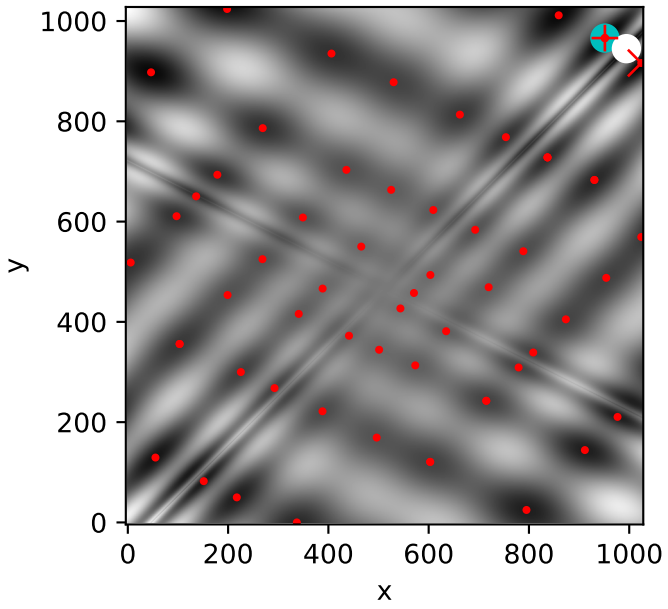
```
results['shgo_sobol'] = optimize.shgo(eggholder, bounds, n=200, iters=5,
                                      sampling_method='sobol')
results['shgo_sobol']
```

```
 message: Optimization terminated successfully.
 success: True
     fun: -959.640662720831
    funl: [-9.596e+02 -9.353e+02 ... -6.591e+01 -6.387e+01]
       x: [ 5.120e+02  4.042e+02]
      xl: [[ 5.120e+02  4.042e+02]
           [ 4.395e+02  4.540e+02]
           ...
           [ 3.165e+01 -8.523e+01]
           [ 5.865e+01 -5.441e+01]]
     nit: 5
    nfev: 3529
   nlfev: 2327
   nljev: 634
   nlhev: 0
```

We'll now plot all found minima on a heatmap of the function:

```
fig = plt.figure()
ax = fig.add_subplot(111)
im = ax.imshow(eggholder(xy), interpolation='bilinear', origin='lower',
               cmap='gray')
ax.set_xlabel('x')
ax.set_ylabel('y')

def plot_point(res, marker='o', color=None):
    ax.plot(512+res.x[0], 512+res.x[1], marker=marker, color=color, ms=10)

plot_point(results['DE'], color='c')  # differential_evolution - cyan
plot_point(results['DA'], color='w')  # dual_annealing.        - white
```

```
# SHGO produces multiple minima, plot them all (with a smaller marker size)
plot_point(results['shgo'], color='r', marker='+')
plot_point(results['shgo_sobol'], color='r', marker='x')
for i in range(results['shgo_sobol'].xl.shape[0]):
    ax.plot(512 + results['shgo_sobol'].xl[i, 0],
            512 + results['shgo_sobol'].xl[i, 1],
            'ro', ms=2)

ax.set_xlim([-4, 514*2])
ax.set_ylim([-4, 514*2])
plt.show()
```



3.4.1 Dual Annealing Optimization

This function implements the Dual Annealing optimization.

3.4.2 Differential Evolution

Differential Evolution is an algorithm used for finding the global minimum of multivariate functions. It is stochastic in nature (does not use gradient methods), and can search large areas

of candidate space, but often requires larger numbers of function evaluations than conventional gradient based techniques.

3.4.3 DIRECT

DIviding RECTangles (DIRECT) is a deterministic global optimization algorithm capable of minimizing a black box function with its variables subject to lower and upper bound constraints by sampling potential solutions in the search space

3.4.4 SHGO

SHGO stands for "simplicial homology global optimization". It is considered appropriate for solving general purpose NLP and blackbox optimization problems to global optimality (low-dimensional problems).

3.4.5 Basin-hopping

Basin-hopping is a two-phase method that combines a global stepping algorithm with local minimization at each step. Designed to mimic the natural process of energy minimization of clusters of atoms, it works well for similar problems with "funnel-like, but rugged" energy landscapes

4 Sequential Parameter Optimization: Using `scipy` Optimizers

As a default optimizer, `spotPython` uses `differential_evolution` from the `scipy.optimize` package. Alternatively, any other optimizer from the `scipy.optimize` package can be used. This chapter describes how different optimizers from the `scipy optimize` package can be used on the surrogate. The optimization algorithms are available from https://docs.scipy.org/doc/scipy/reference/optimize.html

```
import numpy as np
from math import inf
from spotPython.fun.objectivefunctions import analytical
from spotPython.spot import spot
from scipy.optimize import shgo
from scipy.optimize import direct
from scipy.optimize import differential_evolution
from scipy.optimize import dual_annealing
from scipy.optimize import basinhopping
```

4.1 The Objective Function Branin

- The `spotPython` package provides several classes of objective functions.
- We will use an analytical objective function, i.e., a function that can be described by a (closed) formula.
- Here we will use the Branin function. The 2-dim Branin function is

$$y = a * (x2 - b * x1 * *2 + c * x1 - r) * *2 + s * (1 - t) * cos(x1) + s,$$

  where values of a, b, c, r, s and t are: $a = 1, b = 5.1/(4 * pi * *2), c = 5/pi, r = 6, s = 10$ and $t = 1/(8 * pi)$.
- It has three global minima:

  $f(x) = 0.397887$ at $(-\pi, 12.275)$, $(\pi, 2.275)$, and $(9.42478, 2.475)$.

- Input Domain: This function is usually evaluated on the square x1 in [-5, 10] x x2 in [0, 15].

```
from spotPython.fun.objectivefunctions import analytical
lower = np.array([-5,-0])
upper = np.array([10,15])
```

```
fun = analytical(seed=123).fun_branin
```

4.2 The Optimizer

- Differential Evalution from the `scikit.optimize` package, see https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.differential_evolution.html#scipy.optimize.differential_evolution is the default optimizer for the search on the surrogate.
- Other optimiers that are available in `spotPython`:
  - `dual_annealing`
  - `direct`
  - `shgo`
  - `basinhopping`, see https://docs.scipy.org/doc/scipy/reference/optimize.html#global-optimization.
- These can be selected as follows:

  `surrogate_control = "model_optimizer": differential_evolution`
- We will use `differential_evolution`.
- The optimizer can use `1000` evaluations. This value will be passed to the `differential_evolution` method, which has the argument `maxiter` (int). It defines the maximum number of generations over which the entire differential evolution population is evolved, see https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.differential_evolution.html#scipy.optimize.differential_evolution

> **TensorBoard**
>
> Similar to the one-dimensional case, which was introduced in Section Section 7.5, we can use TensorBoard to monitor the progress of the optimization. We will use the same code, only the prefix is different:

```
from spotPython.utils.file import get_experiment_name
from spotPython.utils.init import fun_control_init
from spotPython.utils.file import get_spot_tensorboard_path

PREFIX = "05_DE_"
experiment_name = get_experiment_name(prefix=PREFIX)
print(experiment_name)

fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name))
```

```
05_DE__p040025_2023-12-30_22-55-52
```

```
spot_de = spot.Spot(fun=fun,
                    lower = lower,
                    upper = upper,
                    fun_evals = 20,
                    max_time = inf,
                    seed=125,
                    noise=False,
                    show_models= False,
                    design_control={"init_size": 10},
                    surrogate_control={"n_theta": len(lower),
                                       "model_optimizer": differential_evolution,
                                       "model_fun_evals": 1000,
                                       },
                   fun_control=fun_control)
spot_de.run()
```

```
spotPython tuning: 5.213735995388665 [######----] 55.00%

spotPython tuning: 5.213735995388665 [######----] 60.00%

spotPython tuning: 5.213735995388665 [######----] 65.00%

spotPython tuning: 0.6385092836336437 [#######---] 70.00%

spotPython tuning: 0.40990638686379377 [########--] 75.00%
```

```
spotPython tuning: 0.40990638686379377 [########--] 80.00%

spotPython tuning: 0.40173530984780115 [########--] 85.00%

spotPython tuning: 0.40173530984780115 [#########-] 90.00%

spotPython tuning: 0.40173530984780115 [##########] 95.00%

spotPython tuning: 0.40173530984780115 [##########] 100.00% Done...

<spotPython.spot.spot.Spot at 0x2d14d77d0>
```

4.2.1 TensorBoard

Now we can start TensorBoard in the background with the following command:

```
tensorboard --logdir="./runs"
```

We can access the TensorBoard web server with the following URL:

```
http://localhost:6006/
```

The TensorBoard plot illustrates how `spotPython` can be used as a microscope for the internal mechanisms of the surrogate-based optimization process. Here, one important parameter, the learning rate θ of the Kriging surrogate is plotted against the number of optimization steps.

4.3 Print the Results

```
spot_de.print_results()
```

```
min y: 0.40173530984780115
x0: -3.1559787149178984
x1: 12.256173784719392

[['x0', -3.1559787149178984], ['x1', 12.256173784719392]]
```

4.4 Show the Progress

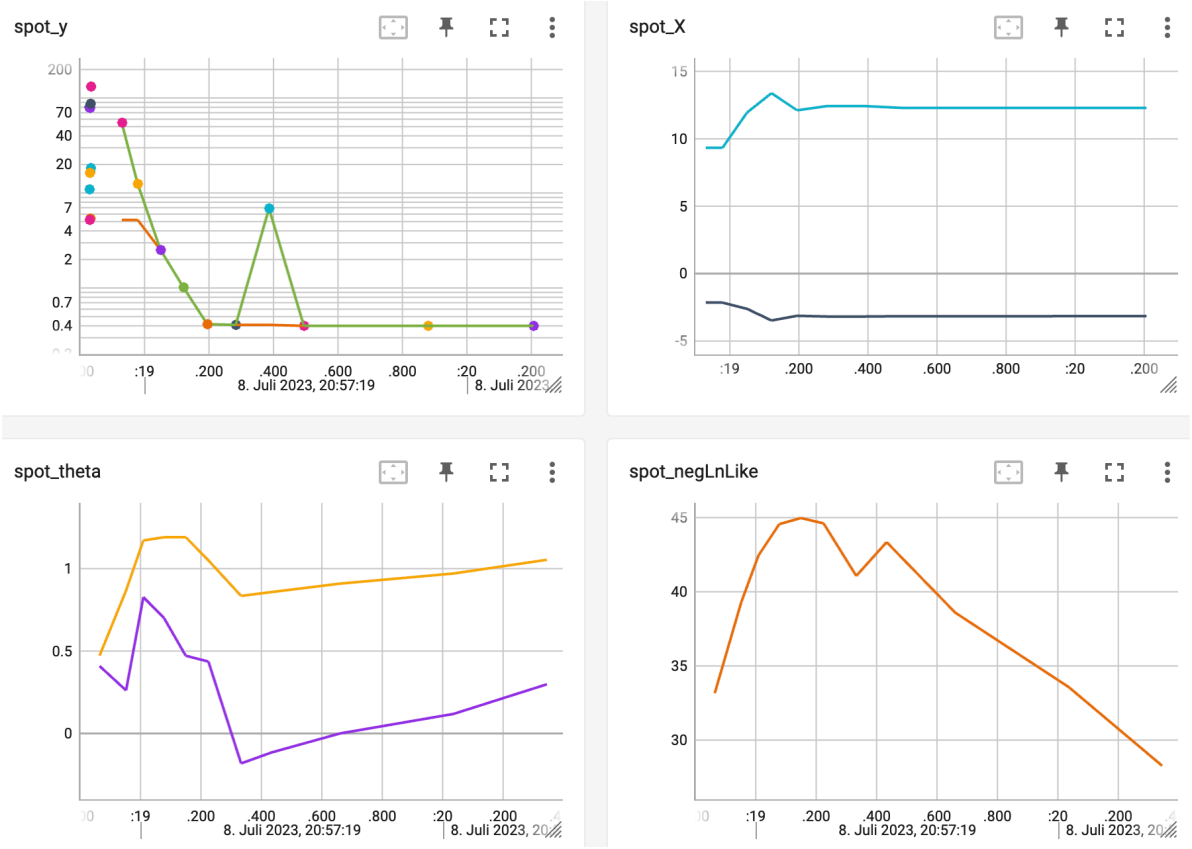
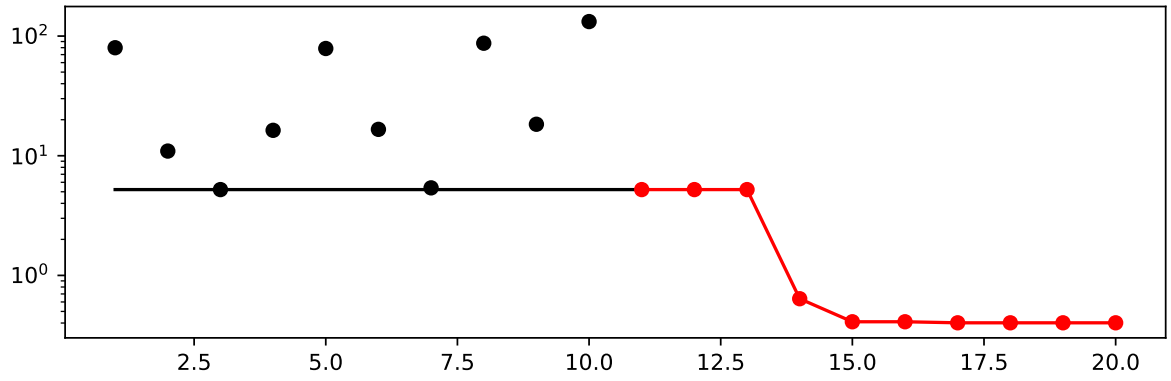
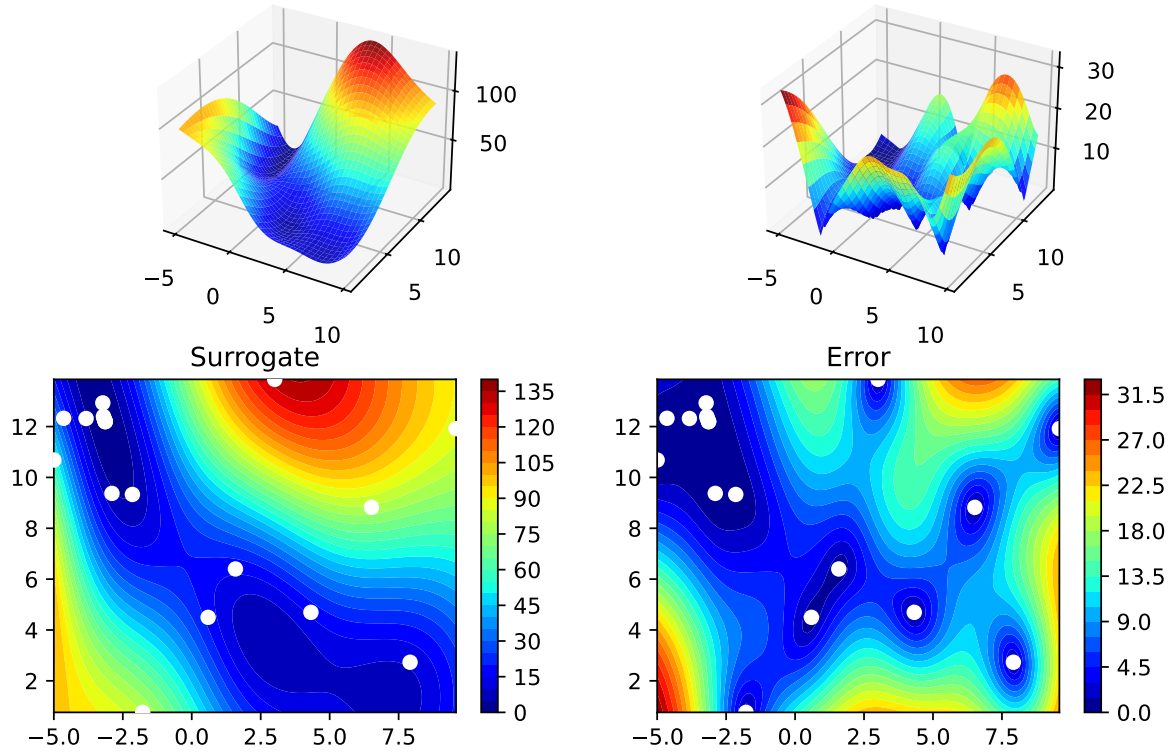


Figure 4.1: TensorBoard visualization of the spotPython optimization process and the surrogate model.

```
spot_de.plot_progress(log_y=True)
```




```
spot_de.surrogate.plot()
```



4.5 Exercises

4.5.1 `dual_annealing`

- Describe the optimization algorithm
- Use the algorithm as an optimizer on the surrogate

4.5.2 `direct`

- Describe the optimization algorithm
- Use the algorithm as an optimizer on the surrogate

4.5.3 `shgo`

- Describe the optimization algorithm
- Use the algorithm as an optimizer on the surrogate

4.5.4 `basinhopping`

- Describe the optimization algorithm
- Use the algorithm as an optimizer on the surrogate

4.5.5 Performance Comparison

Compare the performance and run time of the 5 different optimizers:

```
* `differential_evolution`
* `dual_annealing`
*  `direct`
* `shgo`
* `basinhopping`.
```

The Branin function has three global minima:

- $f(x) = 0.397887$ at
  - $(-\pi, 12.275)$,
  - $(\pi, 2.275)$, and
  - $(9.42478, 2.475)$.

- Which optima are found by the optimizers? Does the `seed` change this behavior?

Part II

Numerical Methods

5 Introduction: Numerical 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. 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 Response Surface Methods: 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.

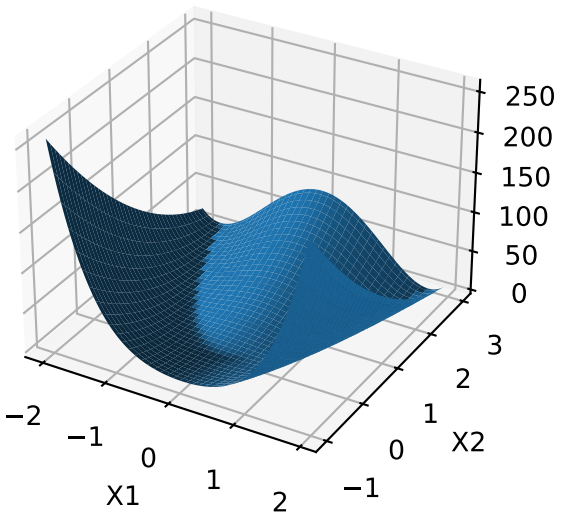
The advantages of RSM include a rich literature, well-established methods often used in manufacturing, the importance of careful experimental design combined with a well-understood model, and the potential to add significant value to scientific inquiry, process refinement, optimization, and more. However, 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.

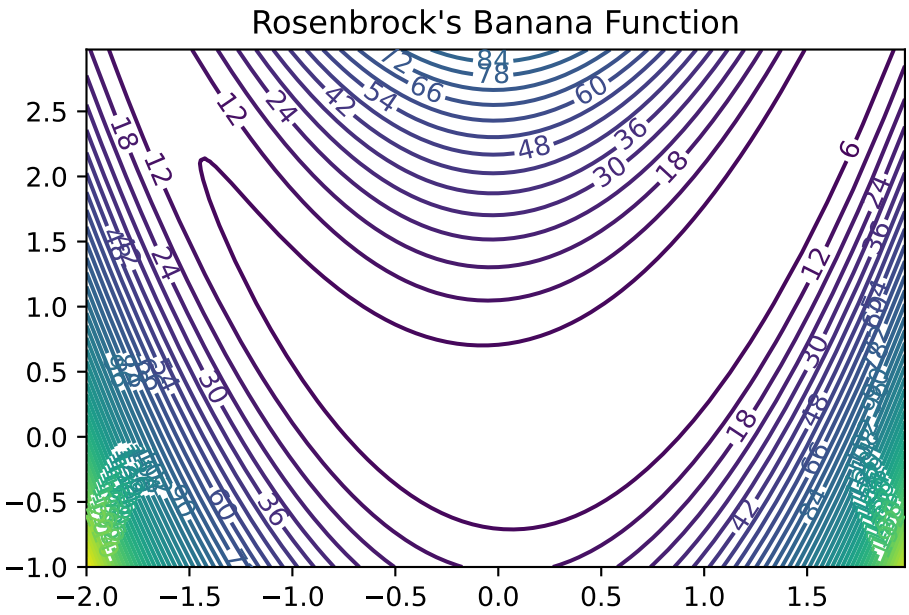
```
import numpy as np
import matplotlib.pyplot as plt

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:

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

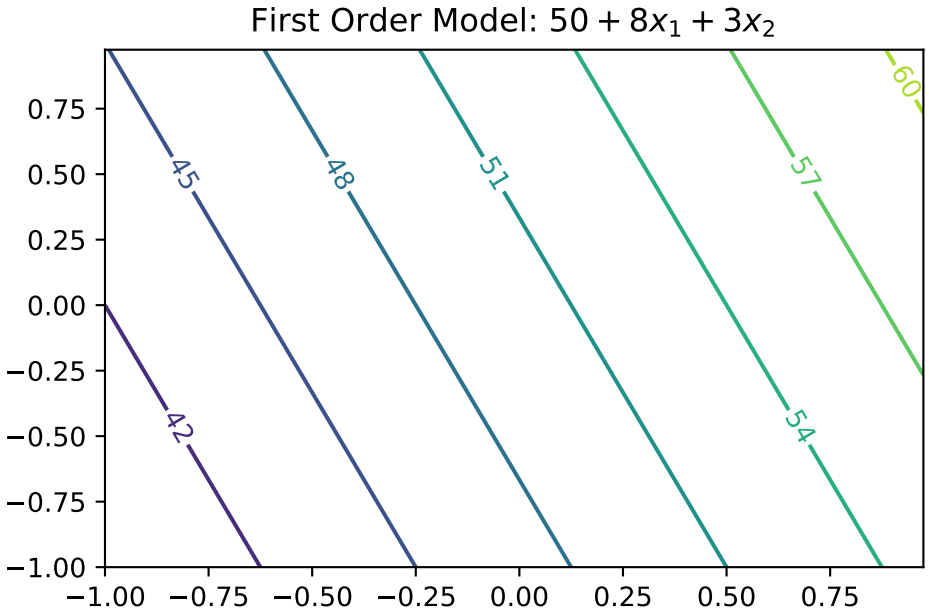
- In practice, such a surface would be obtained by fitting a model to the outcome of a designed experiment
- First-Order Model in python Evaluated on a Grid
- Evaluate model on a grid in a double-unit square centered at the origin
- Coded units are chosen arbitrarily, although one can imagine deploying 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 $\eta = 50 + 8x_1 + 3x_2 - 4x_1 x_2$

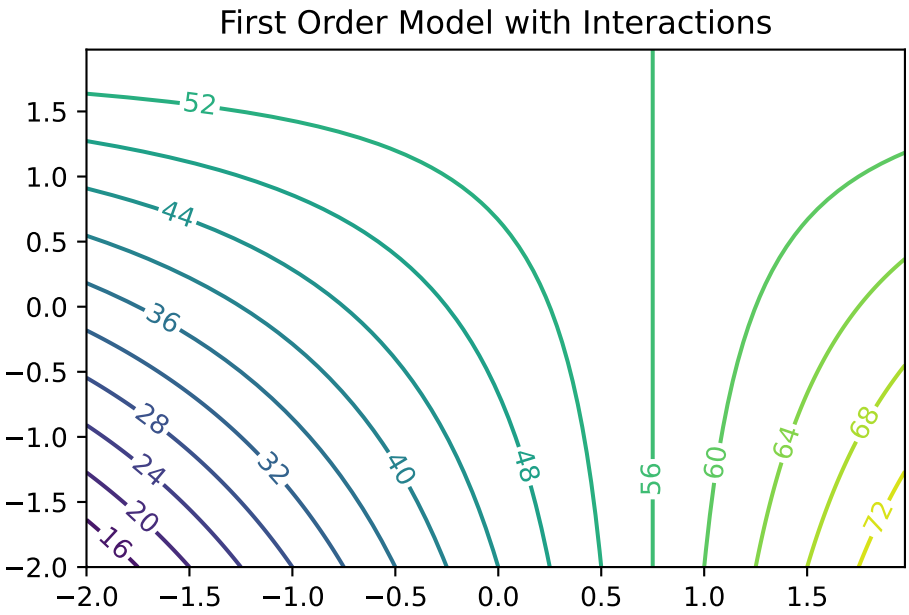
5.2.2 First-order Model with Interactions in python

- 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

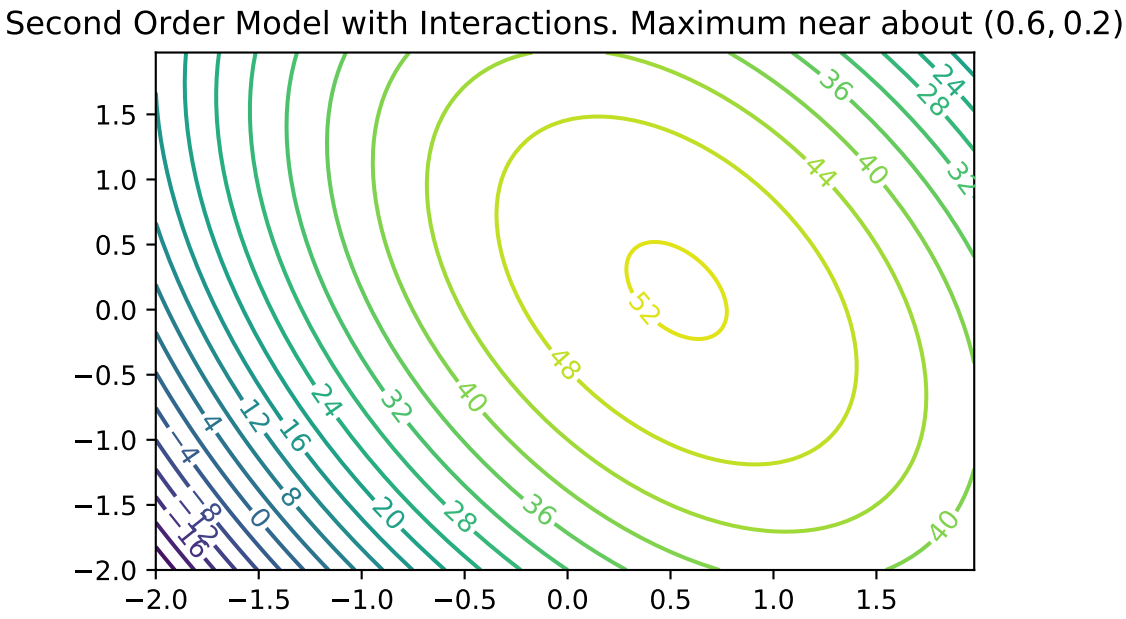
$$\eta = 50 + 8x_1 + 3x_2 - 7x_1^2 - 3x_2^2 - 4x_1 x_2$$

- 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

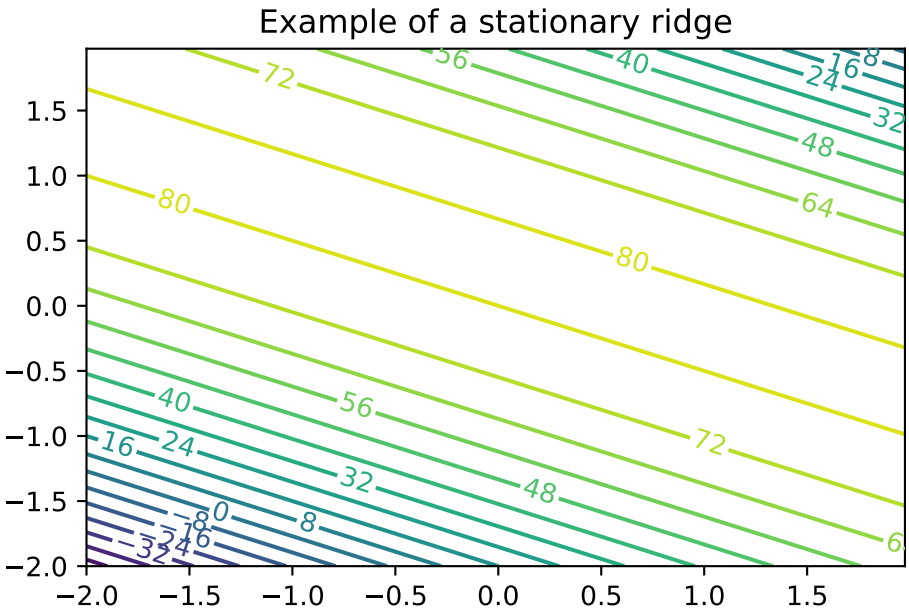
5.3.2 Example: Stationary Ridge

- 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

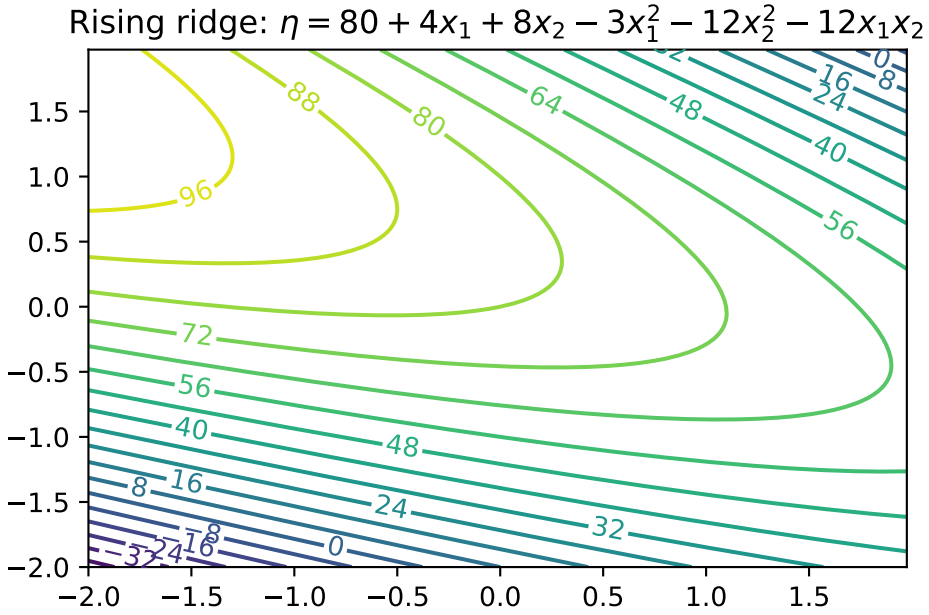
5.3.4 Example: Rising Ridge

- 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.

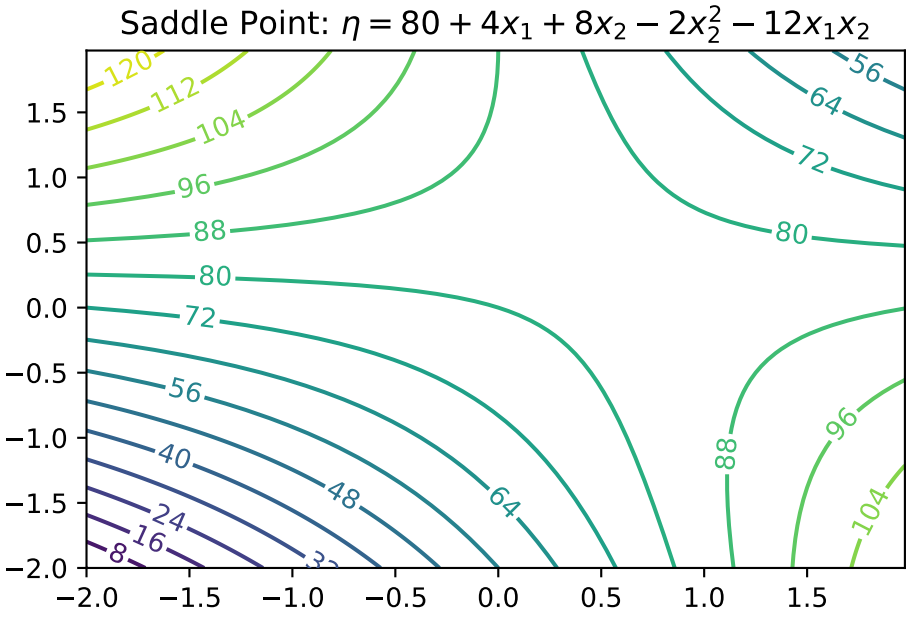
5.3.7 Saddle Point

- 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 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

5.5.1 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

5.6 RSM Experimentation

5.6.1 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

5.6.2 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

5.6.3 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

5.7 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

5.7.1 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

5.7.2 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

5.7.3 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

5.7.4 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

5.8 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.

6 Kriging (Gaussian Process Regression)

6.1 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).

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.

6.1.1 DACE Literature

Two very good texts on computer experiments and surrogate modeling are Santner, Williams, and Notz (2003) and Forrester, Sóbester, and Keane (2008). The former is the canonical reference in the statistics literature and the latter is perhaps more popular in engineering.

6.2 Background: Expectation, Mean

The distribution of a random vector is characterized by some indexes. One of them is the expected value, which is defined as

$$E[X] = \sum_{x \in D_X} x p_X(x) \qquad \text{if } X \text{ is discrete}$$

$$E[X] = \int_{x \in D_X} x f_X(x) \mathrm{d}x \quad \text{if } X \text{ is continuous.}$$

The mean, μ, of a probability distribution is a measure of its central tendency or location. That is, $E(X)$ is defined as the average of all possible values of X, weighted by their probabilities.

6.2.1 Example

Let X denote the number produced by rolling a fair die. Then

$$E(X) = 1 \times 1/6 + 2 \times 1/6 + 3 \times 1/6 + 4 \times 1/6 + 5 \times 1/6 + 6 \times 1/6 = 3.5$$

6.2.2 Sample Mean

The sample mean is an important estimate of the population mean. The sample mean of a sample $\{x_i\}$ $(i = 1, 2, \ldots, n)$ is defined as

$$\overline{x} = \frac{1}{n} \sum_i x_i.$$

6.2.3 Variance and Standard Deviation

If we are trying to predict the value of a random variable X by its mean $\mu = E(X)$, the error will be $X - \mu$. In many situations it is useful to have an idea how large this deviation or error is. Since $E(X - \mu) = E(X) - \mu = 0$, it is necessary to use the absolute value or the square of $(X - \mu)$. The squared error is the first choice, because the derivatives are easier to calculate. These considerations motivate the definition of the variance:

The variance of a random variable X is the mean squared deviation of X from its expected value $\mu = E(X)$.

$$Var(X) = E[(X - \mu)^2]. \tag{6.1}$$

6.2.4 Standard Deviation

Taking the square root of the variance to get back to the same scale of units as X gives the standard deviation. The standard deviation of X is the square root of the variance of X.

$$sd(X) = \sqrt{Var(X)}. \tag{6.2}$$

6.2.5 Calculation of the Standard Deviation with Python

The function `numpy.std` returns the standard deviation, a measure of the spread of a distribution, of the array elements. The argument `ddof` specifies the Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements. By default `ddof` is zero, i.e., `std` uses the formula

$$\sqrt{\frac{1}{N}\sum_i (x_i - \bar{x})^2} \qquad \text{with} \qquad \bar{x} = \sum_{i=1}^{N} x_i/N. \tag{6.3}$$

i Example: Standard Deviation with Python

Consider the array $[1, 2, 3]$: Since $\bar{x} = 2$, the following value is computed:

$$\sqrt{1/3 \times ((1-2)^2 + (2-2)^2 + (3-2)^2)} = \sqrt{2/3}.$$

```
import numpy as np
a = np.array([[1, 2, 3]])
np.std(a)
```

```
0.816496580927726
```

6.2.6 The Empirical Standard Deviation

The empirical standard deviation (which uses $N-1$), $\sqrt{1/2 \times ((1-2)^2 + (2-2)^2 + (3-2)^2)} = \sqrt{2/2}$, can be calculated as follows:

```
np.std(a, ddof=1)
```

```
1.0
```

6.2.7 The Argument "axis"

> Axes along which the standard deviation is computed
>
> - When you compute np.std with axis=0, it calculates the standard deviation along the vertical axis, meaning it computes the standard deviation for each column of the array.
> - On the other hand, when you compute np.std with axis=1, it calculates the standard deviation along the horizontal axis, meaning it computes the standard deviation for each row of the array.
> - If the axis parameter is not specified, np.std computes the standard deviation of the flattened array.

```
A = np.array([[1, 2], [3, 4]])
A
```

```
array([[1, 2],
       [3, 4]])
```

```
np.std(A)
```

```
1.118033988749895
```

```
np.std(A, axis=0)
```

```
array([1., 1.])
```

```
np.std(A, axis=1)
```

```
array([0.5, 0.5])
```

6.2.8 Single Versus Double Precision

In single precision, `std()` can be inaccurate:

```
a = np.zeros((2, 4*4), dtype=np.float32)
a[0, :] = 1.0
a[1, :] = 0.1
a
```

```
array([[1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. ,
        1. , 1. , 1. ],
       [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1,
        0.1, 0.1, 0.1]], dtype=float32)
```

```
np.std(a, axis=0)
```

```
array([0.45, 0.45, 0.45, 0.45, 0.45, 0.45, 0.45, 0.45, 0.45, 0.45, 0.45,
       0.45, 0.45, 0.45, 0.45, 0.45], dtype=float32)
```

```
np.std(a, axis=1)
```

```
array([0., 0.], dtype=float32)
```

```
abs(0.45 - np.std(a))
```

```
1.7881393421514957e-08
```

> **Float data types**
>
> - float32 and float64 are data types in numpy that specify the precision of floating point numbers.

- float32 is a single-precision floating point number that occupies 32 bits of memory. It has a precision of about 7 decimal digits.
- float64 is a double-precision floating point number that occupies 64 bits of memory. It has a precision of about 15 decimal digits.
- The main difference between float32 and float64 is the precision and memory usage. float64 provides a higher precision but uses more memory, while float32 uses less memory but has a lower precision.

Computing the standard deviation in float64 is more accurate (result may vary), see https://numpy.org/devdocs/reference/generated/numpy.std.html.

```
abs(0.45 - np.std(a, dtype=np.float64))
```

```
7.450580707946131e-10
```

Example: 32 versus 64 bit

```
import numpy as np

# Define a number
num = 0.1234567891234567 89

# Convert to float32 and float64
num_float32 = np.float32(num)
num_float64 = np.float64(num)

# Print the number in both formats
print("float32: ", num_float32)
print("float64: ", num_float64)
```

```
float32:  0.12345679
float64:  0.12345678912345678
```

The float32 data type in numpy represents a single-precision floating point number. It uses 32 bits of memory, which gives it a precision of about 7 decimal digits. On the other hand, float64 represents a double-precision floating point number. It uses 64 bits of memory, which gives it a precision of about 15 decimal digits.

The reason float32 shows fewer digits is because it has less precision due to using less memory. The bits of memory are used to store the sign, exponent, and fraction parts of the floating point number, and with fewer bits, you can represent fewer digits accurately.

6.3 Random Numbers in Python

Results from computers are deterministic, so it sounds like a contradiction in terms to generate random numbers on a computer. Standard computers generate pseudo-randomnumbers, i.e., numbers that behave as if they were drawn randomly.

> **Deterministic Random Numbers**
>
> - Idea: Generate deterministically numbers that **look** (behave) as if they were drawn randomly.

6.4 The Uniform Distribution

The probability density function of the uniform distribution is defined as:

$$f_X(x) = \frac{1}{b-a} \qquad \text{for } x \in [a, b].$$

Generate 10 random numbers from a uniform distribution between $a = 0$ and $b = 1$:

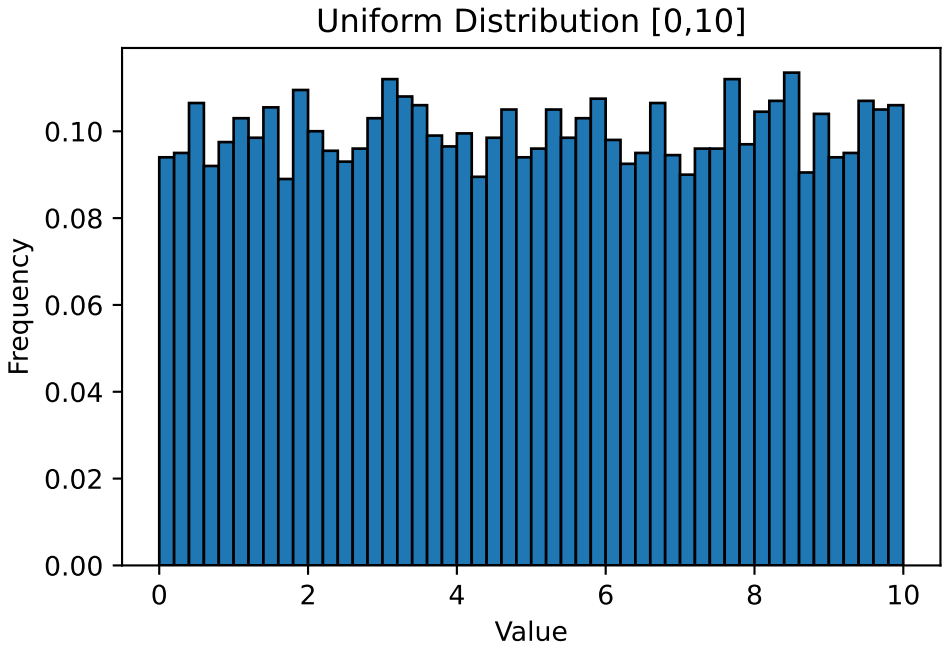
```
import numpy as np
# Initialize the random number generator
rng = np.random.default_rng(seed=123456789)
n = 10
x = rng.uniform(low=0.0, high=1.0, size=n)
x
```

```
array([0.02771274, 0.90670006, 0.88139355, 0.62489728, 0.79071481,
       0.82590801, 0.84170584, 0.47172795, 0.95722878, 0.94659153])
```

Generate 10,000 random numbers from a uniform distribution between 0 and 10 and plot a histogram of the numbers:

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

# Initialize the random number generator
rng = np.random.default_rng(seed=123456789)

# Generate random numbers from a uniform distribution
```

```
x = rng.uniform(low=0, high=10, size=10000)

# Plot a histogram of the numbers
plt.hist(x, bins=50, density=True, edgecolor='black')
plt.title('Uniform Distribution [0,10]')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.show()
```



6.5 The Normal Distribution

The probability density function of the normal distribution is defined as:

$$f_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right), \tag{6.4}$$

where: μ is the mean; σ is the standard deviation.

To generate ten random numbers from a normal distribution, the following command can be used.

```
# generate 10 random numbers between from a normal distribution
import numpy as np
rng = np.random.default_rng()
n = 10
mu, sigma = 2, 0.1
x = rng.normal(mu, sigma, n)
x
```

```
array([2.01037381, 1.9616171 , 1.90861659, 2.02963636, 1.8539017 ,
       2.06482466, 1.91680398, 2.05030727, 1.93364738, 2.03492169])
```

Verify the mean:

```
abs(mu - np.mean(x))
```

```
0.023534947251900862
```

Note: To verify the standard deviation, we use `ddof = 1` (empirical standard deviation):

```
abs(sigma - np.std(x, ddof=1))
```

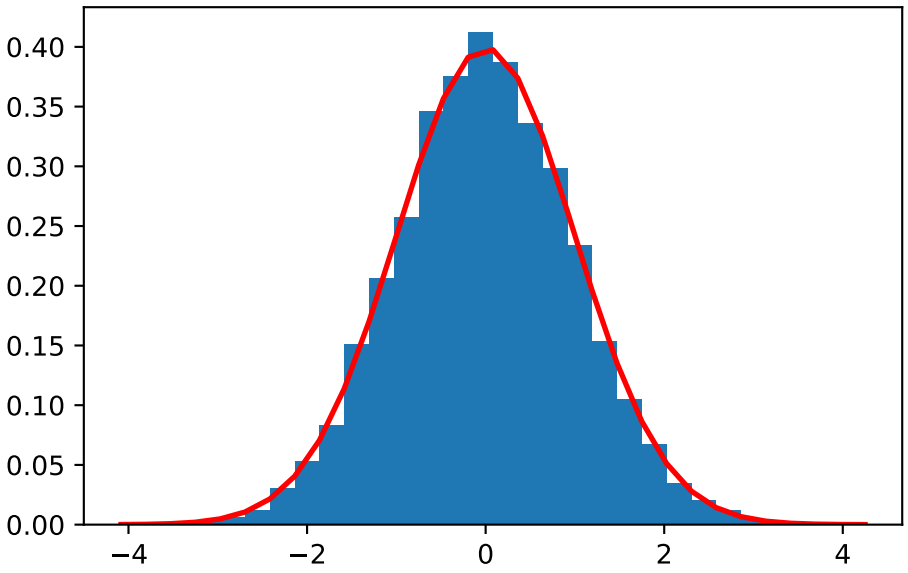
```
0.028591387791829112
```

A normally distributed random variable is a random variable whose associated probability distribution is the normal (or Gaussian) distribution. The normal distribution is a continuous probability distribution characterized by a symmetric bell-shaped curve.

The distribution is defined by two parameters: the mean μ and the standard deviation σ. The mean indicates the center of the distribution, while the standard deviation measures the spread or dispersion of the distribution.

This distribution is widely used in statistics and the natural and social sciences as a simple model for random variables with unknown distributions.

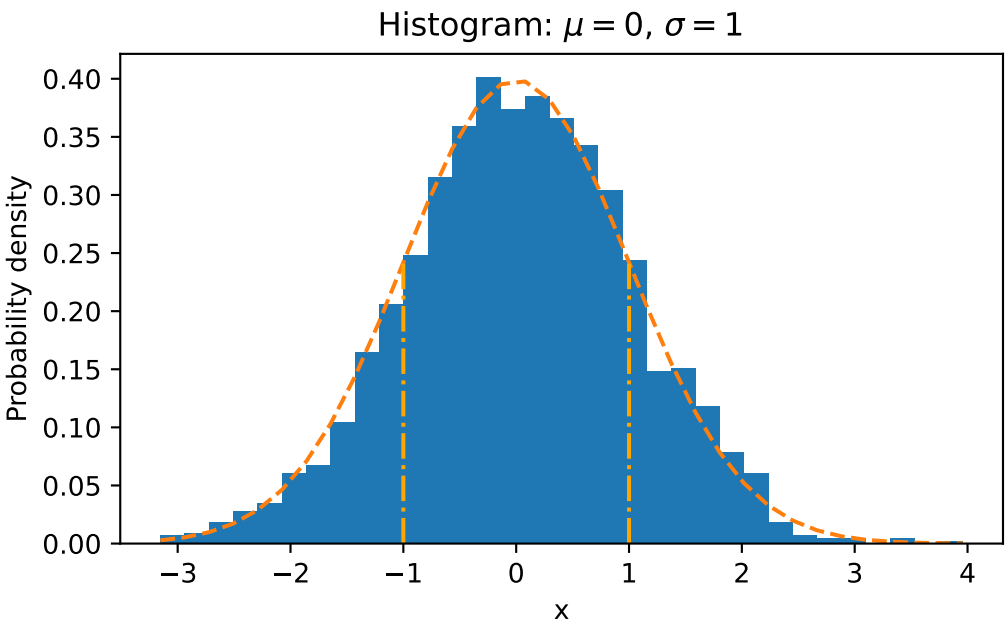
```
plot_normal_distribution(mu=0, sigma=1, num_samples=10000)
```



6.5.1 Visualization of the Standard Deviation

The standard deviation of normal distributed can be visualized in terms of the histogram of X:

- about 68% of the values will lie in the interval within one standard deviation of the mean
- 95% lie within two standard deviation of the mean
- and 99.9% lie within 3 standard deviations of the mean.



6.6 Standardization

To compare statistical properties of random variables which use different units, it is a common practice to transform these random variables into standardized variables. If a random variable X has expectation $E(X) = \mu$ and standard deviation $sd(X) = \sigma > 0$, the random variable

$$X^* = (X - \mu)/\sigma$$

is called X in standard units. It has $E(X^*) = 0$ and $sd(X^*) = 1$.

6.7 Realizations of a Normal Distribution

Realizations of a normal distribution refers to the actual values that you get when you draw samples from a normal distribution. Each sample drawn from the distribution is a realization of that distribution.

For example, if you have a normal distribution with a mean of 0 and a standard deviation of 1, each number you draw from that distribution is a realization.

Here's a Python example:

```python
import numpy as np

# Define the parameters of the normal distribution
mu = 0
sigma = 1

# Draw 10 samples (realizations) from the normal distribution
realizations = np.random.normal(mu, sigma, 10)

print(realizations)
```

```
[ 0.48951662  0.23879586 -0.44811181 -0.610795   -2.02994507  0.60794659
 -0.35410888  0.15258149  0.50127485 -0.78640277]
```

In this code, np.random.normal generates 10 realizations of a normal distribution with a mean of 0 and a standard deviation of 1. The realizations array contains the actual values drawn from the distribution.

6.8 The Multivariate Normal Distribution

The multivariate normal, multinormal, or Gaussian distribution serves as a generalization of the one-dimensional normal distribution to higher dimensions. We will consider k-dimensional random vectors $X = (X_1, X_2, \dots, X_k)$. When drawing samples from this distribution, it results in a set of values represented as $\{x_1, x_2, \dots, x_k\}$. To fully define this distribution, it is necessary to specify its mean μ and covariance matrix Σ. These parameters are analogous to the mean, which represents the central location, and the variance (squared standard deviation) of the one-dimensional normal distribution introduced in Equation 6.4.

In the context of the multivariate normal distribution, the mean takes the form of a coordinate within an k-dimensional space. This coordinate represents the location where samples are most likely to be generated, akin to the peak of the bell curve in a one-dimensional or univariate normal distribution.

> **i** Covariance of two random variables
>
> For two random variables X and Y, the covariance is defined as the expected value (or mean) of the product of their deviations from their individual expected values:
>
> $$\operatorname{cov}(X, Y) = \mathrm{E}\left[(X - \mathrm{E}[X])(Y - \mathrm{E}[Y])\right]$$
>
> The covariance within the multivariate normal distribution denotes the extent to which two variables vary together. The elements of the covariance matrix, such as Σ_{ij}, represent

the covariances between the variables x_i and x_j. These covariances describe how the different variables in the distribution are related to each other in terms of their variability. The probability density function (PDF) of the multivariate normal distribution is defined as:

$$f_X(x) = \frac{1}{\sqrt{(2\pi)^n \det(\Sigma)}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right),$$

where: μ is the $k \times 1$ mean vector; Σ is the $k \times k$ covariance matrix. The covariance matrix Σ is assumed to be positive definite, so that its determinant is strictly positive. For discrete random variables, covariance can be written as:

$$\operatorname{cov}(X,Y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - E(X))(y_i - E(Y)).$$

Figure 6.1 shows draws from a bivariate normal distribution with $\mu = \begin{pmatrix}0\\0\end{pmatrix}$ and $\Sigma = \begin{pmatrix}9 & 4\\4 & 9\end{pmatrix}$.

```
import numpy as np
rng = np.random.default_rng()
import matplotlib.pyplot as plt
mean = [0, 0]
cov = [[9, 4], [4, 9]]  # diagonal covariance
x, y = rng.multivariate_normal(mean, cov, 1000).T
# Create a scatter plot of the numbers
plt.scatter(x, y, s=2)
plt.axis('equal')
plt.grid()
plt.title(f"Bivariate Normal. Mean zero and positive covariance: {cov}")
plt.show()
```

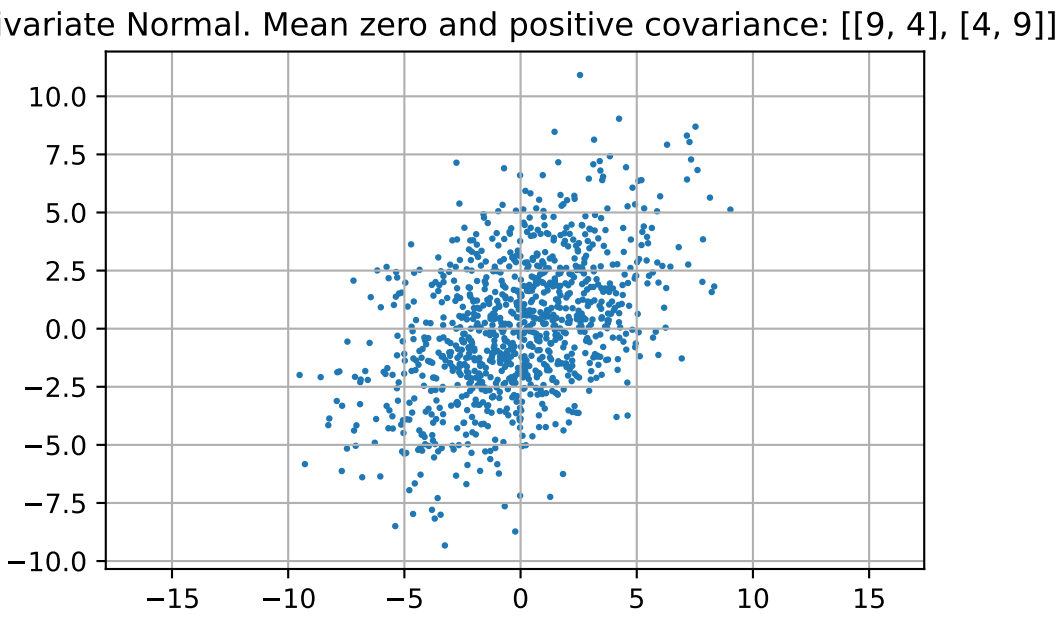


Figure 6.1: Bivariate Normal. Mean zero and covariance $\Sigma = \begin{pmatrix} 9 & 4 \\ 4 & 9 \end{pmatrix}$

The covariance matrix of a bivariate normal distribution determines the shape, orientation, and spread of the distribution in the two-dimensional space.

The diagonal elements of the covariance matrix (σ_1^2, σ_2^2) are the variances of the individual variables. They determine the spread of the distribution along each axis. A larger variance corresponds to a greater spread along that axis.

The off-diagonal elements of the covariance matrix (σ_{12}, σ_{21}) are the covariances between the variables. They determine the orientation and shape of the distribution. If the covariance is positive, the distribution is stretched along the line $y = x$, indicating that the variables tend to increase together. If the covariance is negative, the distribution is stretched along the line $y = -x$, indicating that one variable tends to decrease as the other increases. If the covariance is zero, the variables are uncorrelated and the distribution is axis-aligned.

In Figure 6.1, the variances are identical and the variables are correlated (covariance is 4), so the distribution is stretched along the line $y = x$.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal

# Parameters
```

```
mu = np.array([0, 0])
cov = np.array([[9, 4], [4, 9]])

# Create grid and multivariate normal
x = np.linspace(-10,10,100)
y = np.linspace(-10,10,100)
X, Y = np.meshgrid(x,y)
pos = np.empty(X.shape + (2,))
pos[:, :, 0] = X; pos[:, :, 1] = Y
rv = multivariate_normal(mu, cov)

fig = plt.figure()
ax = plt.axes(projection='3d')
surf=ax.plot_surface(X, Y, rv.pdf(pos),cmap='viridis',linewidth=0)
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
ax.set_title('Bivariate Normal Distribution')
fig.colorbar(surf, shrink=0.5, aspect=10)
plt.show()
```

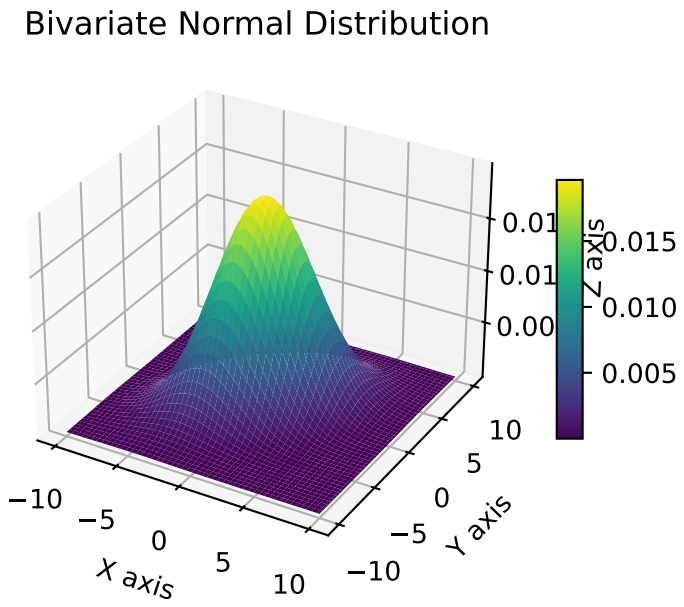


Figure 6.2: Bivariate Normal. Mean zero and covariance $\Sigma = \begin{pmatrix} 9 & 4 \\ 4 & 9 \end{pmatrix}$

6.8.1 The Bivariate Normal Distribution with Mean Zero and Zero Covariances $\sigma_{12} = \sigma_{21} = 0$

$$\Sigma = \begin{pmatrix} 9 & 0 \\ 0 & 9 \end{pmatrix}$$

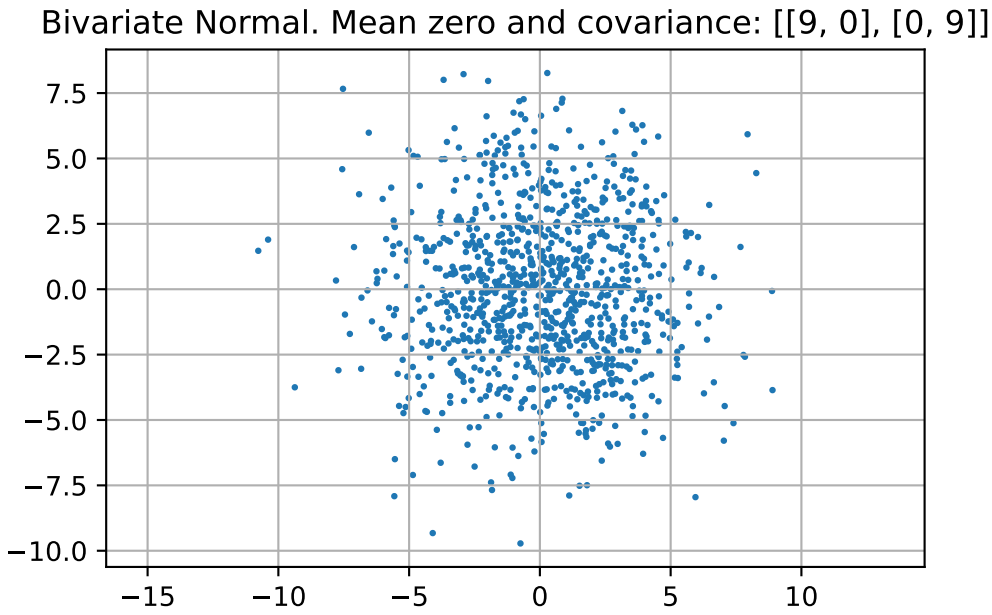


Figure 6.3: Bivariate Normal. Mean zero and covariance $\Sigma = \begin{pmatrix} 9 & 0 \\ 0 & 9 \end{pmatrix}$

6.8.2 The Bivariate Normal Distribution with Mean Zero and Negative Covariances $\sigma_{12} = \sigma_{21} = -4$

$$\Sigma = \begin{pmatrix} 9 & -4 \\ -4 & 9 \end{pmatrix}$$

6.9 Positive Definite Matrices

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

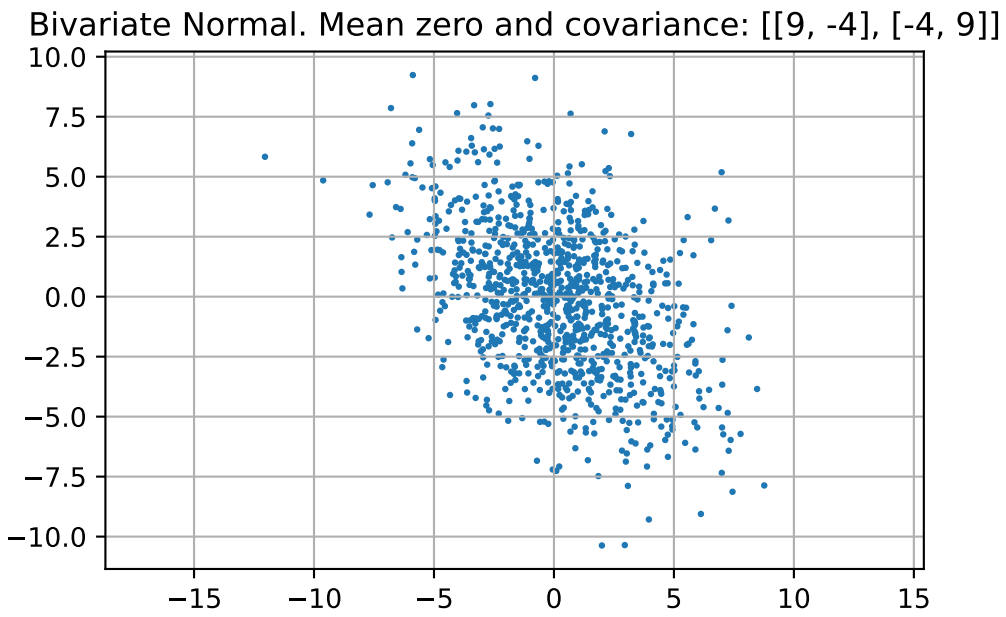


Figure 6.4: Bivariate Normal. Mean zero and covariance $\Sigma = \begin{pmatrix} 9 & -4 \\ -4 & 9 \end{pmatrix}$

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.

```python
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
```

6.9.1 Cholesky Decomposition

More effficent (and check if symmetric) is based on Cholesky decomposition.

```
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
```

Example: Cholesky decomposition.

`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.]]
```

6.10 Multivariate Normal Distribution: Maximum Likelihood Estimation

Consider the first n terms of an identically and independently distributed (i.i..d.) sequence $X^{(j)}$ of k-dimensional multivariate normal random vectors, i.e., $X^{(j)} \sim N(\mu, \Sigma)$, $j = 1, 2, ...$ The joint probability density function of the j-th term of the sequence is

$$f_X(x_j) = \frac{1}{\sqrt{(2\pi)^k \det(\Sigma)}} \exp\left(-\frac{1}{2}(x_j - \mu)^T \Sigma^{-1}(x_j - \mu)\right),$$

where: μ is the $k \times 1$ mean vector; Σ is the $k \times k$ covariance matrix. The covariance matrix Σ is assumed to be positive definite, so that its determinant is strictly positive. We use $x_1, ... x_n$, i.e., the realizations of the first n random vectors in the sequence, to estimate the two unknown parameters μ and Σ.

The likelihood function is defined as the joint probability density function of the observed data, viewed as a function of the unknown parameters. Since the terms in the sequence are independent, their joint density is equal to the product of their marginal densities. As a consequence, the likelihood function can be written as the product of the individual densities:

$$\begin{aligned} L(\mu, \Sigma) &= \prod_{j=1}^{n} f_X(x_j) = \prod_{j=1}^{n} \frac{1}{\sqrt{(2\pi)^k \det(\Sigma)}} \exp\left(-\frac{1}{2}(x_j - \mu)^T \Sigma^{-1}(x_j - \mu)\right) \\ &= \frac{1}{(2\pi)^{nk/2} \det(\Sigma)^{n/2}} \exp\left(-\frac{1}{2}\sum_{j=1}^{n}(x_j - \mu)^T \Sigma^{-1}(x_j - \mu)\right). \end{aligned}$$

The log-likelihood function is

$$\ell(\mu, \Sigma) = -\frac{nk}{2}\ln(2\pi) - \frac{n}{2}\ln(\det(\Sigma)) - \frac{1}{2}\sum_{j=1}^{n}(x_j - \mu)^T \Sigma^{-1}(x_j - \mu).$$

The likelihood function is well-defined only if $\det(\Sigma) > 0$.

6.11 Introduction to Gaussian Processes

The concept of GP (Gaussian Process) regression can be understood as a simple extension of linear modeling. It is worth noting that this approach goes by various names and acronyms, including "kriging," a term derived from geostatistics, as introduced by Matheron in 1963.

Additionally, it is referred to as Gaussian spatial modeling or a Gaussian stochastic process, and machine learning (ML) researchers often use the term Gaussian process regression (GPR). In all of these instances, the central focus is on regression. This involves training on both inputs and outputs, with the ultimate objective of making predictions and quantifying uncertainty (referred to as uncertainty quantification or UQ).

However, it's important to emphasize that GPs are not a universal solution for every problem. Specialized tools may outperform GPs in specific, non-generic contexts, and GPs have their own set of limitations that need to be considered.

6.11.1 Gaussian Process Prior

In the context of GP, any finite collection of realizations, which is represented by n observations, is modeled as having a multivariate normal (MVN) distribution. The characteristics of these realizations can be fully described by two key parameters:

1. Their mean, denoted as an n-vector μ.
2. The covariance matrix, denoted as an $n \times n$ matrix Σ. This covariance matrix encapsulates the relationships and variability between the individual realizations within the collection.

6.11.2 Covariance Function

The covariance function is defined by inverse exponentiated squared Euclidean distance:

$$\Sigma(\vec{x}, \vec{x}') = \exp\{-||\vec{x} - \vec{x}'||^2\},$$

where $\vec{x}$ and $\vec{x}'$ are two points in the k-dimensional input space and $\|\cdot\|$ denotes the Euclidean distance, i.e.,

$$||\vec{x} - \vec{x}'||^2 = \sum_{i=1}^{k} (x_i - x_i')^2.$$

An 1-d example is shown in Figure 6.5.

```
visualize_inverse_exp_squared_distance(5, 0.0, [0.5, 1, 2.0])
```

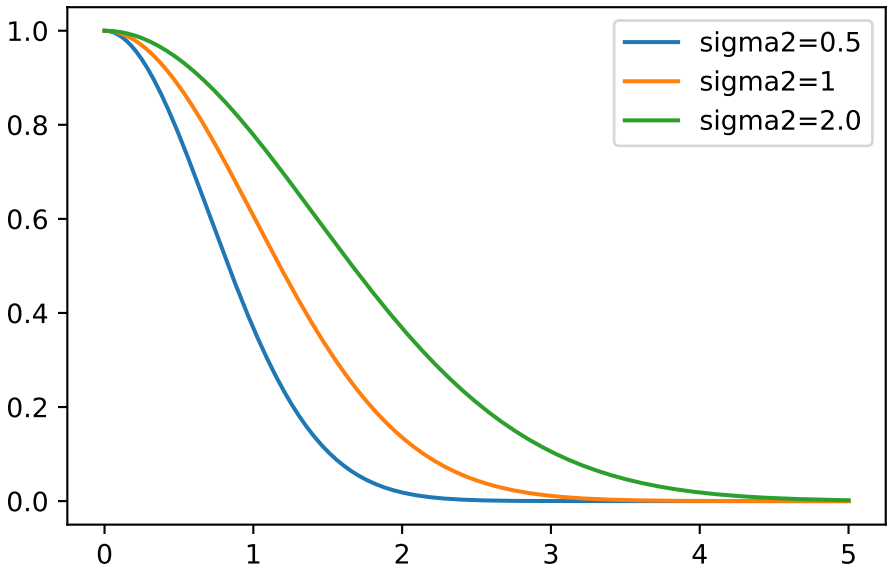


Figure 6.5: One-dim inverse exponentiated squared Euclidean distance

The covariance function is also referred to as the kernel function. The *Gaussian* kernel uses an additional parameter, σ^2, to control the rate of decay. This parameter is referred to as the length scale or the characteristic length scale. The covariance function is then defined as

$$\Sigma(\vec{x}, \vec{x}') = \exp\{-||\vec{x} - \vec{x}'||^2/(2\sigma^2)\}. \tag{6.5}$$

The covariance decays exponentially fast as $\vec{x}$ and $\vec{x}'$ become farther apart. Observe that

$$\Sigma(\vec{x}, \vec{x}) = 1$$

and

$$\Sigma(\vec{x}, \vec{x}') < 1$$

for $\vec{x} \neq \vec{x}'$. The function $\Sigma(\vec{x}, \vec{x}')$ must be positive definite.

Positive Definiteness

Positive definiteness in the context of the covariance matrix Σ_n is a fundamental requirement. It is determined by evaluating $\Sigma(x_i, x_j)$ at pairs of n $\vec{x}$-values, denoted as $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n$. The condition for positive definiteness is that for all $\vec{x}$ vectors that are not equal to zero, the expression $\vec{x}^\top \Sigma_n \vec{x}$ must be greater than zero. This property is

essential when intending to use Σ_n as a covariance matrix in multivariate normal (MVN) analysis. It is analogous to the requirement in univariate Gaussian distributions where the variance parameter, σ^2, must be positive.

Gaussian Processes (GPs) can be effectively utilized to generate random data that follows a smooth functional relationship. The process involves the following steps:

1. Select a set of $\vec{x}$-values, denoted as $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n$.
2. Define the covariance matrix Σ_n by evaluating $\Sigma_n^{ij} = \Sigma(\vec{x}_i, \vec{x}_j)$ for $i, j = 1, 2, \ldots, n$.
3. Generate an n-variate realization Y that follows a multivariate normal distribution with a mean of zero and a covariance matrix Σ_n, expressed as $Y \sim \mathcal{N}_n(0, \Sigma_n)$.
4. Visualize the result by plotting it in the x-y plane.

6.11.3 Construction of the Covariance Matrix

Here is an one-dimensional example. The process begins by creating an input grid using $\vec{x}$-values. This grid consists of 100 elements, providing the basis for further analysis and visualization.

```
import numpy as np
n = 100
X = np.linspace(0, 10, n, endpoint=False).reshape(-1,1)
```

In the context of this discussion, the construction of the covariance matrix, denoted as Σ_n, relies on the concept of inverse exponentiated squared Euclidean distances. However, it's important to note that a modification is introduced later in the process. Specifically, the diagonal of the covariance matrix is augmented with a small value, represented as "eps" or ϵ.

The reason for this augmentation is that while inverse exponentiated distances theoretically ensure the covariance matrix's positive definiteness, in practical applications, the matrix can sometimes become numerically ill-conditioned. By adding a small value to the diagonal, such as ϵ, this ill-conditioning issue is mitigated. In this context, ϵ is often referred to as "jitter."

```
import numpy as np
from numpy import array, zeros, power, ones, exp, multiply, eye, linspace, mat, spacing, s
from numpy.linalg import cholesky, solve
from numpy.random import multivariate_normal
def build_Sigma(X, sigma2):
    n = X.shape[0]
    k = X.shape[1]
    D = zeros((k, n, n))
```

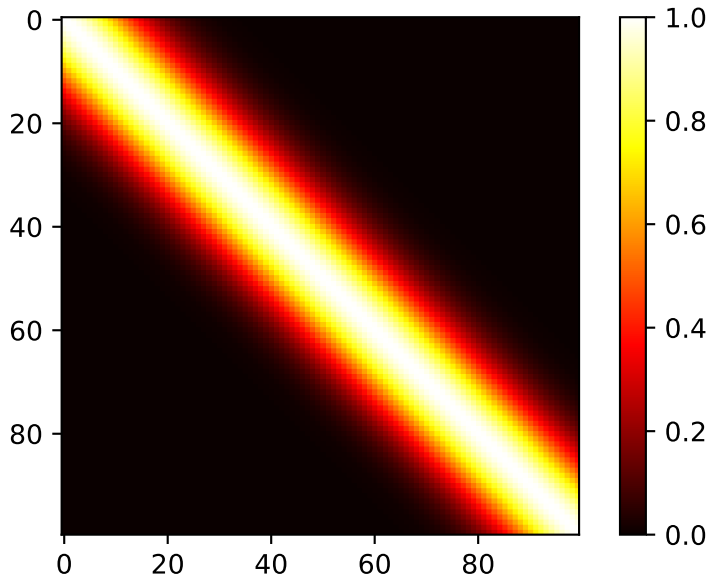
```
    for l in range(k):
        for i in range(n):
            for j in range(i, n):
                D[l, i, j] = 1/(2*sigma2[l])*(X[i,l] - X[j,l])**2
    D = sum(D)
    D = D + D.T
    return exp(-D)
```

```
sigma2 = np.array([1.0])
Sigma = build_Sigma(X, sigma2)
np.round(Sigma[:3,:], 3)
```

```
array([[1.   , 0.995, 0.98 , 0.956, 0.923, 0.882, 0.835, 0.783, 0.726,
        0.667, 0.607, 0.546, 0.487, 0.43 , 0.375, 0.325, 0.278, 0.236,
        0.198, 0.164, 0.135, 0.11 , 0.089, 0.071, 0.056, 0.044, 0.034,
        0.026, 0.02 , 0.015, 0.011, 0.008, 0.006, 0.004, 0.003, 0.002,
        0.002, 0.001, 0.001, 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   ],
       [0.995, 1.   , 0.995, 0.98 , 0.956, 0.923, 0.882, 0.835, 0.783,
        0.726, 0.667, 0.607, 0.546, 0.487, 0.43 , 0.375, 0.325, 0.278,
        0.236, 0.198, 0.164, 0.135, 0.11 , 0.089, 0.071, 0.056, 0.044,
        0.034, 0.026, 0.02 , 0.015, 0.011, 0.008, 0.006, 0.004, 0.003,
        0.002, 0.002, 0.001, 0.001, 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   ],
       [0.98 , 0.995, 1.   , 0.995, 0.98 , 0.956, 0.923, 0.882, 0.835,
        0.783, 0.726, 0.667, 0.607, 0.546, 0.487, 0.43 , 0.375, 0.325,
        0.278, 0.236, 0.198, 0.164, 0.135, 0.11 , 0.089, 0.071, 0.056,
        0.044, 0.034, 0.026, 0.02 , 0.015, 0.011, 0.008, 0.006, 0.004,
        0.003, 0.002, 0.002, 0.001, 0.001, 0.   , 0.   , 0.   , 0.   ,
```

```
0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
0.   ]])
```

```
import matplotlib.pyplot as plt
plt.imshow(Sigma, cmap='hot', interpolation='nearest')
plt.colorbar()
plt.show()
```



6.11.4 Generation of Random Samples and Plotting the Realizations of the Random Function

In the context of the multivariate normal distribution, the next step is to utilize the previously constructed covariance matrix denoted as `Sigma`. It is used as an essential component in generating random samples from the multivariate normal distribution.

The function `multivariate_normal` is employed for this purpose. It serves as a random number generator specifically designed for the multivariate normal distribution. In this case, the mean of the distribution is set equal to `mean`, and the covariance matrix is provided as `Psi`.

The argument `size` specifies the number of realizations, which, in this specific scenario, is set to one.

By default, the mean vector is initialized to zero. To match the number of samples, which is equivalent to the number of rows in the `X` and `Sigma` matrices, the argument `zeros(n)` is used, where `n` represents the number of samples (here taken from the size of the matrix, e.g.,: `Sigma.shape[0]`).

```
rng = np.random.default_rng(seed=12345)
Y = rng.multivariate_normal(zeros(Sigma.shape[0]), Sigma, size = 1, check_valid="raise").r
Y.shape
```

```
(100, 1)
```

Now we can plot the results, i.e., a finite realization of the random function $Y()$ under a GP prior with a particular covariance structure. We will plot those `X` and `Y` pairs as connected points on an x-y plane.

```
import matplotlib.pyplot as plt
plt.plot(X, Y)
plt.title("Realization of Random Functions under a GP prior.\n sigma2: {}".format(sigma2[0
plt.show()
```

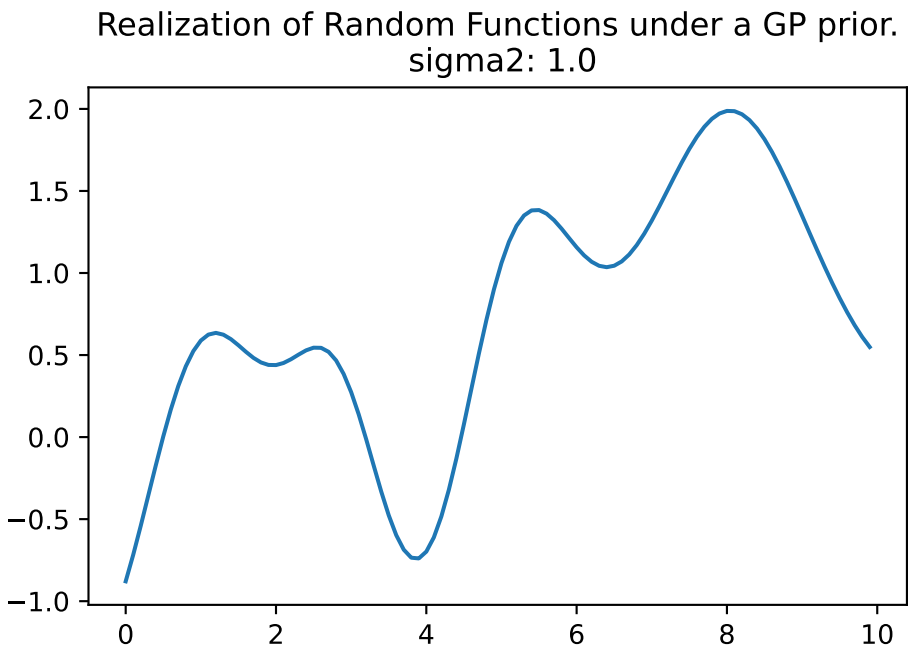


Figure 6.6: Realization of one random function under a GP prior. sigma2: 1.0

```
rng = np.random.default_rng(seed=12345)
Y = rng.multivariate_normal(zeros(Sigma.shape[0]), Sigma, size = 3, check_valid="raise")
plt.plot(X, Y.T)
plt.title("Realization of Three Random Functions under a GP prior.\n sigma2: {}".format(si
plt.show()
```

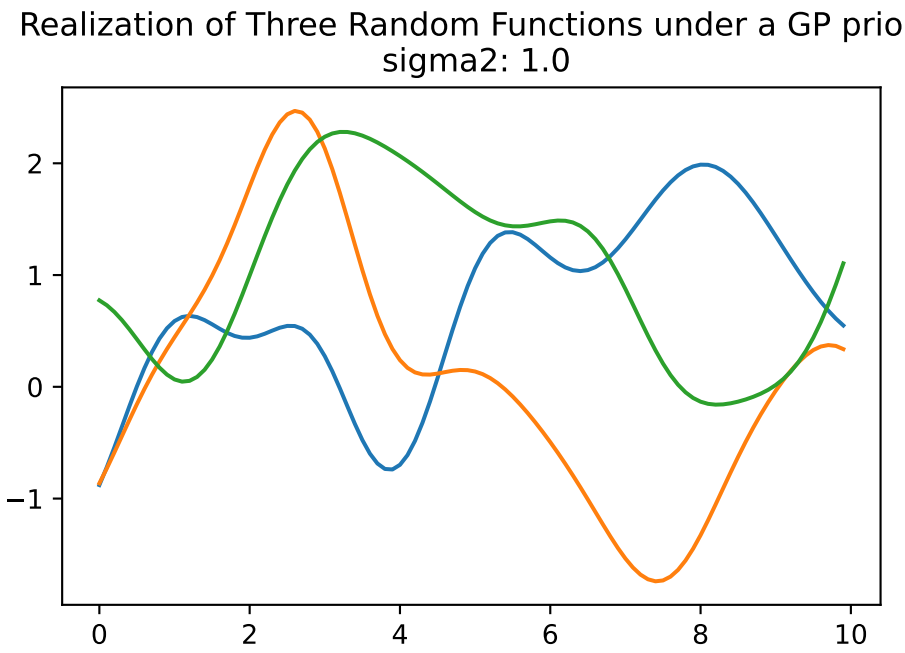


Figure 6.7: Realization of three random functions under a GP prior. sigma2: 1.0

6.11.5 Properties of the 1d Example

6.11.5.1 Several Bumps:

In this analysis, we observe several bumps in the x-range of $[0, 10]$. These bumps in the function occur because shorter distances exhibit high correlation, while longer distances tend to be essentially uncorrelated. This leads to variations in the function's behavior:

- When x and x' are one σ unit apart, the correlation is $\exp\left(-\sigma^2/(2\sigma^2)\right) = \exp(-1/2) \approx 0.61$, i.e., a relative high correlation.
- 2σ apart means correlation $\exp(-2^2/2) \approx 0.14$, i.e., only small correlation.
- 4σ apart means correlation $\exp(-4^2/2) \approx 0.0003$, i.e., nearly no correlation—variables are considered independent for almost all practical application.

6.11.5.2 Smoothness:

The function plotted in Figure 6.6 represents only a finite realization, which means that we have data for a limited number of pairs, specifically 100 points. These points appear smooth in a tactile sense because they are closely spaced, and the plot function connects the dots with lines to create the appearance of smoothness. The complete surface, which can be conceptually

extended to an infinite realization over a compact domain, is exceptionally smooth in a calculus sense due to the covariance function's property of being infinitely differentiable.

6.11.5.3 Scale of Two:

Regarding the scale of the Y values, they have a range of approximately $[-2, 2]$, with a 95% probability of falling within this range. In standard statistical terms, 95% of the data points typically fall within two standard deviations of the mean, which is a common measure of the spread or range of data.

```
import numpy as np
from numpy import array, zeros, power, ones, exp, multiply, eye, linspace, mat, spacing, s
from numpy.random import multivariate_normal

def build_Sigma(X, sigma2):
    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] = 1/(2*sigma2[l])*(X[i,l] - X[j,l])**2
    D = sum(D)
    D = D + D.T
    return exp(-D)

def plot_mvn( a=0, b=10, sigma2=1.0, size=1, n=100, show=True):
    X = np.linspace(a, b, n, endpoint=False).reshape(-1,1)
    sigma2 = np.array([sigma2])
    Sigma = build_Sigma(X, sigma2)
    rng = np.random.default_rng(seed=12345)
    Y = rng.multivariate_normal(zeros(Sigma.shape[0]), Sigma, size = size, check_valid="ra
    plt.plot(X, Y.T)
    plt.title("Realization of Random Functions under a GP prior.\n sigma2: {}".format(sigm
    if show:
        plt.show()
```

```
plot_mvn(a=0, b=10, sigma2=10.0, size=3, n=250)
```

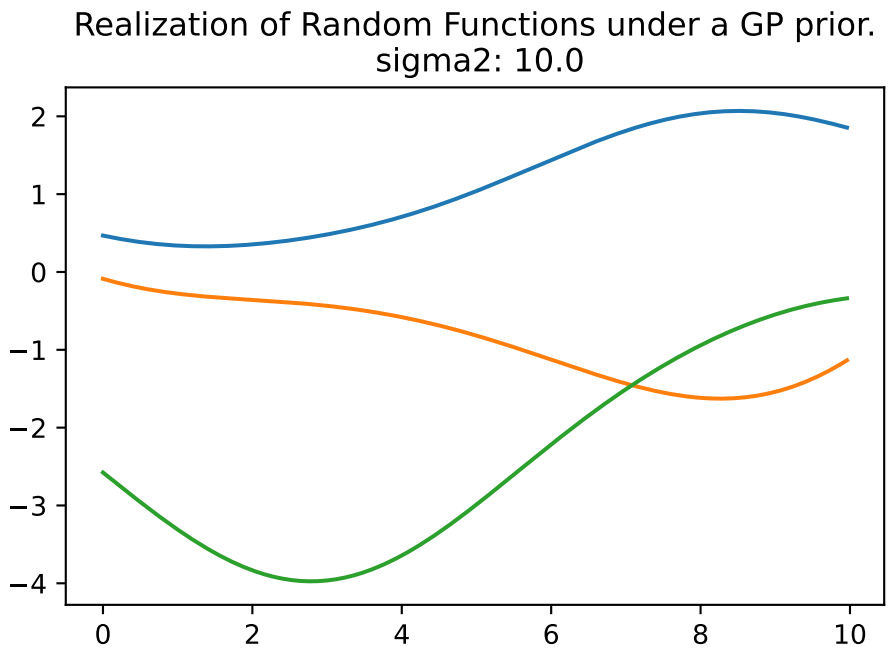


Figure 6.8: Realization of Random Functions under a GP prior. sigma2: 10

```
plot_mvn(a=0, b=10, sigma2=0.1, size=3, n=250)
```

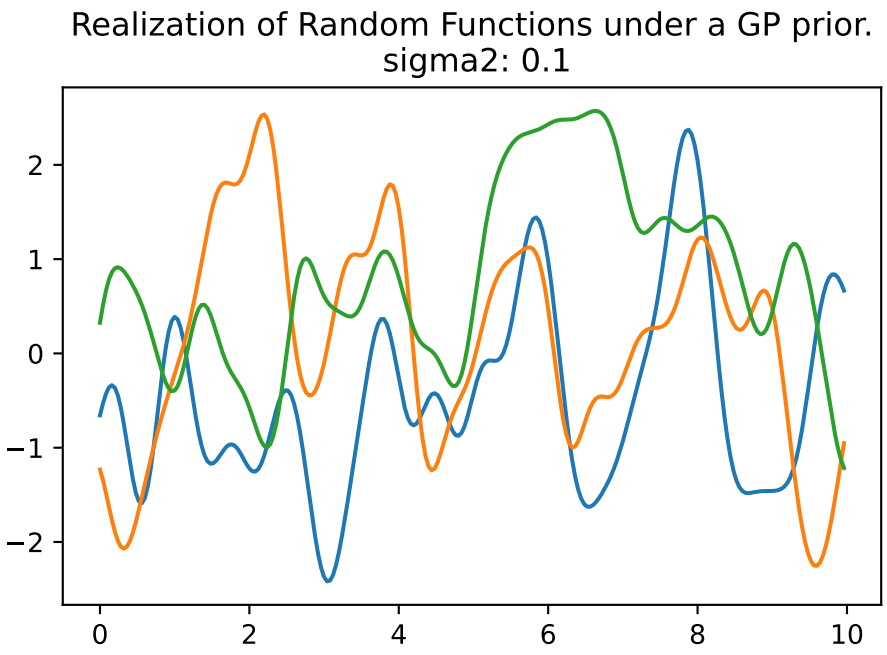


Figure 6.9: Realization of Random Functions under a GP prior. sigma2: 0.1

6.12 Kriging: Modeling Basics

6.12.1 The Kriging Basis Function

k-dimensional basis functions of the form

$$\psi(\vec{x}^{(i)}, \vec{x}^{(j)}) = \exp\left(-\sum_{l=1}^{k} \theta_l |x_l^{(i)} - x_l^{(j)}|^{p_l}\right)$$

are used in a method known as Kriging. Note, $\vec{x}^{(i)}$ denotes the k-dim vector $\vec{x}^{(i)} = (x_1^{(i)}, \ldots, x_k^{(i)})^T$.

The Kriging basis function is related to the 1-dim Gaussian basis function (Equation 6.5), which is defined as

$$\Sigma(\vec{x}^{(i)}, \vec{x}^{(j)}) = \exp\{-||\vec{x}^{(i)} - \vec{x}^{(j)}||^2/(2\sigma^2)\}. \tag{6.6}$$

There are some differences between Gaussian basis functions and Kriging basis functions:

- Where the Gaussian basis function has $1/(2\sigma^2)$, the Kriging basis has a vector $\theta = [\theta_1, \theta_2, \ldots, \theta_k]^T$.
- The θ vector allows the width of the basis function to vary from dimension to dimension.

- In the Gaussian basis function, the exponent is fixed at 2, Kriging allows this exponent p_l to vary (typically from 1 to 2).

6.12.2 The Correlation Coefficient

In a bivariate normal distribution, the covariance matrix and the correlation coefficient are closely related. The covariance matrix Σ for a bivariate normal distribution is a 2×2 matrix that looks like this:

$$\Sigma = \begin{pmatrix} \sigma_1^2 & \sigma_{12} \\ \sigma_{21} & \sigma_2^2 \end{pmatrix},$$

where σ_1^2 and σ_2^2 are the variances of X_1 and X_2, and $\sigma_{12} = \sigma_{21}$ is the covariance between X_1 and X_2.

The correlation coefficient, often denoted as ρ, is a normalized measure of the linear relationship between two variables. It is calculated from the covariance and the standard deviations σ_1 and σ_2 (or the square roots of the variances) of X_1 and X_2 as follows:

$$\rho = \sigma_{12}/(\sqrt{\sigma_1^2} \times \sqrt{\sigma_2^2}) = \sigma_{12}/(\sigma_1 \times \sigma_2).$$

So we can express the correlation coefficient ρ in terms of the elements of the covariance matrix Σ. It can be interpreted as follows: The correlation coefficient ranges from -1 to 1. A value of 1 means that X_1 and X_2 are perfectly positively correlated, a value of -1 means they are perfectly negatively correlated, and a value of 0 means they are uncorrelated. This gives the same information as the covariance, but on a standardized scale that does not depend on the units of X_1 and X_2.

6.12.3 Covariance Matrix and Correlation Matrix

i Covariance and Correlation (taken from @Forr08a)

Covariance is a measure of the correlation between two or more sets of random variables.

$$\text{Cov}(X, Y) = E[(X - E[X])(Y - E[Y])] = E[XY] - E[X]E[Y]$$

From the covariance, we can derive the correlation

$$\text{Corr}(X, Y) = \frac{\text{Cov}(X, Y)}{\sqrt{\text{Var}(X)\text{Var}(Y)}} = \frac{\text{Cov}(X, Y)}{\sigma_X \sigma_Y}. \tag{6.7}$$

For a vector of random variables

$$Y = \left((Y^{(l)}, \ldots, Y^{(n)})\right)^T$$

the covariance matrix is a matrix of covariances between the random variables

$$\Sigma = \mathrm{Cov}(Y, Y) = \begin{pmatrix} \mathrm{Cov}(Y^{(1)}, Y^{(1)}) & \ldots & \mathrm{Cov}(Y^{(1)}, Y^{(n)}) \\ \vdots & \ddots & \vdots \\ \mathrm{Cov}(Y^{(n)}, Y^{(1)}) & \ldots & \mathrm{Cov}(Y^{(n)}, Y^{(n)}) \end{pmatrix},$$

and from Equation 6.7

$$\mathrm{Cov}(Y, Y) = \sigma_Y^2 \mathrm{Cor}(Y, Y).$$

You can compute the correlation matrix Ψ from a covariance matrix Σ in Python using the numpy library. The correlation matrix is computed by dividing each element of the covariance matrix by the product of the standard deviations of the corresponding variables.

The function `covariance_to_correlation` first computes the standard deviations of the variables with `np.sqrt(np.diag(cov))`. It then computes the correlation matrix by dividing each element of the covariance matrix by the product of the standard deviations of the corresponding variables with `cov / np.outer(std_devs, std_devs)`.

```
import numpy as np

def covariance_to_correlation(cov):
    # Compute standard deviations
    std_devs = np.sqrt(np.diag(cov))

    # Compute correlation matrix
    corr = cov / np.outer(std_devs, std_devs)

    return corr

cov = np.array([[9, -4], [-4, 9]])
print(covariance_to_correlation(cov))
```

```
[[ 1.         -0.44444444]
 [-0.44444444  1.        ]]
```

6.12.4 Building the Kriging Correlation Matrix

Consider a set of k-dimensional sample data (sample size is n)

$$X = \{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n\},$$

i.e., X is a (n, k)-matrix, with related responses

$$\vec{y} = \{y_1, y_2, \dots, y_n\}.$$

We are interested in finding an expression for the predicted value at a new point $\vec{x}$. We are going to interpret the observed responses as if they are generated by a stochastic process, which will be denoted as follows:

$$Y = [Y(\vec{x}_1), \dots, Y(\vec{x}_n)]^T.$$

The random process has a mean of $\vec{\mu}$ (which is a $(n \times 1)$-dimensional vector). The n k-dimensional random variables are correlated with each other using the basis function expression

$$\text{cor}[Y(\vec{x}_i), Y(\vec{x}_j)] = \exp\left(-\sum_{l=1}^{k} \theta_l |x_{il} - x_{jl}|^{p_l}\right).$$

From this, the (n, n) correlation matrix Ψ of the observed data can be computed:

$$\Psi = \begin{pmatrix} \text{cor}[Y(\vec{x}_1), Y(\vec{x}_1)] & \dots & \text{cor}[Y(\vec{x}_1), Y(\vec{x}_n)] \\ \vdots & \ddots & \vdots \\ \text{cor}[Y(\vec{x}_n), Y(\vec{x}_1)] & \dots & \text{cor}[Y(\vec{x}_n), Y(\vec{x}_n)] \end{pmatrix}.$$

Example: 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:

$$D_1 = \begin{pmatrix} x_{11} - x_{11} & x_{11} - x_{21} & x_{11} - x_{31} & x_{11} - x_{41} \\ x_{21} - x_{11} & x_{21} - x_{21} & x_{21} - x_{31} & x_{21} - x_{41} \\ x_{31} - x_{11} & x_{31} - x_{21} & x_{31} - x_{31} & x_{31} - x_{41} \\ x_{41} - x_{11} & x_{41} - x_{21} & x_{41} - x_{31} & x_{41} - x_{41} \end{pmatrix}$$

$$D_2 = \begin{pmatrix} x_{12} - x_{12} & x_{12} - x_{22} & x_{12} - x_{32} & x_{12} - x_{42} \\ x_{22} - x_{12} & x_{22} - x_{22} & x_{22} - x_{32} & x_{22} - x_{42} \\ x_{32} - x_{12} & x_{32} - x_{22} & x_{32} - x_{32} & x_{32} - x_{42} \\ x_{42} - x_{12} & x_{42} - x_{22} & x_{42} - x_{32} & x_{42} - x_{42} \end{pmatrix}$$

$$D_3 = \begin{pmatrix} x_{13} - x_{13} & x_{13} - x_{23} & x_{13} - x_{33} & x_{13} - x_{43} \\ x_{23} - x_{13} & x_{23} - x_{23} & x_{23} - x_{33} & x_{23} - x_{43} \\ x_{33} - x_{13} & x_{33} - x_{23} & x_{33} - x_{33} & x_{33} - x_{43} \\ x_{43} - x_{13} & x_{43} - x_{23} & x_{43} - x_{33} & x_{43} - x_{43} \end{pmatrix}$$

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:

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

Finally,

$$\Psi = \exp(-D)$$

is computed.
Next, we will demonstrate how this computation can be implemented in Python.

```
from numpy import (array, zeros, power, ones, exp, multiply,
                   eye, linspace, mat, spacing, sqrt, arange,
                   append, ravel)
from numpy.linalg import cholesky, solve
theta = np.array([1,2,3])
X = np.array([ [1,0,0], [0,1,0], [100, 100, 100], [101, 100, 100]])
X
```

```
array([[  1,   0,   0],
       [  0,   1,   0],
       [100, 100, 100],
       [101, 100, 100]])
```

```
def build_Psi(X, theta):
    n = X.shape[0]
    k = X.shape[1]
    D = zeros((k, n, n))
    for l in range(k):
        for i in range(n):
            for j in range(i, n):
                D[l, i, j] = theta[l]*(X[i,l] - X[j,l])**2
    D = sum(D)
    D = D + D.T
    return exp(-D)
```

```
Psi = build_Psi(X, theta)
Psi
```

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

Example: The Correlation Matrix (Using Existing Functions)

The same result as computed in the previous example can be obtained with existing python functions, e.g., from the package `scipy`.

```
from scipy.spatial.distance import squareform
from scipy.spatial.distance import pdist

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.        ]])
```

6.12.5 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:  200000000.53159264
```

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

```
2.163953413738652
```

6.13 Kriging Modeling and Prediction in a Nutshell

We will use the Kriging correlation Ψ to predict new values based on the observed data. The matrix algebra involved for calculating the likelihood is the most computationally intensive part of the Kriging process. Care must be taken that the computer code is as efficient as possible.

Basic elements of the Kriging based surrogate optimization such as interpolation, expected improvement, and regression are presented. The presentation follows the approach described in Forrester, Sóbester, and Keane (2008) and Bartz et al. (2022).

6.13.1 The Kriging Model

Consider sample data $\vec{X}$ and $\vec{y}$ from n locations that are available in matrix form: $\vec{X}$ is a $(n \times k)$ matrix, where k denotes the problem dimension and $\vec{y}$ is a $(n \times 1)$ vector.

The observed responses $\vec{y}$ are considered as if they are from a stochastic process, which will be denoted as

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

The set of random vectors (also referred to as a *random field*) has a mean of $\vec{1}\mu$, which is a $(n \times 1)$ vector.

6.13.2 Correlations

The random vectors are correlated with each other using the basis function expression

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

The $(n \times n)$ correlation matrix of the observed sample data is

$$\vec{\Psi} = \begin{pmatrix} \text{cor}\left(\vec{Y}(\vec{x}^{(i)}), \vec{Y}(\vec{x}^{(l)})\right) & \dots & \text{cor}\left(\vec{Y}(\vec{x}^{(i)}), \vec{Y}(\vec{x}^{(l)})\right) \\ \vdots & \vdots & \vdots \\ \text{cor}\left(\vec{Y}(\vec{x}^{(i)}), \vec{Y}(\vec{x}^{(l)})\right) & \dots & \text{cor}\left(\vec{Y}(\vec{x}^{(i)}), \vec{Y}(\vec{x}^{(l)})\right) \end{pmatrix}.$$

Note: correlations depend on the absolute distances between sample points $|x_j^{(n)} - x_j^{(n)}|$ and the parameters p_j and θ_j.

Correlation is intuitive, because when two points move close together, then $|x_l^{(i)} - x_l| \to 0$ and $\exp(-|x_l^{(i)} - x_l| \to 1$, points show very close correlation and $Y(x_l^{(i)}) = Y(x_l)$.

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 6.10.

```
visualize_inverse_exp_squared_distance(5, 0, theta_values=[0.5, 1, 2.0])
```

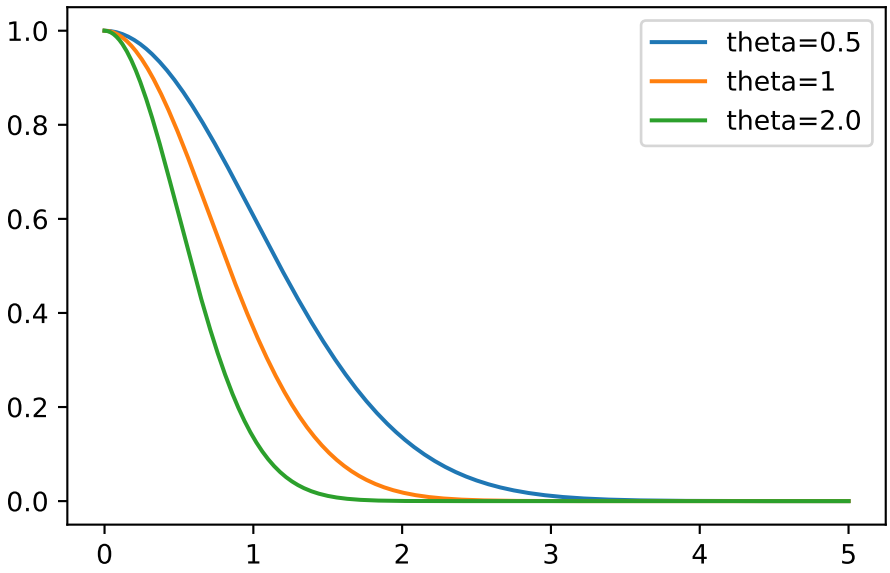


Figure 6.10: Theta set to 1/2, 1, and 2

6.13.3 MLE to estimate θ and p

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 $\vec{y}$, which can be expressed in terms of the sample data

$$L\left(\vec{Y}(\vec{x}^{(1)}),\ldots,\vec{Y}(\vec{x}^{(n)})|\mu,\sigma\right) = \frac{1}{(2\pi\sigma)^{n/2}|\vec{\Psi}|^{1/2}}\exp\left\{\frac{-(\vec{y}-\vec{1}\mu)^T\vec{\Psi}^{-1}(\vec{y}-\vec{1}\mu)}{2\sigma^2}\right\},$$

and formulated as the log-likelihood:

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

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

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

and

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

Combining the equations leads to the concentrated log-likelihood:

$$\ln(L) = -\frac{n}{2}\ln(\hat{\sigma}) - \frac{1}{2}\ln|\vec{\Psi}|. \tag{6.8}$$

> **Note: The Concentrated Log-Likelihood**
>
> - The first term in Equation 6.8 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 equation is not differentiable.

6.13.4 Tuning 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 is build with the optimized θ and p values. This is best (most likely) Kriging model for the given data y. We will skip the optimization step here and use $\theta = 1$ and $p = 2$.

6.13.5 Kriging Prediction

- Main idea for prediction:
  - 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 $\vec{X}$ 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 $\tilde{\vec{y}} = (\vec{y}^T, \hat{y})^T$.
- A vector of correlations between the observed data and the new prediction is defined as

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

The augmented correlation matrix is constructed as

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

The log-likelihood of the augmented data is

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

The MLE for $\hat{y}$ can be calculated as

$$\hat{y}(\vec{x}) = \hat{\mu} + \vec{\psi}^T \vec{\vec{\Psi}}^{-1} (\vec{y} - \vec{1}\hat{\mu}). \tag{6.9}$$

6.13.6 Properties of the Predictor

This 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 $\vec{\Psi}^{-1}$ is $\vec{\psi}$, and $\vec{\psi}\vec{\Psi}^{-1}$ is the ith unit vector. Hence, $\hat{y}(\vec{x}^{(i)}) = y^{(i)}$.

i 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.

1. Calculating the Correlation Matrix Ψ

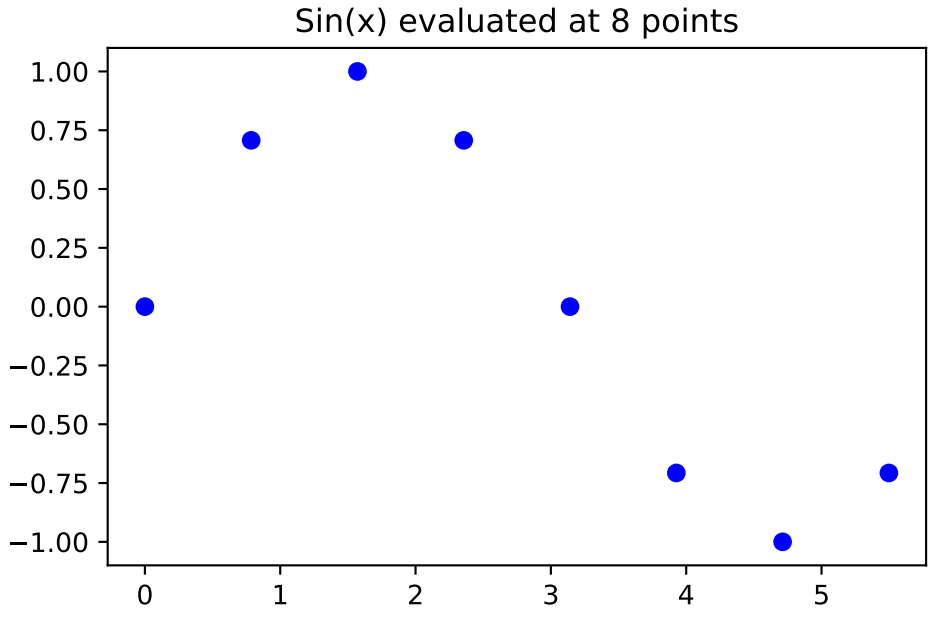
The correlation matrix Ψ is based on the pairwise squared distances between the input locations. Here we will use $n = 8$ sample locations and θ is set to 1.0.

```
n = 8
X = np.linspace(0, 2*np.pi, n, endpoint=False).reshape(-1,1)
# theta should be an array (of one value, for the moment, will be changed later)
theta = np.array([1.0])
Psi = build_Psi(X, theta)
```

Evaluate at sample points

```
y = np.sin(X)
```

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



2. Computing the ψ Vector
Distances between testing locations x and training data locations X.

```
from scipy.spatial.distance import cdist

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)
    print(D.shape)
    psi = exp(-D)
    # return psi transpose to be consistent with the literature
    return(psi.T)
```

3. Predicting at New Locations

We would like to predict at $m = 100$ new 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)
psi = build_psi(X, x, theta)
```

```
(100, 8)
```

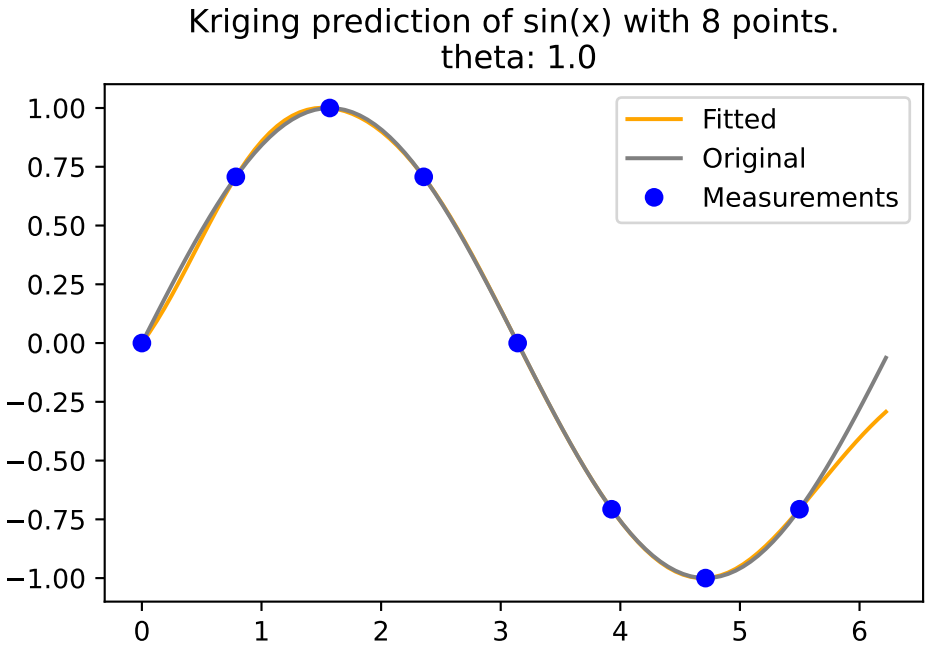
Computation of the predictive equations.

```
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)))
```

To compute f, Equation 6.9 is used.

4. Visualization

```
import matplotlib.pyplot as plt
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()
```



7 Introduction to `spotPython`

Surrogate model based optimization methods are common approaches in simulation and optimization. SPOT was developed because there is a great need for sound statistical analysis of simulation and optimization algorithms. SPOT includes methods for tuning based on classical regression and analysis of variance techniques. It presents tree-based models such as classification and regression trees and random forests as well as Bayesian optimization (Gaussian process models, also known as Kriging). Combinations of different meta-modeling approaches are possible. SPOT comes with a sophisticated surrogate model based optimization method, that can handle discrete and continuous inputs. Furthermore, any model implemented in `scikit-learn` can be used out-of-the-box as a surrogate in `spotPython`.

SPOT implements key techniques such as exploratory fitness landscape analysis and sensitivity analysis. It can be used to understand the performance of various algorithms, while simultaneously giving insights into their algorithmic behavior.

The `spot` loop consists of the following steps:

1. Init: Build initial design X
2. Evaluate initial design on real objective f: $y = f(X)$
3. Build surrogate: $S = S(X, y)$
4. Optimize on surrogate: $X_0 = \text{optimize}(S)$
5. Evaluate on real objective: $y_0 = f(X_0)$
6. Impute (Infill) new points: $X = X \cup X_0$, $y = y \cup y_0$.
7. Got 3.

Central Idea: Evaluation of the surrogate model `S` is much cheaper (or / and much faster) than running the real-world experiment f. We start with a small example.

7.1 Example: `Spot` and the Sphere Function

```
import numpy as np
from math import inf
from spotPython.fun.objectivefunctions import analytical
from spotPython.spot import spot
from scipy.optimize import shgo
```

```
from scipy.optimize import direct
from scipy.optimize import differential_evolution
import matplotlib.pyplot as plt
```

7.1.1 The Objective Function: Sphere

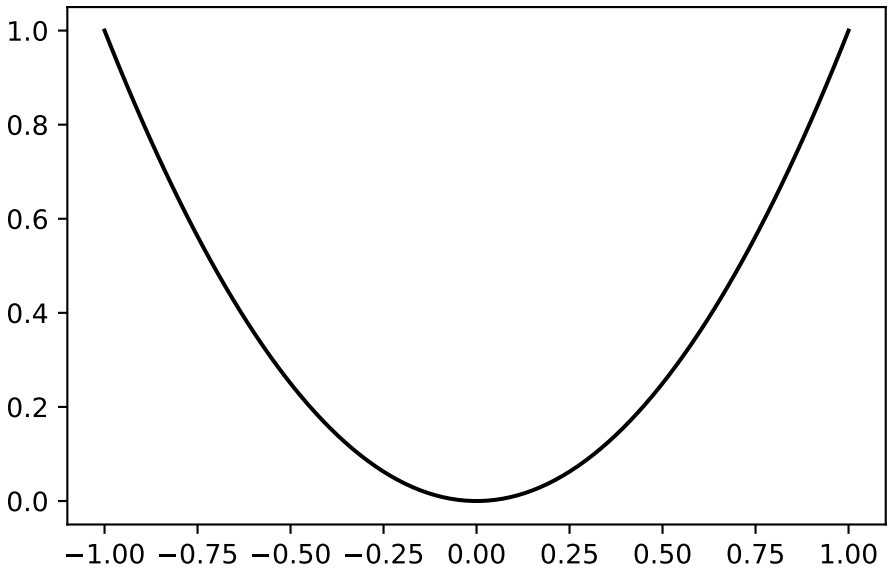
The `spotPython` package provides several classes of objective functions. We will use an analytical objective function, i.e., a function that can be described by a (closed) formula:

$$f(x) = x^2$$

```
fun = analytical().fun_sphere
```

We can apply the function `fun` to input values and plot the result:

```
x = np.linspace(-1,1,100).reshape(-1,1)
y = fun(x)
plt.figure()
plt.plot(x, y, "k")
plt.show()
```



```
spot_0 = spot.Spot(fun=fun,
                   lower = np.array([-1]),
                   upper = np.array([1]))
```

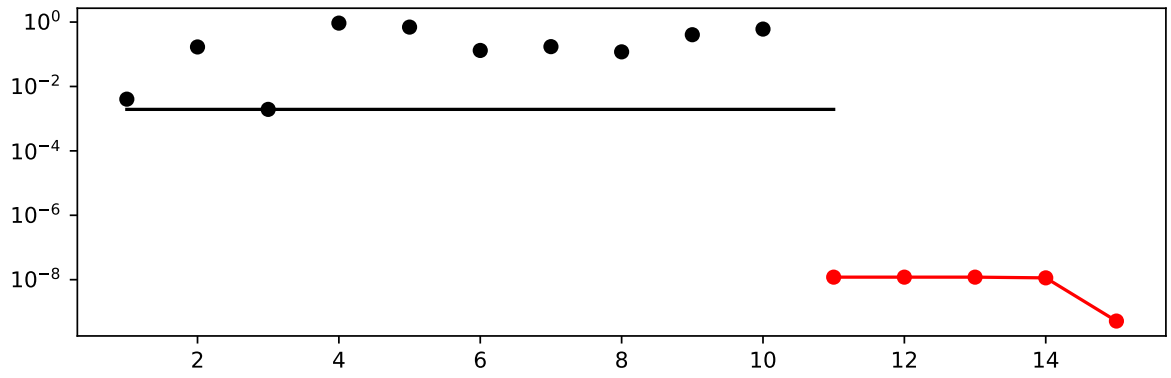
```
spot_0.run()
```

```
spotPython tuning: 1.1986325668379847e-08 [#######---] 73.33%

spotPython tuning: 1.1986325668379847e-08 [########--] 80.00%

spotPython tuning: 1.1986325668379847e-08 [#########-] 86.67%

spotPython tuning: 1.1385311249270152e-08 [#########-] 93.33%

spotPython tuning: 5.189351944607845e-10 [##########] 100.00% Done...

<spotPython.spot.spot.Spot at 0x145c9b850>
```

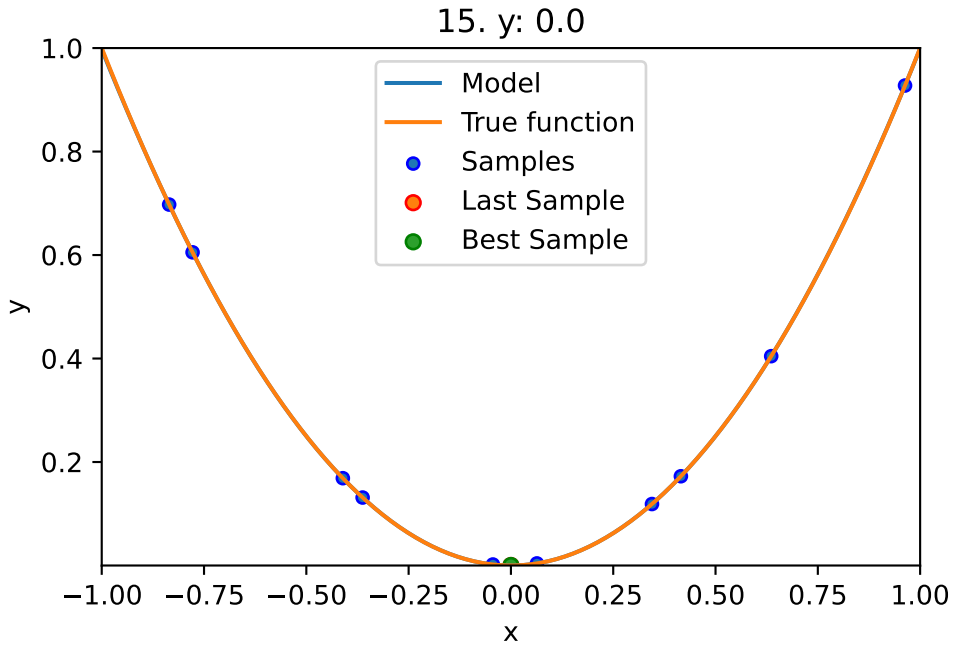
```
spot_0.print_results()
```

```
min y: 5.189351944607845e-10
x0: 2.2780149131662516e-05

[['x0', 2.2780149131662516e-05]]
```

```
spot_0.plot_progress(log_y=True)
```



```
spot_0.plot_model()
```



7.2 `Spot` Parameters: `fun_evals`, `init_size` and `show_models`

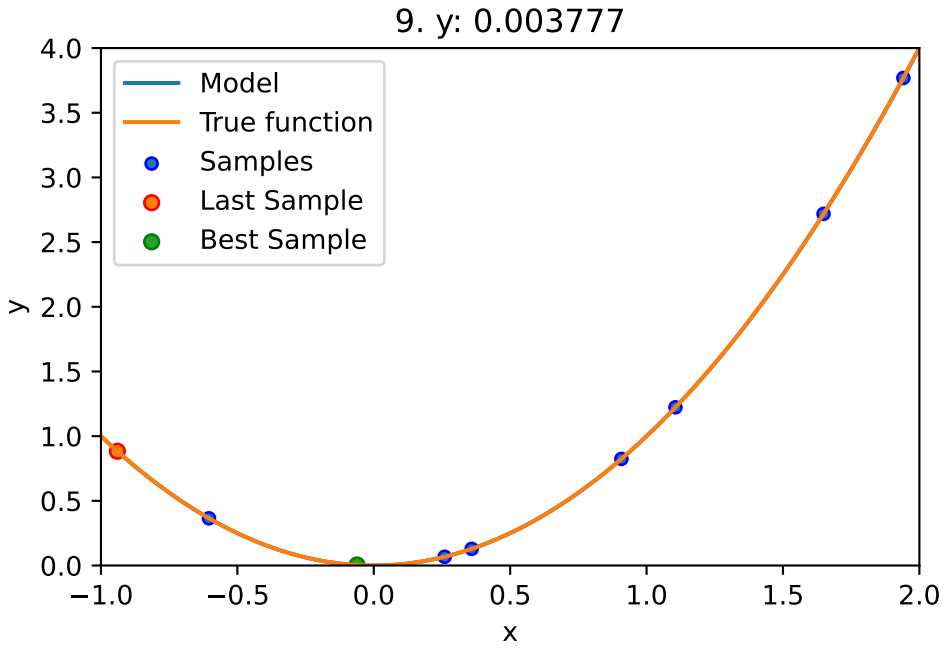
We will modify three parameters:

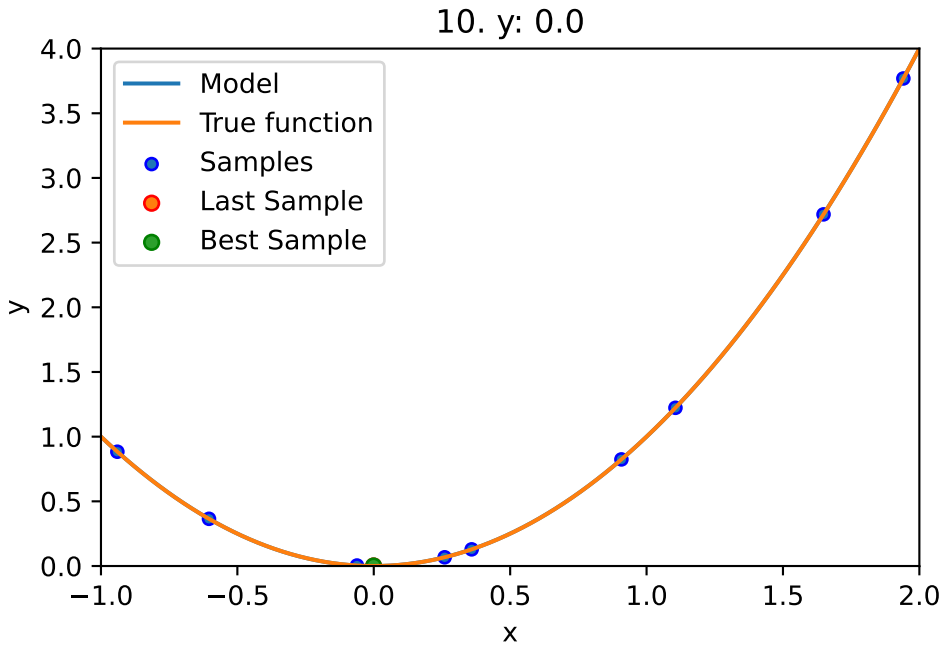
1. The number of function evaluations (`fun_evals`)
2. The size of the initial design (`init_size`)
3. The parameter `show_models`, which visualizes the search process for 1-dim functions.

The full list of the `Spot` parameters is shown in the Help System and in the notebook `spot_doc.ipynb`.

```
spot_1 = spot.Spot(fun=fun,
                   lower = np.array([-1]),
                   upper = np.array([2]),
                   fun_evals= 10,
                   seed=123,
                   show_models=True,
                   design_control={"init_size": 9})
```

```
spot_1.run()
```


9. y: 0.003777
Model
True function
Samples
Last Sample
Best Sample
y
x
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0


10. y: 0.0
Model
True function
Samples
Last Sample
Best Sample
y
x
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

```
spotPython tuning: 3.675700131680327e-07 [##########] 100.00% Done...

<spotPython.spot.spot.Spot at 0x156168b50>
```

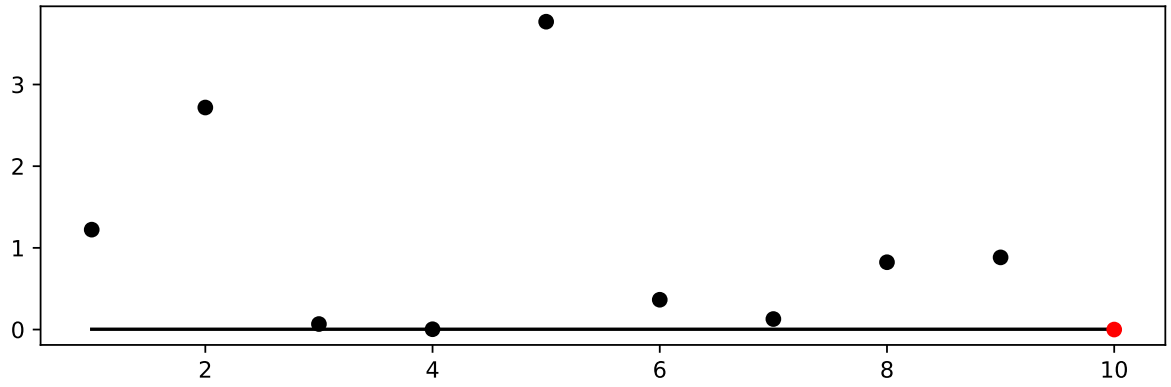
7.3 Print the Results

```
spot_1.print_results()
```

```
min y: 3.675700131680327e-07
x0: -0.0006062755257867768

[['x0', -0.0006062755257867768]]
```

7.4 Show the Progress

```
spot_1.plot_progress()
```



7.5 Visualizing the Optimization and Hyperparameter Tuning Process with TensorBoard

`spotPython` supports the visualization of the hyperparameter tuning process with TensorBoard. The following example shows how to use TensorBoard with `spotPython`.

First, we define an “experiment name” to identify the hyperparameter tuning process. The experiment name is used to create a directory for the TensorBoard files.

```
from spotPython.utils.file import get_experiment_name
from spotPython.utils.init import fun_control_init
from spotPython.utils.file import get_spot_tensorboard_path

PREFIX = "01"
experiment_name = get_experiment_name(prefix=PREFIX)
print(experiment_name)

fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name))
```

```
01_p040025_2023-12-30_22-56-49
```

Since the `spot_tensorboard_path` is defined, `spotPython` will log the optimization process in the TensorBoard files. The TensorBoard files are stored in the directory `spot_tensorboard_path`. We can pass the TensorBoard information to the `Spot` method via the `fun_control` dictionary.

```
spot_tuner = spot.Spot(fun=fun,
                   lower = np.array([-1]),
                   upper = np.array([2]),
                   fun_evals= 10,
                   seed=123,
                   show_models=False,
                   design_control={"init_size": 5},
                   fun_control=fun_control,)
spot_tuner.run()
```

```
spotPython tuning: 2.7704798574631272e-05 [######----] 60.00%

spotPython tuning: 7.761255945313609e-07 [#######---] 70.00%

spotPython tuning: 7.729540372746933e-07 [########--] 80.00%

spotPython tuning: 3.677408011584826e-07 [#########-] 90.00%

spotPython tuning: 1.1649997893485877e-09 [##########] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2da225a90>
```

Now we can start TensorBoard in the background. The TensorBoard process will read the TensorBoard files and visualize the hyperparameter tuning process. From the terminal, we can start TensorBoard with the following command:

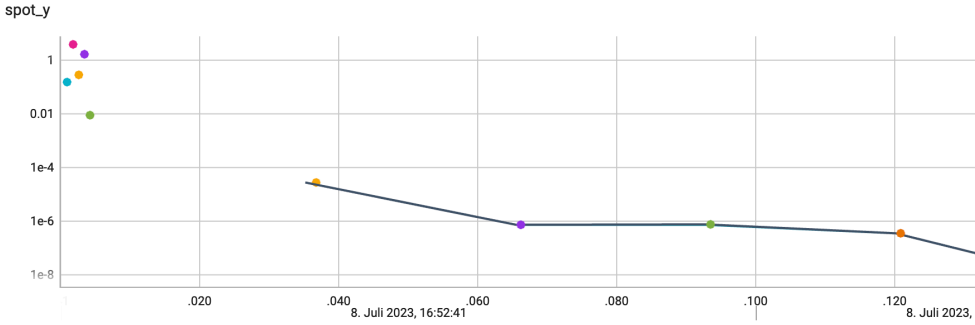
```
tensorboard --logdir="./runs"
```

`logdir` is the directory where the TensorBoard files are stored. In our case, the TensorBoard files are stored in the directory `./runs`.

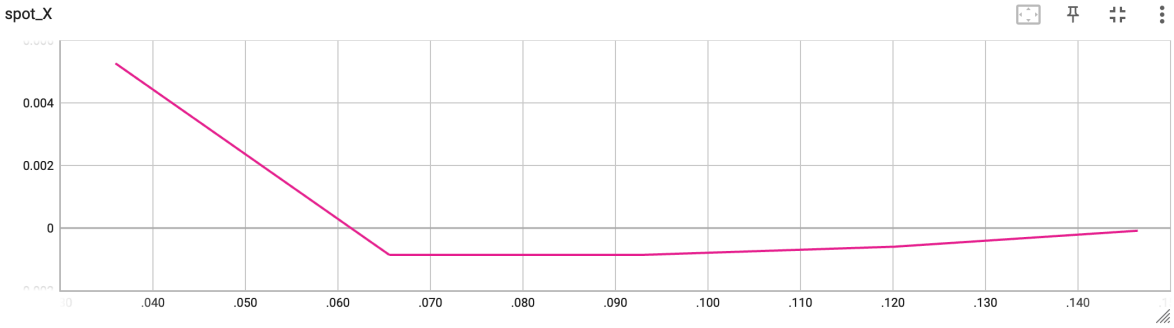
TensorBoard will start a web server on port 6006. We can access the TensorBoard web server with the following URL:

```
http://localhost:6006/
```

The first TensorBoard visualization shows the objective function values plotted against the wall time. The wall time is the time that has passed since the start of the hyperparameter tuning process. The five initial design points are shown in the upper left region of the plot. The line visualizes the optimization process.



The second TensorBoard visualization shows the input values, i.e., x_0, plotted against the wall time.



The third TensorBoard plot illustrates how `spotPython` can be used as a microscope for the internal mechanisms of the surrogate-based optimization process. Here, one important

parameter, the learning rate θ of the Kriging surrogate is plotted against the number of optimization steps.

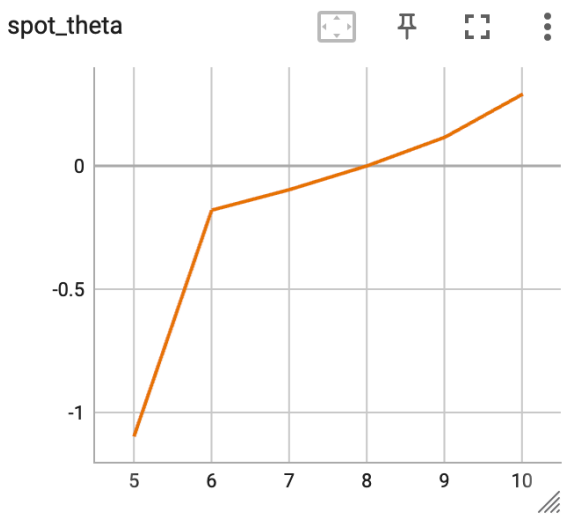


Figure 7.1: TensorBoard visualization of the spotPython process.

8 Multi-dimensional Functions

This chapter illustrates how high-dimensional functions can be optimzed and analyzed.

8.1 Example: `Spot` and the 3-dim Sphere Function

```
import numpy as np
from spotPython.fun.objectivefunctions import analytical
from spotPython.spot import spot
```

8.1.1 The Objective Function: 3-dim Sphere

- The `spotPython` package provides several classes of objective functions.
- We will use an analytical objective function, i.e., a function that can be described by a (closed) formula:

$$f(x) = \sum_{i}^{n} x_i^2$$

- Here we will use $n = 3$.

```
fun = analytical().fun_sphere
```

- The size of the `lower` bound vector determines the problem dimension.
- Here we will use `-1.0 * np.ones(3)`, i.e., a three-dim function.
- We will use three different `theta` values (one for each dimension), i.e., we set `surrogate_control={"n_theta": 3}`.

> **TensorBoard**
>
> Similar to the one-dimensional case, which was introduced in Section Section 7.5, we can use TensorBoard to monitor the progress of the optimization. We will use the same code,

only the prefix is different:

```
from spotPython.utils.file import get_experiment_name
from spotPython.utils.init import fun_control_init
from spotPython.utils.file import get_spot_tensorboard_path

PREFIX = "02"
experiment_name = get_experiment_name(prefix=PREFIX)
print(experiment_name)

fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name))
```

```
02_p040025_2023-12-30_22-57-04
```

```
spot_3 = spot.Spot(fun=fun,
                   lower = -1.0*np.ones(3),
                   upper = np.ones(3),
                   var_name=["Pressure", "Temp", "Lambda"],
                   show_progress=True,
                   surrogate_control={"n_theta": 3},
                   fun_control=fun_control,
                )

spot_3.run()
```

```
spotPython tuning: 0.03443414446852215 [#######---] 73.33%

spotPython tuning: 0.031346790207570745 [########--] 80.00%

spotPython tuning: 0.0009630779801151435 [#########-] 86.67%

spotPython tuning: 8.57685255248343e-05 [#########-] 93.33%

spotPython tuning: 7.337173529617274e-05 [##########] 100.00% Done...

<spotPython.spot.spot.Spot at 0x2d32e6690>
```

Now we can start TensorBoard in the background with the following command:

```
tensorboard --logdir="./runs"
```

We can access the TensorBoard web server with the following URL:

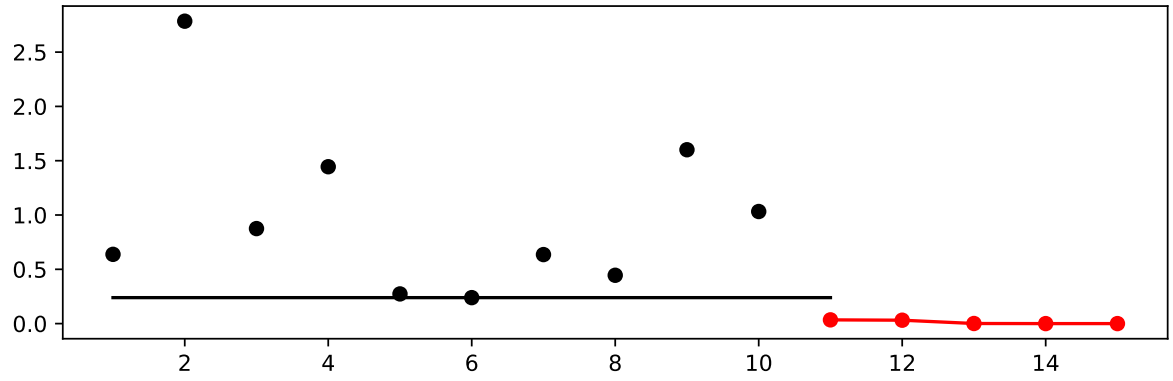
```
http://localhost:6006/
```

8.1.2 Results

```
spot_3.print_results()
```

```
min y: 7.337173529617274e-05
Pressure: 0.005530391439119171
Temp: 0.001931974999553984
Lambda: 0.00624931823700702

[['Pressure', 0.005530391439119171],
 ['Temp', 0.001931974999553984],
 ['Lambda', 0.00624931823700702]]
```

```
spot_3.plot_progress()
```

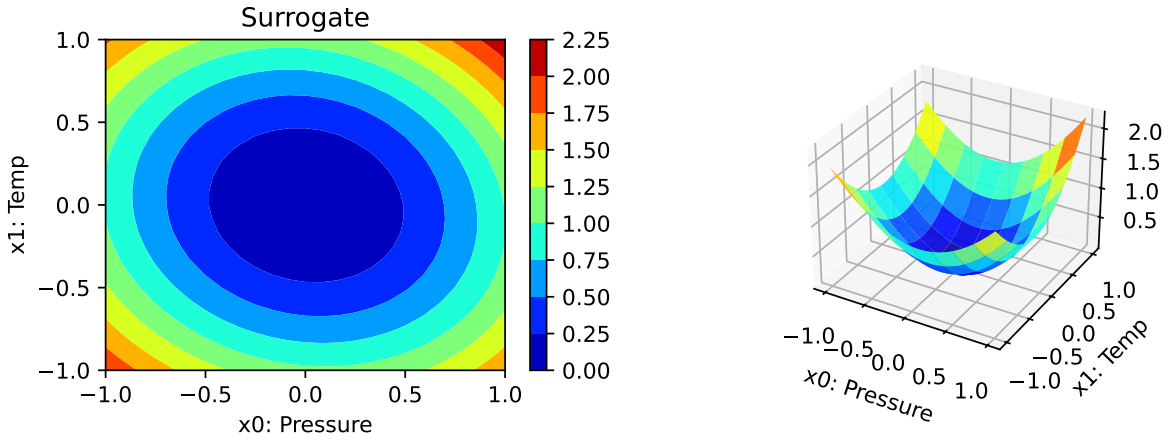


8.1.3 A Contour Plot

- We can select two dimensions, say $i = 0$ and $j = 1$, and generate a contour plot as follows.

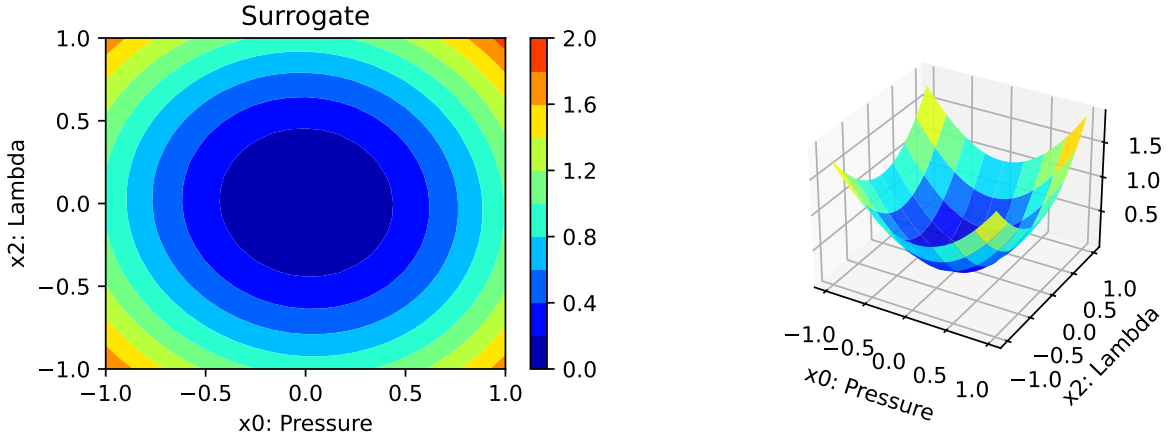
– Note: We have specified identical `min_z` and `max_z` values to generate comparable plots!

```
spot_3.plot_contour(i=0, j=1, min_z=0, max_z=2.25)
```



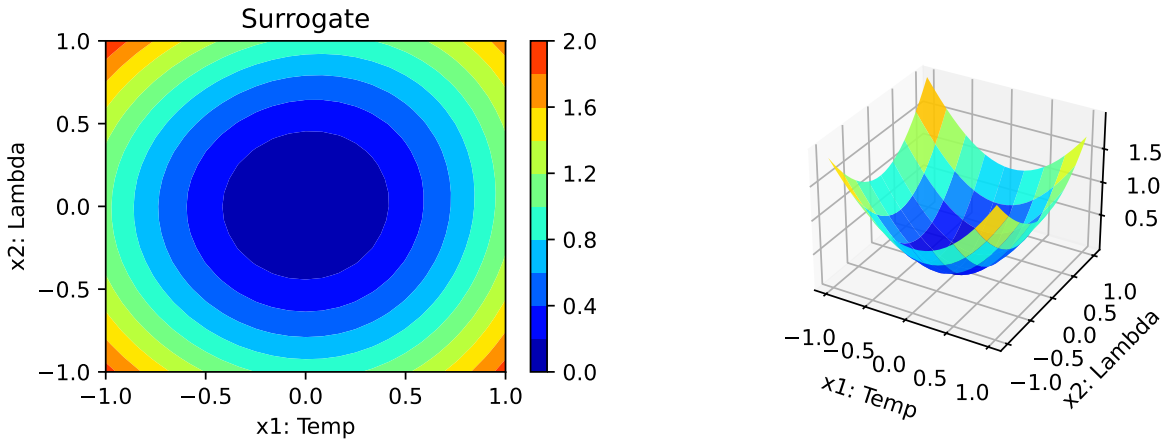
- In a similar manner, we can plot dimension $i = 0$ and $j = 2$:

```
spot_3.plot_contour(i=0, j=2, min_z=0, max_z=2.25)
```



- The final combination is $i = 1$ and $j = 2$:

```
spot_3.plot_contour(i=1, j=2, min_z=0, max_z=2.25)
```



- The three plots look very similar, because the `fun_sphere` is symmetric.
- This can also be seen from the variable importance:

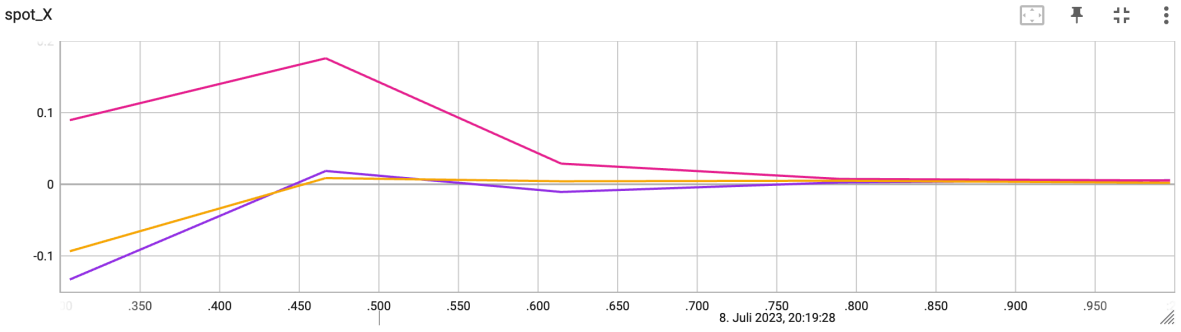
```
spot_3.print_importance()
```

```
Pressure:  95.28112765785582
Temp:  100.0
Lambda:  86.8087715378692
```

```
[['Pressure', 95.28112765785582],
 ['Temp', 100.0],
 ['Lambda', 86.8087715378692]]
```

8.1.4 TensorBoard

The second TensorBoard visualization shows the input values, i.e., $x_0, \ldots, x_2$, plotted against the wall time.



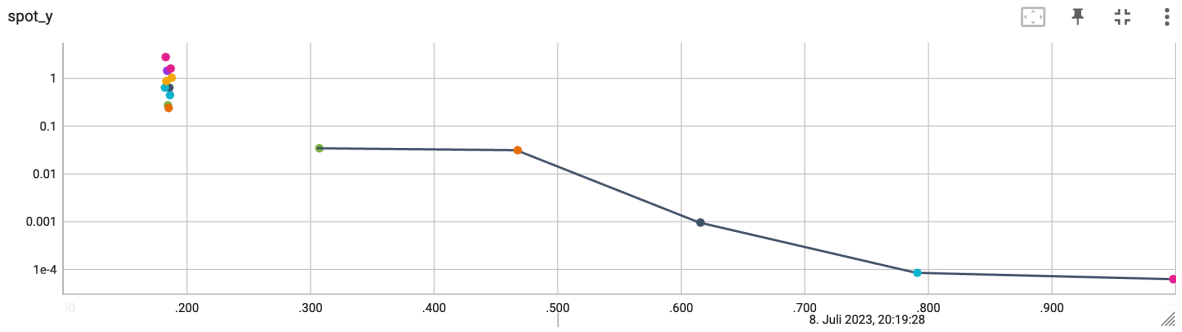


Figure 8.1: TensorBoard visualization of the spotPython process. Objective function values plotted against wall time.

The third TensorBoard plot illustrates how `spotPython` can be used as a microscope for the internal mechanisms of the surrogate-based optimization process. Here, one important parameter, the learning rate θ of the Kriging surrogate is plotted against the number of optimization steps.

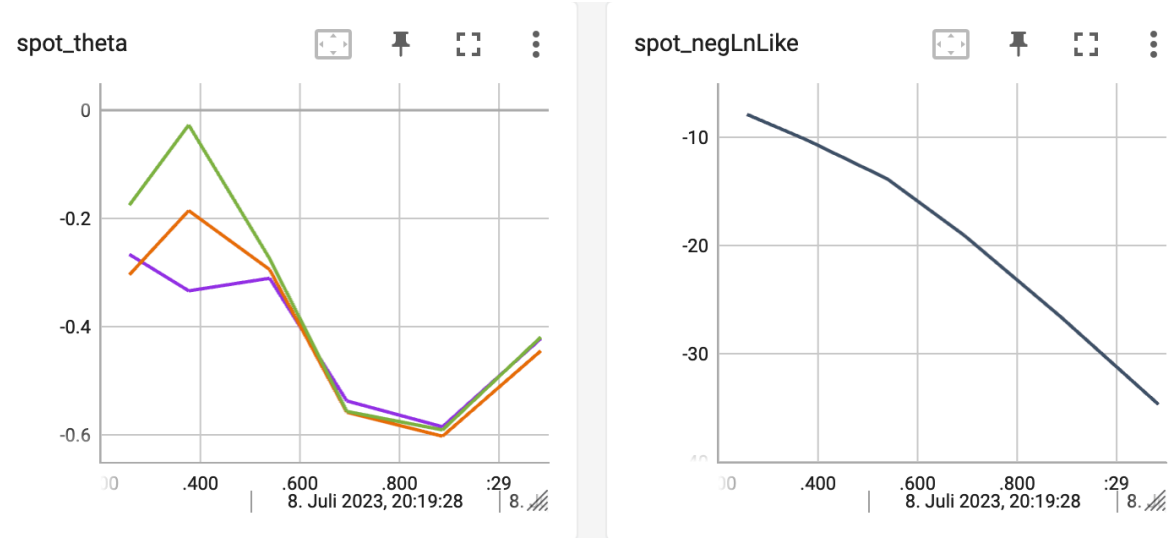


Figure 8.2: TensorBoard visualization of the spotPython surrogate model.

8.2 Conclusion

Based on this quick analysis, we can conclude that all three dimensions are equally important (as expected, because the analytical function is known).

8.3 Exercises

- Important:
  - Results from these exercises should be added to this document, i.e., you should submit an updated version of this notebook.
  - Please combine your results using this notebook.
  - Only one notebook from each group!
  - Presentation is based on this notebook. No addtional slides are required!
  - spotPython version `0.16.11` (or greater) is required

8.3.1 The Three Dimensional `fun_cubed`

- The input dimension is `3`. The search range is $-1 \leq x \leq 1$ for all dimensions.
- Generate contour plots
- Calculate the variable importance.
- Discuss the variable importance:
  - Are all variables equally important?
  - If not:
    - Which is the most important variable?
    - Which is the least important variable?

8.3.2 The Ten Dimensional `fun_wing_wt`

- The input dimension is `10`. The search range is $0 \leq x \leq 1$ for all dimensions.
- Calculate the variable importance.
- Discuss the variable importance:
  - Are all variables equally important?
  - If not:
    - Which is the most important variable?
    - Which is the least important variable?
  - Generate contour plots for the three most important variables. Do they confirm your selection?

8.3.3 The Three Dimensional `fun_runge`

- The input dimension is `3`. The search range is $-5 \leq x \leq 5$ for all dimensions.
- Generate contour plots
- Calculate the variable importance.

- Discuss the variable importance:
  - Are all variables equally important?
  - If not:
    - Which is the most important variable?
    - Which is the least important variable?

8.3.4 The Three Dimensional `fun_linear`

- The input dimension is `3`. The search range is $-5 \leq x \leq 5$ for all dimensions.
- Generate contour plots
- Calculate the variable importance.
- Discuss the variable importance:
  - Are all variables equally important?
  - If not:
    - Which is the most important variable?
    - Which is the least important variable?

9 Isotropic and Anisotropic Kriging

This chapter illustrates the difference between isotropic and anisotropic Kriging models. The difference is illustrated with the help of the `spotPython` package. Isotropic Kriging models use the same `theta` value for every dimension. Anisotropic Kriging models use different `theta` values for each dimension.

9.1 Example: Isotropic `Spot` Surrogate and the 2-dim Sphere Function

```
import numpy as np
from math import inf
from spotPython.fun.objectivefunctions import analytical
from spotPython.spot import spot
```

9.1.1 The Objective Function: 2-dim Sphere

- The `spotPython` package provides several classes of objective functions.
- We will use an analytical objective function, i.e., a function that can be described by a (closed) formula:

$$f(x, y) = x^2 + y^2$$

```
fun = analytical().fun_sphere
fun_control = {"sigma": 0,
               "seed": 123}
```

- The size of the `lower` bound vector determines the problem dimension.
- Here we will use `np.array([-1, -1])`, i.e., a two-dim function.

```
spot_2 = spot.Spot(fun=fun,
                   lower = np.array([-1, -1]),
                   upper = np.array([1, 1]))
```

```
spot_2.run()
```

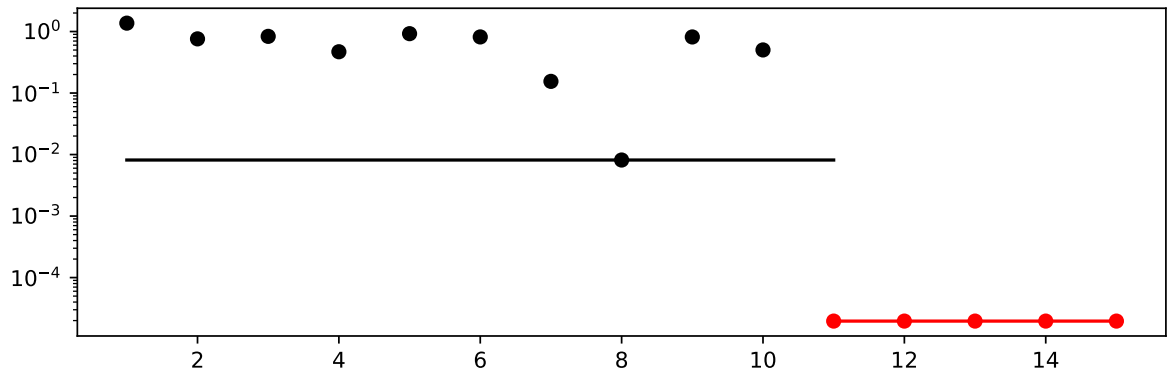
```
spotPython tuning: 1.961941240809186e-05 [#######---] 73.33%

spotPython tuning: 1.961941240809186e-05 [########--] 80.00%

spotPython tuning: 1.961941240809186e-05 [#########-] 86.67%

spotPython tuning: 1.961941240809186e-05 [#########-] 93.33%

spotPython tuning: 1.961941240809186e-05 [##########] 100.00% Done...

<spotPython.spot.spot.Spot at 0x13533c610>
```

9.1.2 Results

```
spot_2.print_results()
```

```
min y: 1.961941240809186e-05
x0: 0.0016552607778336345
x1: 0.004108469808268944

[['x0', 0.0016552607778336345], ['x1', 0.004108469808268944]]
```

```
spot_2.plot_progress(log_y=True)
```



9.2 Example With Anisotropic Kriging

- The default parameter setting of `spotPython`'s Kriging surrogate uses the same `theta` value for every dimension.
- This is referred to as "using an isotropic kernel".
- If different `theta` values are used for each dimension, then an anisotropic kernel is used
- To enable anisotropic models in `spotPython`, the number of `theta` values should be larger than one.
- We can use `surrogate_control={"n_theta": 2}` to enable this behavior (2 is the problem dimension).

> **TensorBoard**
>
> Similar to the one-dimensional case, which was introduced in Section Section 7.5, we can use TensorBoard to monitor the progress of the optimization. We will use the same code, only the prefix is different:
>
> ```
> from spotPython.utils.file import get_experiment_name
> from spotPython.utils.init import fun_control_init
> from spotPython.utils.file import get_spot_tensorboard_path
>
> PREFIX = "03"
> experiment_name = get_experiment_name(prefix=PREFIX)
> print(experiment_name)
>
> fun_control = fun_control_init(
>     spot_tensorboard_path=get_spot_tensorboard_path(experiment_name))
> ```
>
> ```
> 03_p040025_2023-12-30_22-57-23
> ```

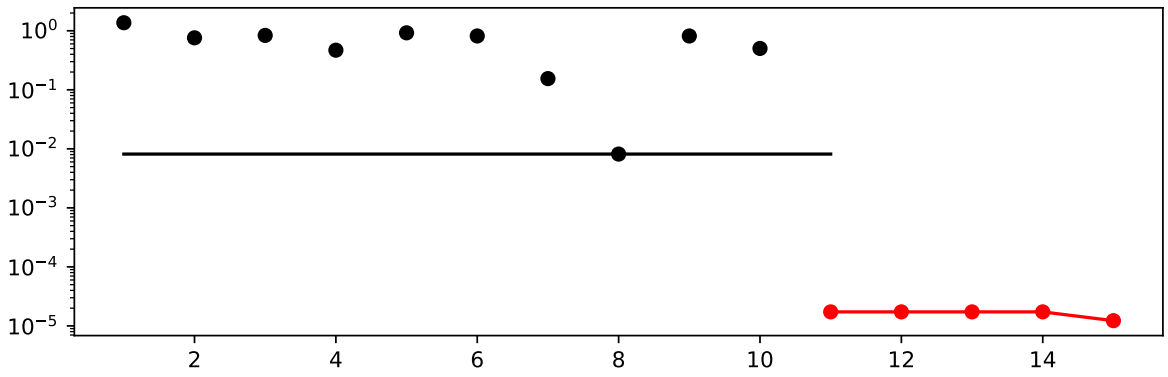
```
spot_2_anisotropic = spot.Spot(fun=fun,
                    lower = np.array([-1, -1]),
                    upper = np.array([1, 1]),
                    surrogate_control={"n_theta": 2},
                    fun_control=fun_control)
spot_2_anisotropic.run()
```

```
spotPython tuning: 1.7290078116309417e-05 [#######---] 73.33%

spotPython tuning: 1.7290078116309417e-05 [########--] 80.00%
```

```
spotPython tuning: 1.7290078116309417e-05 [#########-] 86.67%

spotPython tuning: 1.7290078116309417e-05 [#########-] 93.33%

spotPython tuning: 1.2267580433005057e-05 [##########] 100.00% Done...

<spotPython.spot.spot.Spot at 0x144ab0910>
```

- The search progress of the optimization with the anisotropic model can be visualized:

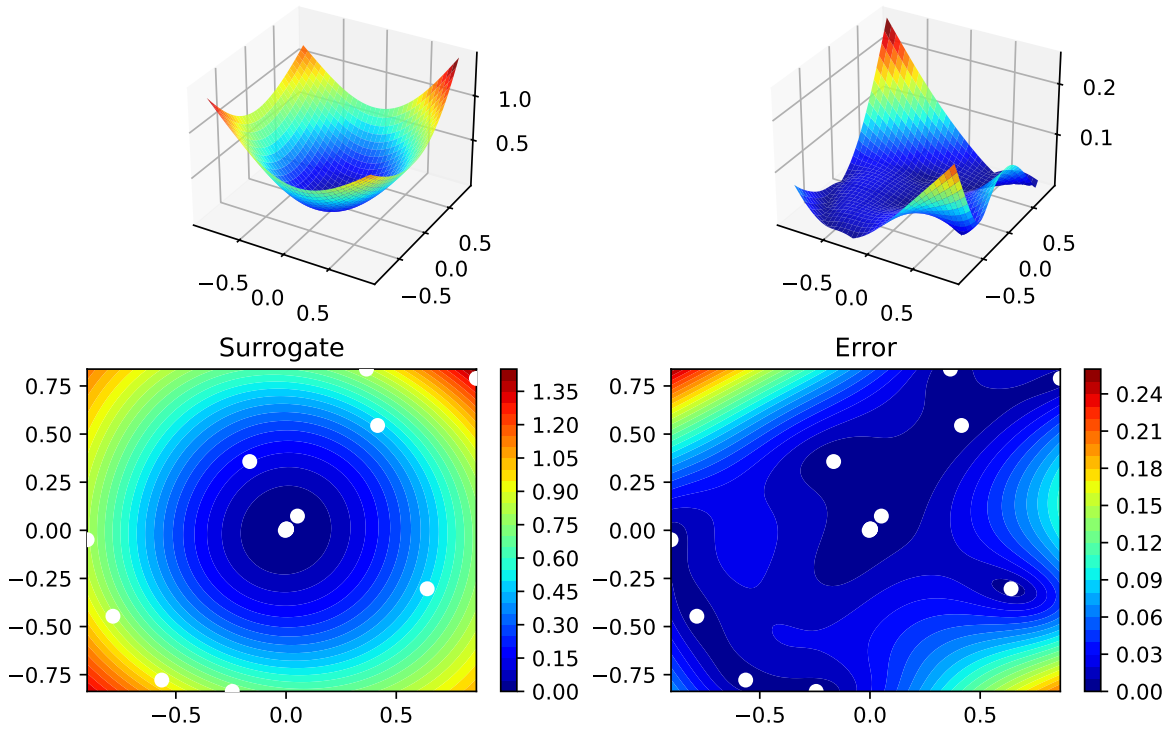
```
spot_2_anisotropic.plot_progress(log_y=True)
```




```
spot_2_anisotropic.print_results()
```

```
min y: 1.2267580433005057e-05
x0: -0.003457387366511919
x1: -0.000560404345886819

[['x0', -0.003457387366511919], ['x1', -0.000560404345886819]]
```

```
spot_2_anisotropic.surrogate.plot()
```



9.2.1 Taking a Look at the `theta` Values

9.2.1.1 `theta` Values from the `spot` Model

- We can check, whether one or several `theta` values were used.
- The `theta` values from the surrogate can be printed as follows:

```
spot_2_anisotropic.surrogate.theta
```

```
array([-0.30709835, -0.13263039])
```

- Since the surrogate from the isotropic setting was stored as `spot_2`, we can also take a look at the `theta` value from this model:

```
spot_2.surrogate.theta
```

```
array([-0.15830282])
```

9.2.1.2 TensorBoard

Now we can start TensorBoard in the background with the following command:

```
tensorboard --logdir="./runs"
```

We can access the TensorBoard web server with the following URL:

```
http://localhost:6006/
```

The TensorBoard plot illustrates how `spotPython` can be used as a microscope for the internal mechanisms of the surrogate-based optimization process. Here, one important parameter, the learning rate θ of the Kriging surrogate is plotted against the number of optimization steps.

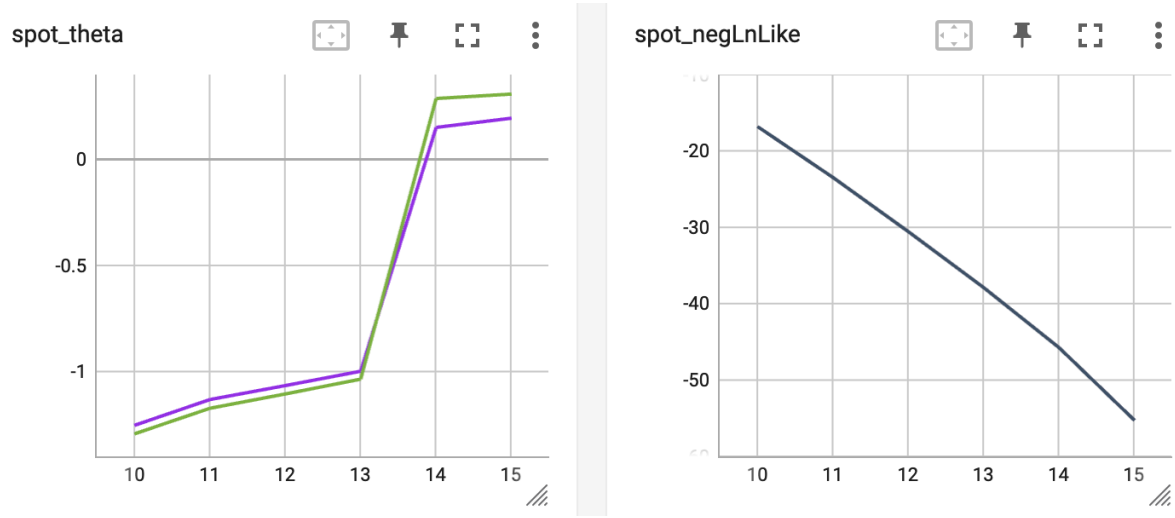


Figure 9.1: TensorBoard visualization of the spotPython surrogate model.

9.3 Exercises

9.3.1 `fun_branin`

- Describe the function.
  - The input dimension is 2. The search range is $-5 \leq x_1 \leq 10$ and $0 \leq x_2 \leq 15$.
- Compare the results from `spotPython` run a) with isotropic and b) anisotropic surrogate models.

- Modify the termination criterion: instead of the number of evaluations (which is specified via `fun_evals`), the time should be used as the termination criterion. This can be done as follows (`max_time=1` specifies a run time of one minute):

```
fun_evals=inf,
max_time=1,
```

9.3.2 `fun_sin_cos`

- Describe the function.
  - The input dimension is 2. The search range is $-2\pi \leq x_1 \leq 2\pi$ and $-2\pi \leq x_2 \leq 2\pi$.
- Compare the results from `spotPython` run a) with isotropic and b) anisotropic surrogate models.
- Modify the termination criterion (`max_time` instead of `fun_evals`) as described for `fun_branin`.

9.3.3 `fun_runge`

- Describe the function.
  - The input dimension is 2. The search range is $-5 \leq x_1 \leq 5$ and $-5 \leq x_2 \leq 5$.
- Compare the results from `spotPython` run a) with isotropic and b) anisotropic surrogate models.
- Modify the termination criterion (`max_time` instead of `fun_evals`) as described for `fun_branin`.

9.3.4 `fun_wingwt`

- Describe the function.
  - The input dimension is 10. The search ranges are between 0 and 1 (values are mapped internally to their natural bounds).
- Compare the results from `spotPython` run a) with isotropic and b) anisotropic surrogate models.
- Modify the termination criterion (`max_time` instead of `fun_evals`) as described for `fun_branin`.

10 Using sklearn Surrogates in spotPython

Besides the internal kriging surrogate, which is used as a default by spotPython, any surrogate model from scikit-learn can be used as a surrogate in spotPython. This chapter explains how to use scikit-learn surrogates in spotPython.

```
import numpy as np
from math import inf
from spotPython.fun.objectivefunctions import analytical
from spotPython.spot import spot
```

10.1 Example: Branin Function with spotPython's Internal Kriging Surrogate

10.1.1 The Objective Function Branin

- The spotPython package provides several classes of objective functions.
- We will use an analytical objective function, i.e., a function that can be described by a (closed) formula.
- Here we will use the Branin function:

  ```
  y = a * (x2 - b * x1**2 + c * x1 - r) ** 2 + s * (1 - t) * np.cos(x1) + s,
  where values of a, b, c, r, s and t are: a = 1, b = 5.1 / (4*pi**2),
  c = 5 / pi, r = 6, s = 10 and t = 1 / (8*pi).
  ```

- It has three global minima:

  ```
  f(x) = 0.397887 at (-pi, 12.275), (pi, 2.275), and (9.42478, 2.475).
  ```

```
from spotPython.fun.objectivefunctions import analytical
lower = np.array([-5,-0])
upper = np.array([10,15])
```

```
fun = analytical().fun_branin
```

TensorBoard

Similar to the one-dimensional case, which was introduced in Section Section 7.5, we can use TensorBoard to monitor the progress of the optimization. We will use the same code, only the prefix is different:

```
from spotPython.utils.file import get_experiment_name
from spotPython.utils.init import fun_control_init
from spotPython.utils.file import get_spot_tensorboard_path

PREFIX = "04"
experiment_name = get_experiment_name(prefix=PREFIX)
print(experiment_name)

fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name))
```

```
04_p040025_2023-12-30_22-57-50
```

10.1.2 Running the surrogate model based optimizer `Spot`:

```
spot_2 = spot.Spot(fun=fun,
                   lower = lower,
                   upper = upper,
                   fun_evals = 20,
                   max_time = inf,
                   seed=123,
                   design_control={"init_size": 10},
                   fun_control=fun_control)
```

```
spot_2.run()
```

```
spotPython tuning: 3.1468324340130396 [######----] 55.00%

spotPython tuning: 3.1468324340130396 [######----] 60.00%

spotPython tuning: 3.1468324340130396 [######----] 65.00%
```

```
spotPython tuning: 3.1468324340130396 [#######---] 70.00%

spotPython tuning: 1.1487178651597567 [########--] 75.00%

spotPython tuning: 1.023873960670663 [########--] 80.00%

spotPython tuning: 0.4157039136851921 [########--] 85.00%

spotPython tuning: 0.40177808730611275 [#########-] 90.00%

spotPython tuning: 0.39909168407470474 [##########] 95.00%

spotPython tuning: 0.39909168407470474 [##########] 100.00% Done...

<spotPython.spot.spot.Spot at 0x2d51b9a90>
```

10.1.3 TensorBoard

Now we can start TensorBoard in the background with the following command:

```
tensorboard --logdir="./runs"
```

We can access the TensorBoard web server with the following URL:

```
http://localhost:6006/
```

The TensorBoard plot illustrates how `spotPython` can be used as a microscope for the internal mechanisms of the surrogate-based optimization process. Here, one important parameter, the learning rate θ of the Kriging surrogate is plotted against the number of optimization steps.

10.1.4 Print the Results

```
spot_2.print_results()
```

```
min y: 0.39909168407470474
x0: 3.1523096108334228
x1: 2.292209322210569

[['x0', 3.1523096108334228], ['x1', 2.292209322210569]]
```

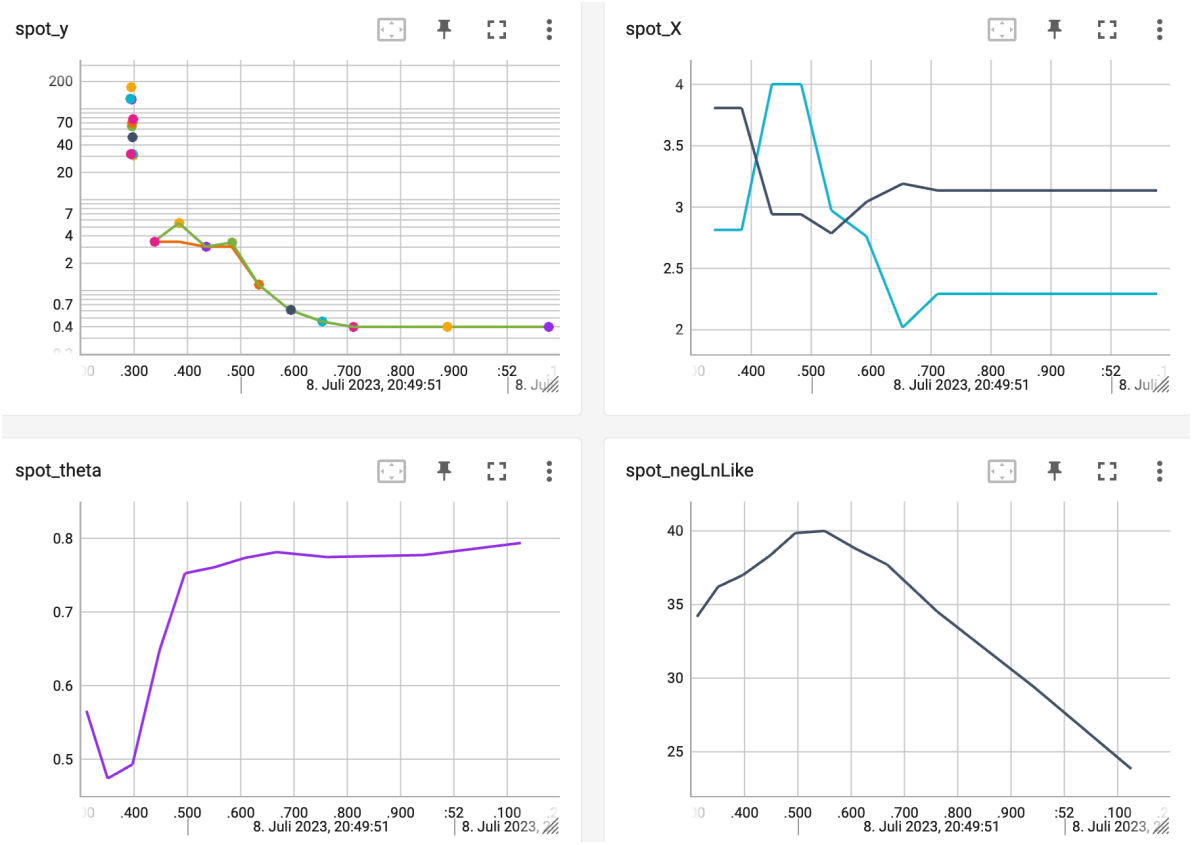
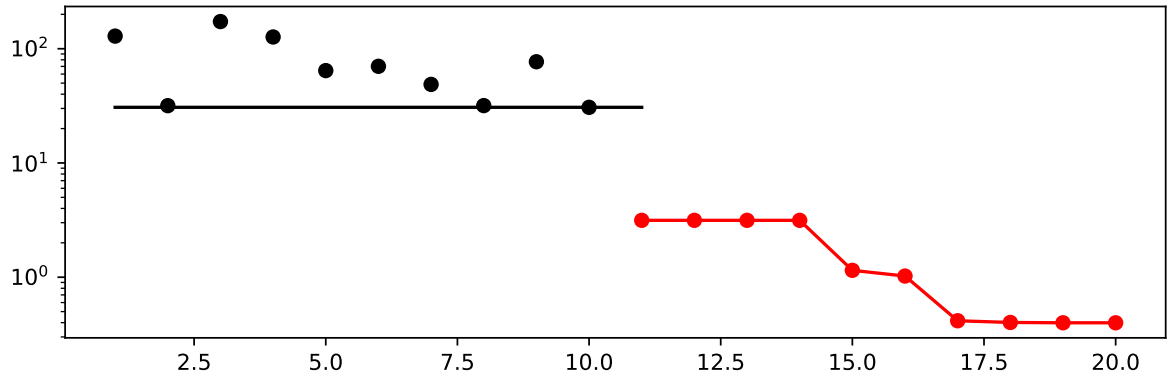


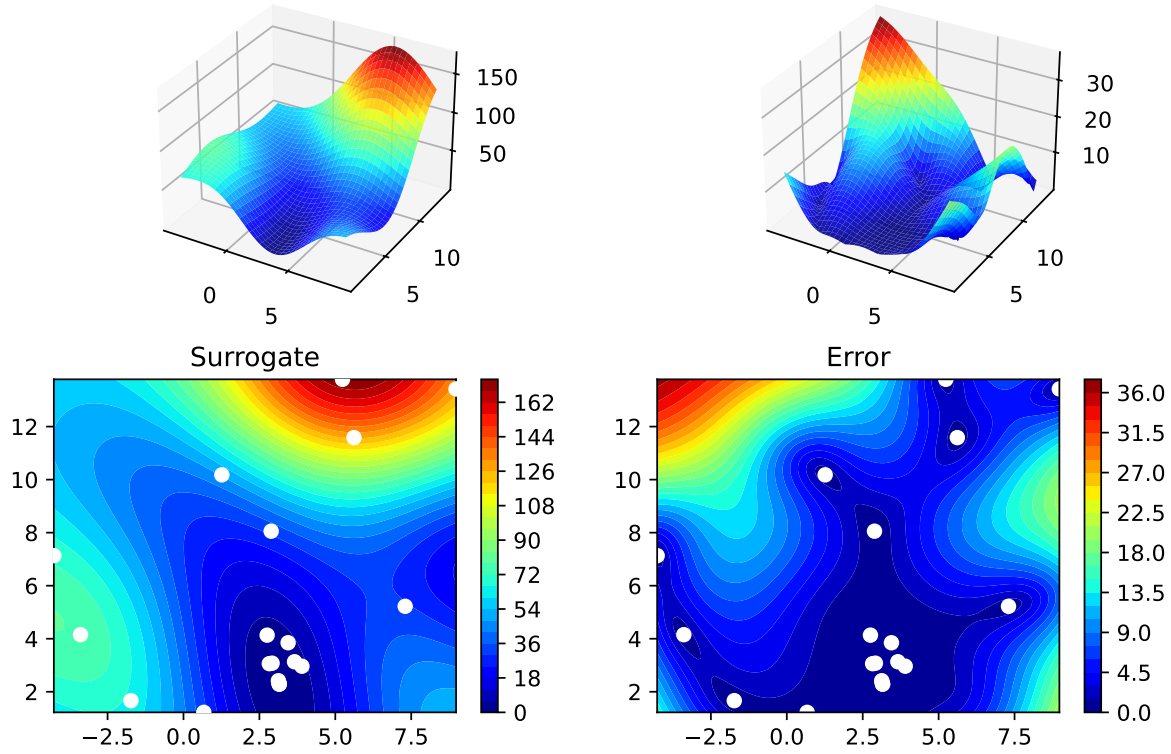
Figure 10.1: TensorBoard visualization of the spotPython optimization process and the surrogate model.

10.1.5 Show the Progress and the Surrogate

```
spot_2.plot_progress(log_y=True)
```




```
spot_2.surrogate.plot()
```



10.2 Example: Using Surrogates From scikit-learn

- Default is the `spotPython` (i.e., the internal) `kriging` surrogate.
- It can be called explicitely and passed to `Spot`.

```
from spotPython.build.kriging import Kriging
S_0 = Kriging(name='kriging', seed=123)
```

- Alternatively, models from `scikit-learn` can be selected, e.g., Gaussian Process, RBFs, Regression Trees, etc.

```
# Needed for the sklearn surrogates:
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn import linear_model
from sklearn import tree
import pandas as pd
```

- Here are some additional models that might be useful later:

```
S_Tree = DecisionTreeRegressor(random_state=0)
S_LM = linear_model.LinearRegression()
S_Ridge = linear_model.Ridge()
S_RF = RandomForestRegressor(max_depth=2, random_state=0)
```

10.2.1 GaussianProcessRegressor as a Surrogate

- To use a Gaussian Process model from `sklearn`, that is similar to `spotPython`'s `Kriging`, we can proceed as follows:

```
kernel = 1 * RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e2))
S_GP = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=9)
```

- The scikit-learn GP model `S_GP` is selected for `Spot` as follows:

  `surrogate = S_GP`

- We can check the kind of surogate model with the command `isinstance`:

```
isinstance(S_GP, GaussianProcessRegressor)
```

```
True
```

```
isinstance(S_0, Kriging)
```

```
True
```

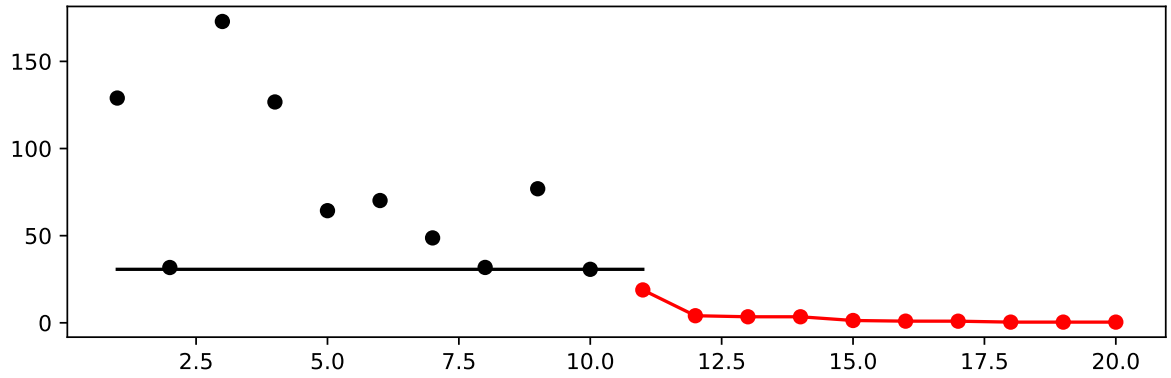
- Similar to the `Spot` run with the internal `Kriging` model, we can call the run with the `scikit-learn` surrogate:

```
fun = analytical(seed=123).fun_branin
spot_2_GP = spot.Spot(fun=fun,
                    lower = lower,
                    upper = upper,
                    fun_evals = 20,
                    seed=123,
                    design_control={"init_size": 10},
                    surrogate = S_GP)
spot_2_GP.run()
```

```
spotPython tuning: 18.865102092040814 [######----] 55.00%

spotPython tuning: 4.067063943633956 [######----] 60.00%

spotPython tuning: 3.461921636551292 [######----] 65.00%

spotPython tuning: 3.461921636551292 [#######---] 70.00%

spotPython tuning: 1.3283243814960493 [########--] 75.00%

spotPython tuning: 0.9549503108510411 [########--] 80.00%

spotPython tuning: 0.9352250003678275 [########--] 85.00%

spotPython tuning: 0.39960822331589974 [#########-] 90.00%
```

```
spotPython tuning: 0.3981631812933486 [##########] 95.00%

spotPython tuning: 0.3981631812933486 [##########] 100.00% Done...

<spotPython.spot.spot.Spot at 0x2dce8d990>
```

```
spot_2_GP.plot_progress()
```




```
spot_2_GP.print_results()
```

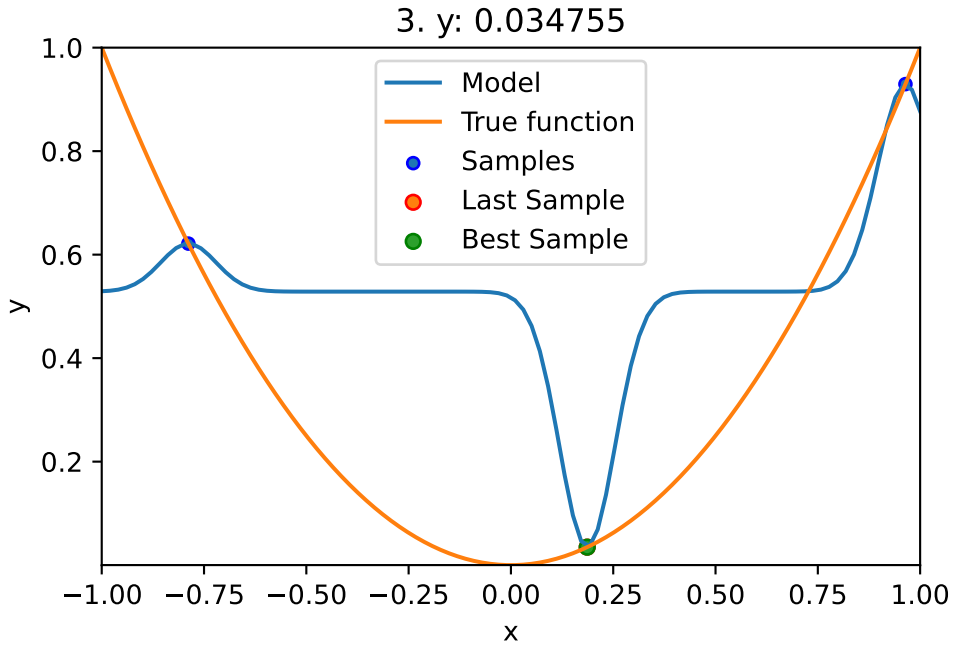
```
min y: 0.3981631812933486
x0: 3.1491652274330053
x1: 2.2698186003445153

[['x0', 3.1491652274330053], ['x1', 2.2698186003445153]]
```

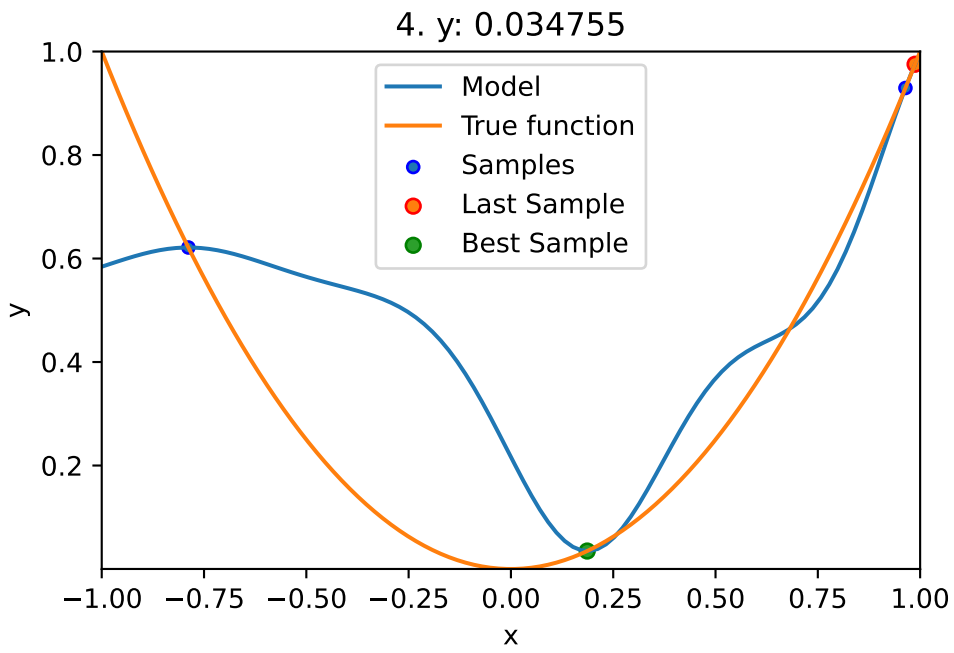
10.3 Example: One-dimensional Sphere Function With spotPython's Kriging

- In this example, we will use an one-dimensional function, which allows us to visualize the optimization process.
  - `show_models= True` is added to the argument list.

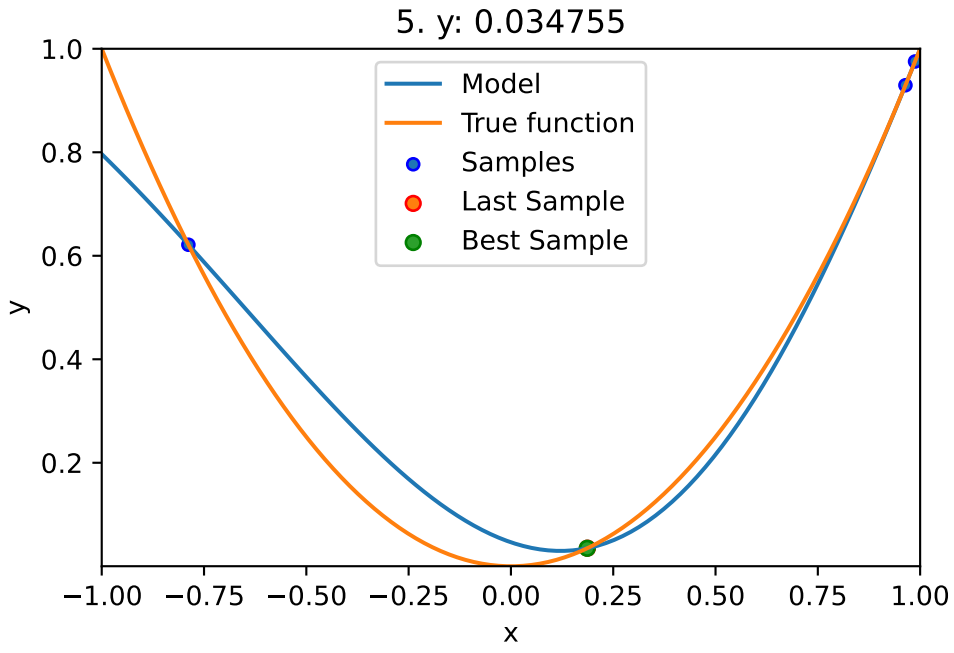
```
from spotPython.fun.objectivefunctions import analytical
lower = np.array([-1])
upper = np.array([1])
fun = analytical(seed=123).fun_sphere
```

```
spot_1 = spot.Spot(fun=fun,
                   lower = lower,
                   upper = upper,
                   fun_evals = 10,
                   max_time = inf,
                   seed=123,
                   show_models= True,
                   tolerance_x = np.sqrt(np.spacing(1)),
                   design_control={"init_size": 3},)
spot_1.run()
```

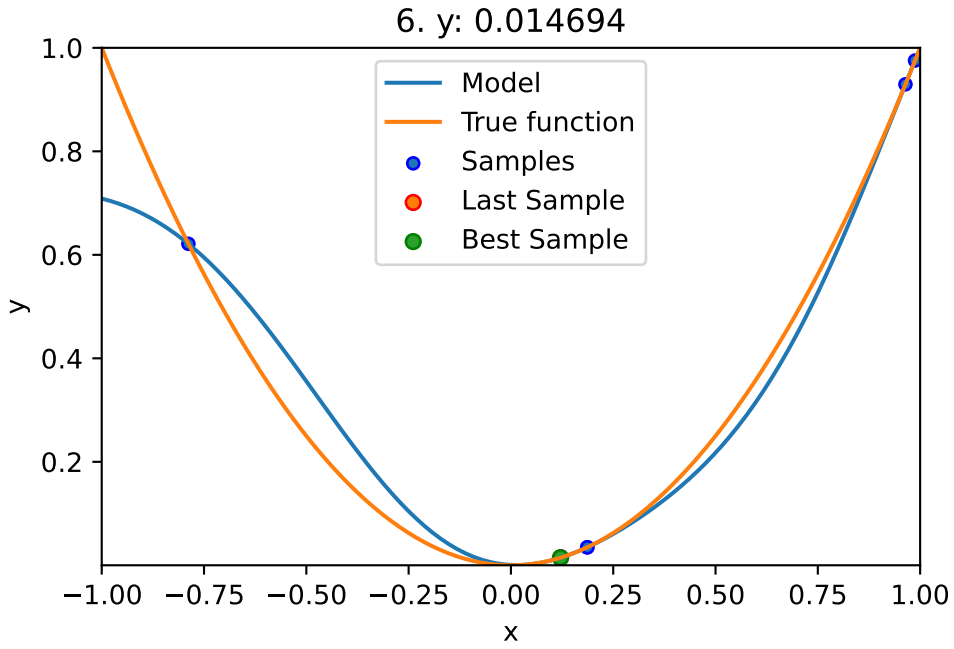




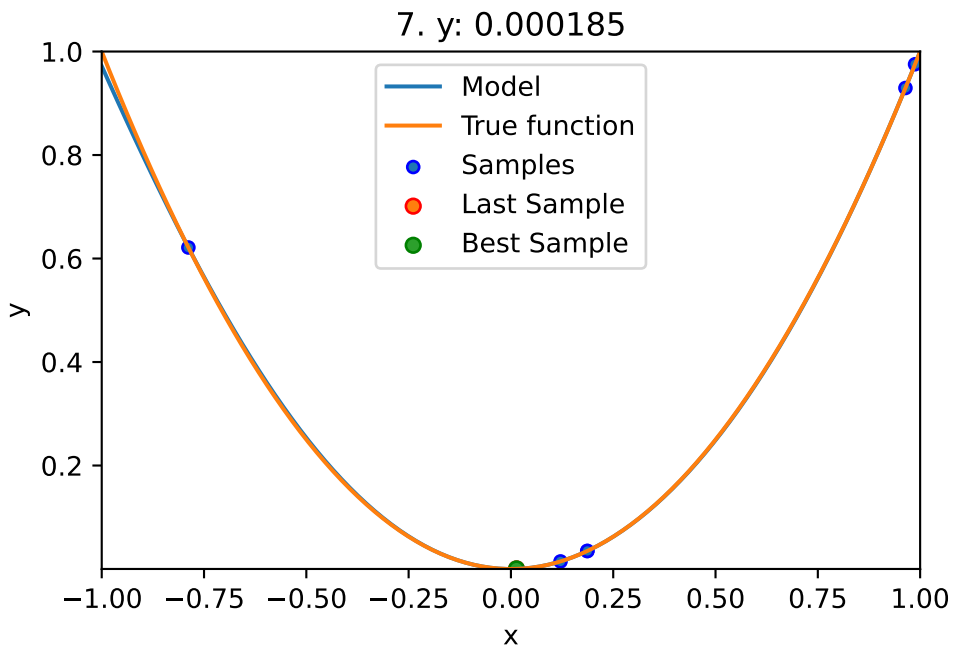

```
spotPython tuning: 0.03475493366922229 [####------] 40.00%
```



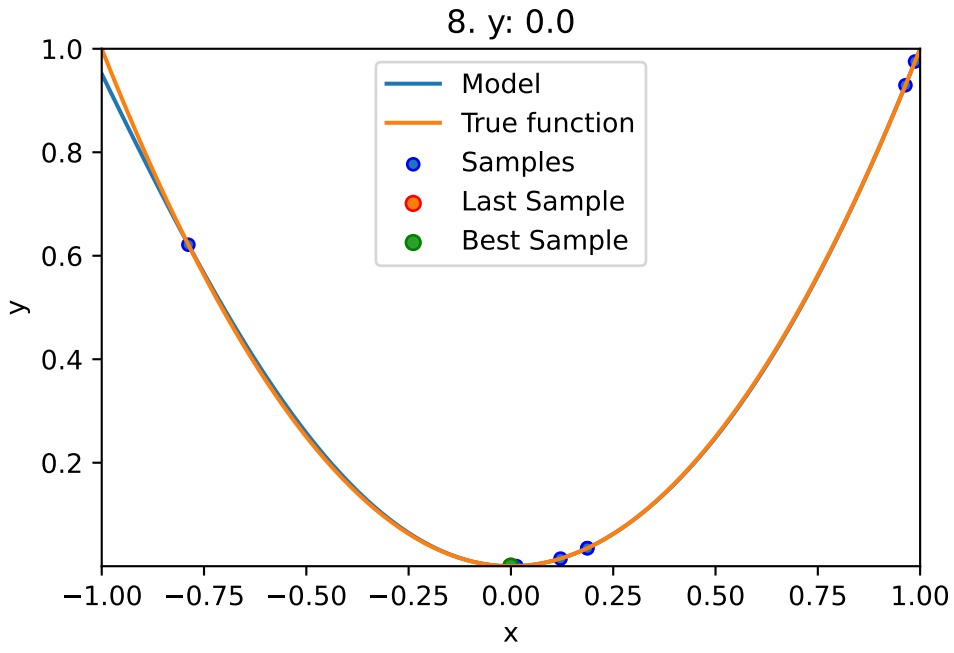
```
spotPython tuning: 0.03475493366922229 [#####-----] 50.00%
```



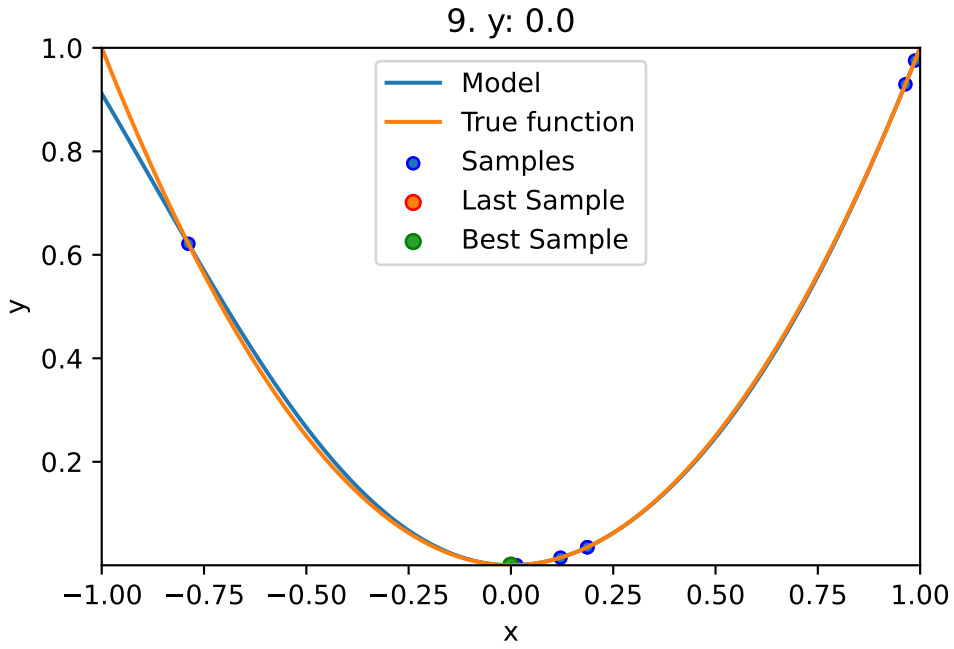

```
spotPython tuning: 0.014693634061986652 [######----] 60.00%
```



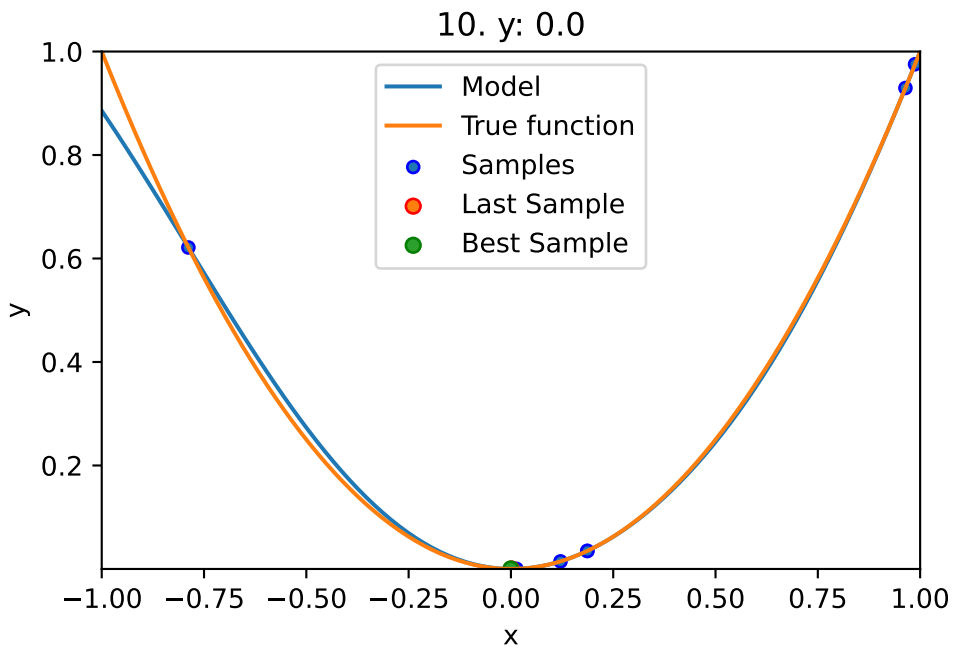

```
spotPython tuning: 0.0001850752755504842 [#######---] 70.00%
```



```
spotPython tuning: 4.643556417950029e-08 [########--] 80.00%
```




```
spotPython tuning: 4.643556417950029e-08 [#########-] 90.00%
```




```
spotPython tuning: 4.643556417950029e-08 [##########] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2d6c3d210>
```

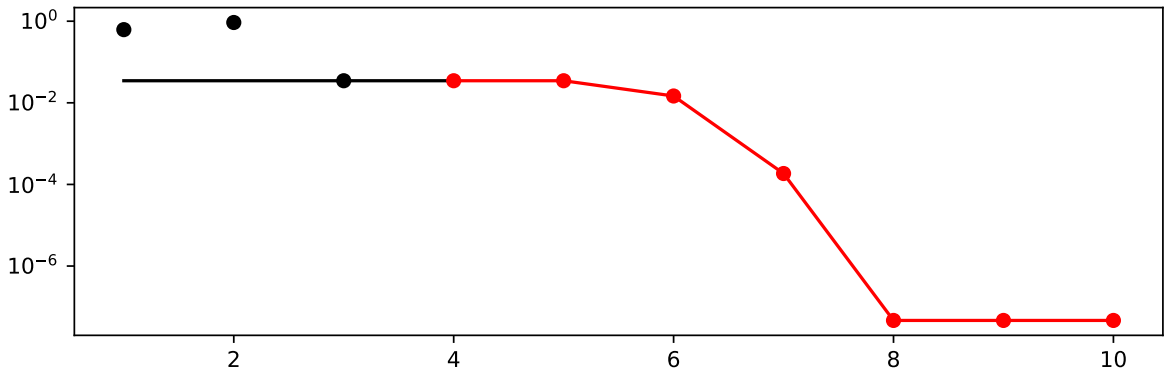
10.3.1 Results

```
spot_1.print_results()
```

```
min y: 4.643556417950029e-08
x0: -0.0002154891277524235
```

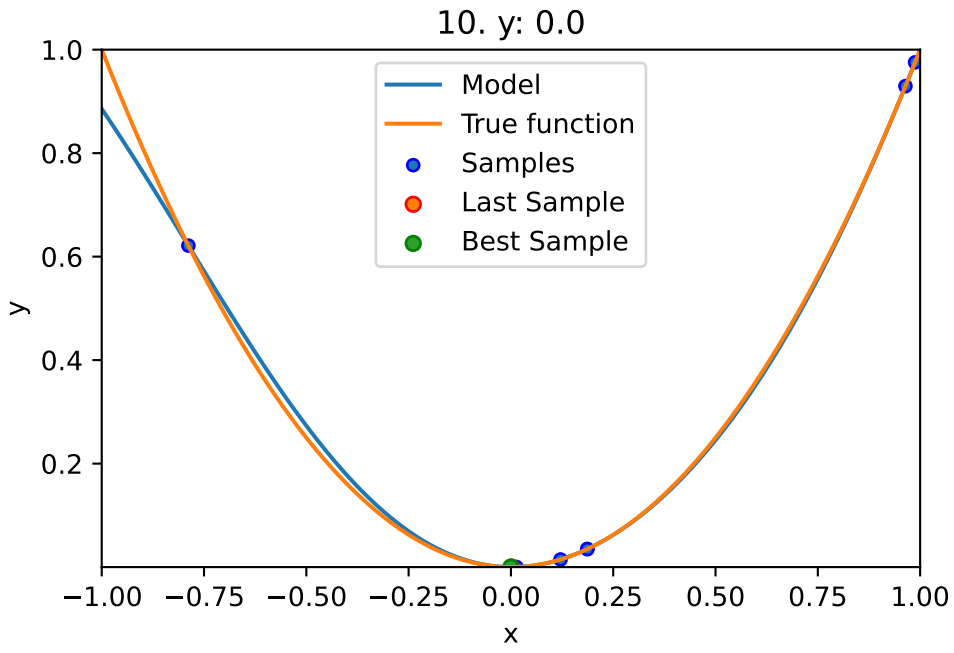
```
[['x0', -0.0002154891277524235]]
```

```
spot_1.plot_progress(log_y=True)
```



- The method `plot_model` plots the final surrogate:

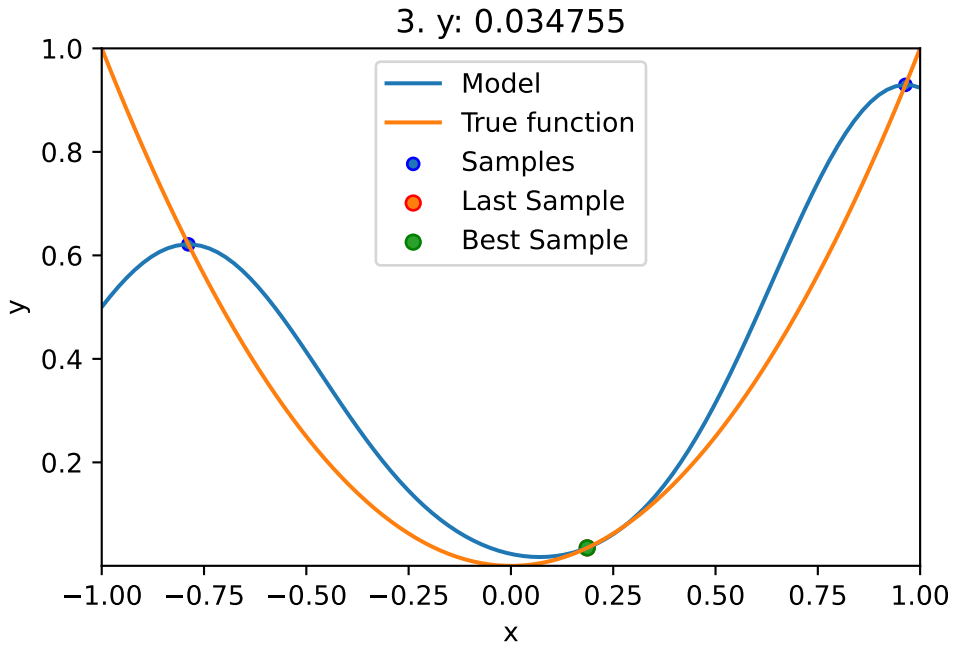
```
spot_1.plot_model()
```

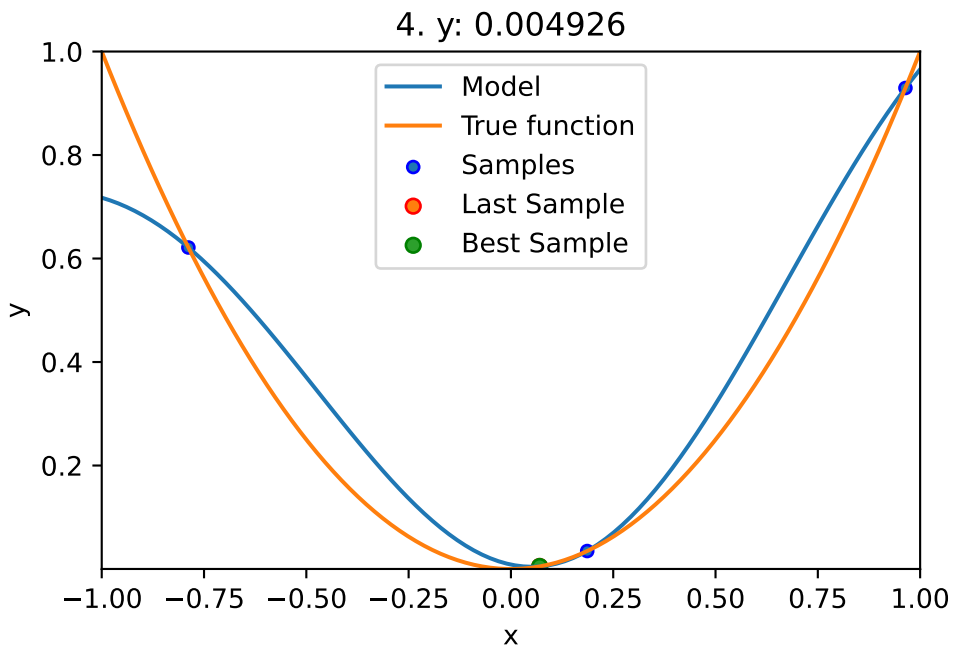


10.4 Example: `Sklearn` Model GaussianProcess

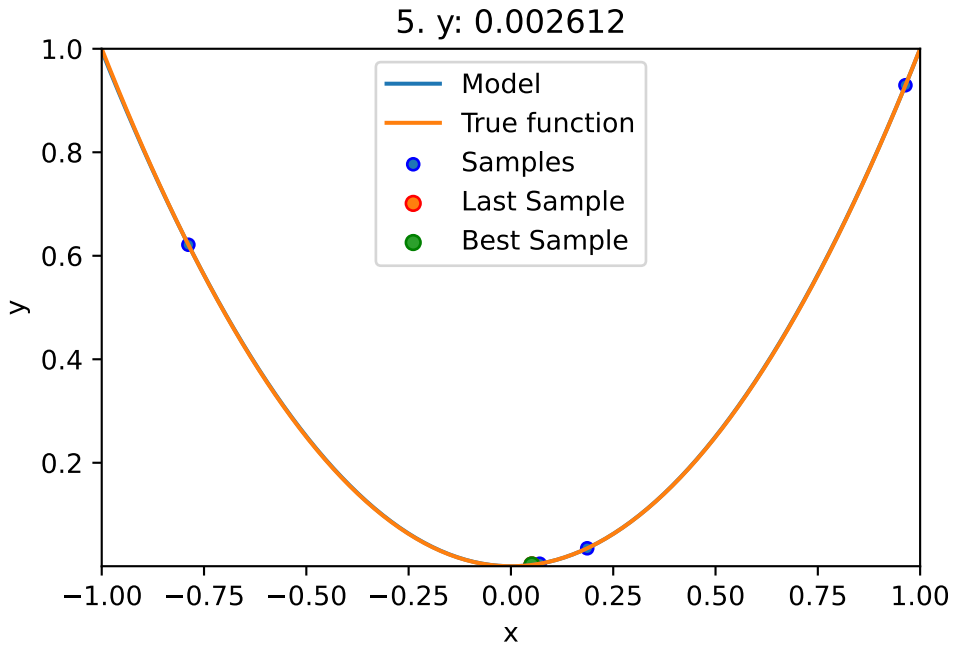
- This example visualizes the search process on the `GaussianProcessRegression` surrogate from `sklearn`.
- Therefore `surrogate = S_GP` is added to the argument list.

```
fun = analytical(seed=123).fun_sphere
spot_1_GP = spot.Spot(fun=fun,
                      lower = lower,
                      upper = upper,
                      fun_evals = 10,
                      max_time = inf,
                      seed=123,
                      show_models= True,
                      design_control={"init_size": 3},
                      surrogate = S_GP)
spot_1_GP.run()
```

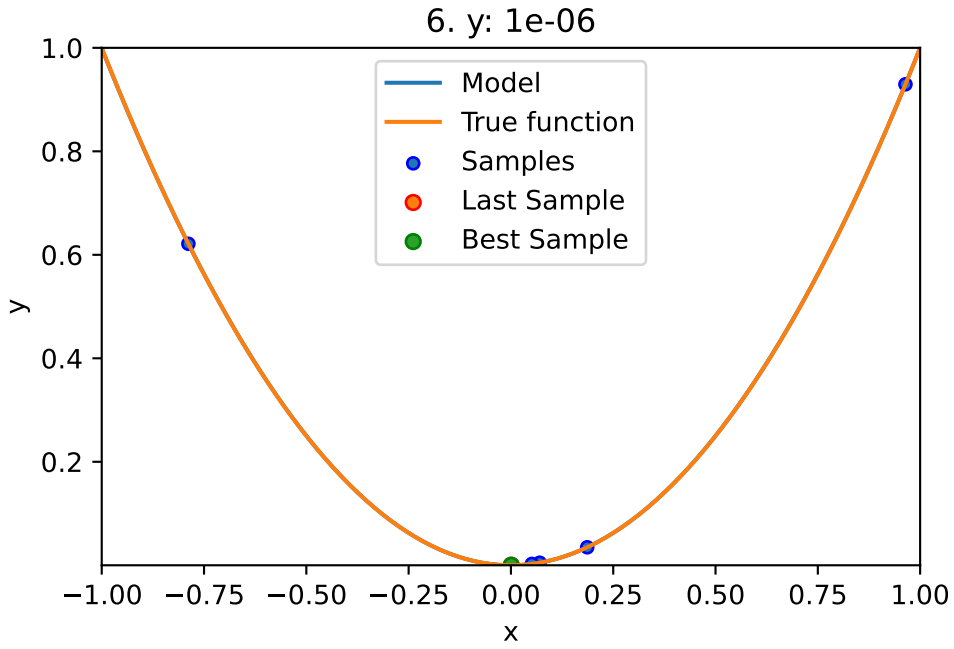




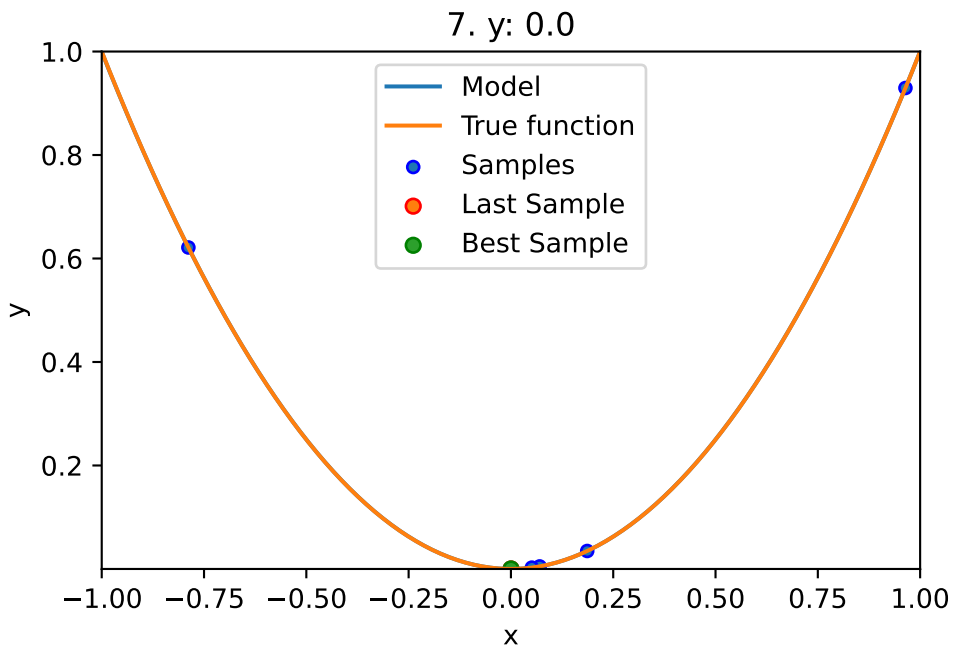

```
spotPython tuning: 0.0049257617153734565 [####------] 40.00%
```



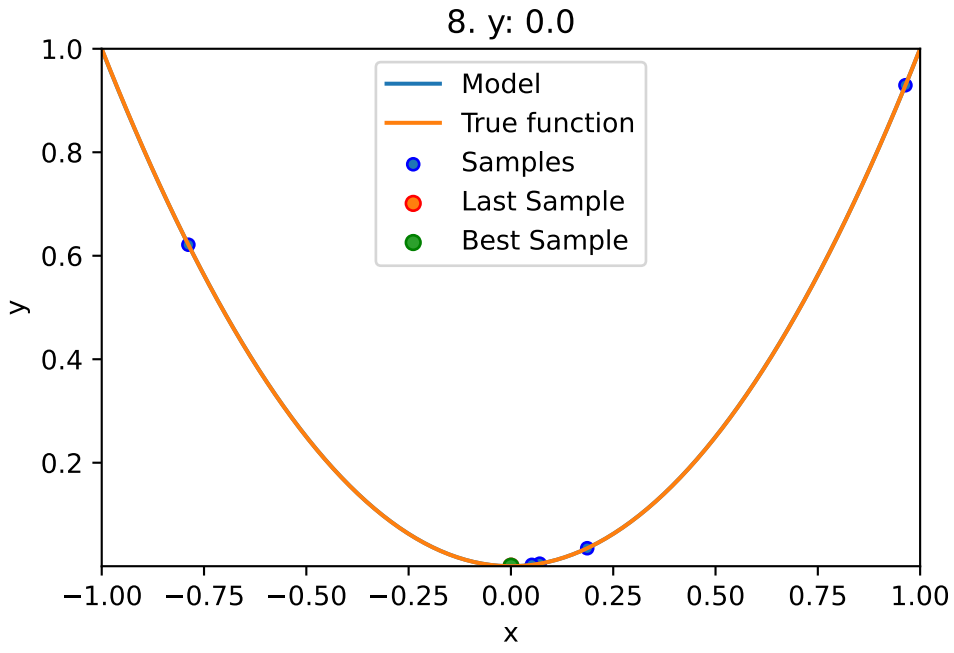
```
spotPython tuning: 0.002612076484523571 [#####-----] 50.00%
```



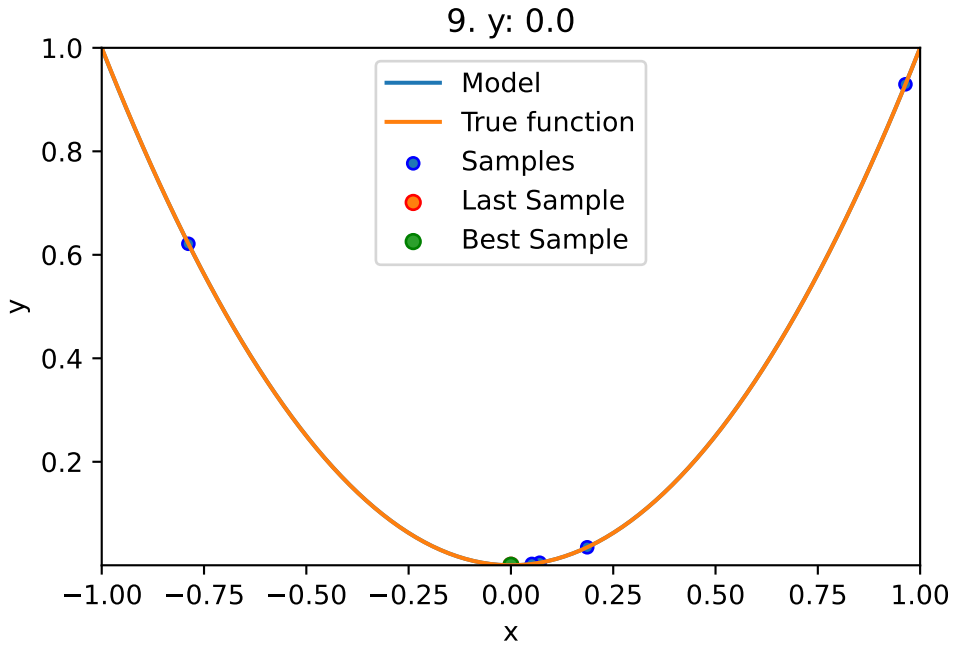

```
spotPython tuning: 5.912935436232656e-07 [######----] 60.00%
```



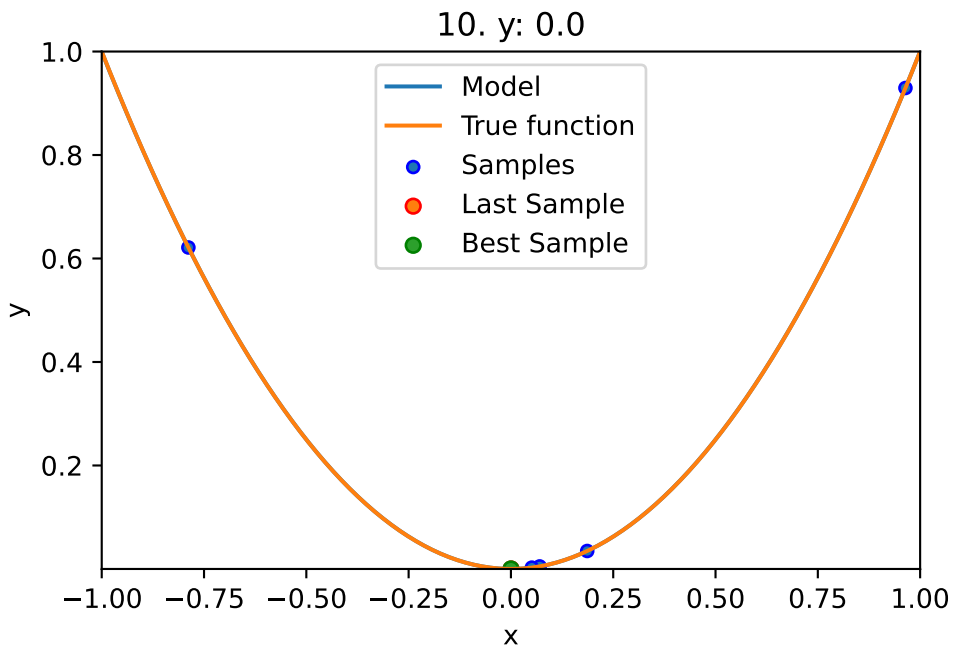

```
spotPython tuning: 4.211659449449057e-08 [#######---] 70.00%
```



```
spotPython tuning: 4.211659449449057e-08 [########--] 80.00%
```




```
spotPython tuning: 1.6119389142446866e-09 [#########-] 90.00%
```




```
spotPython tuning: 1.6119389142446866e-09 [##########] 100.00% Done...
```

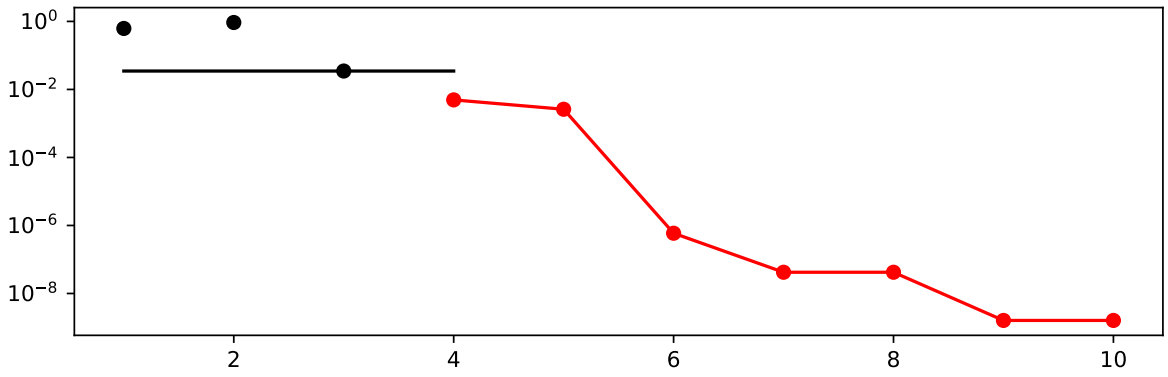
```
<spotPython.spot.spot.Spot at 0x2ddbd20d0>
```

```
spot_1_GP.print_results()
```

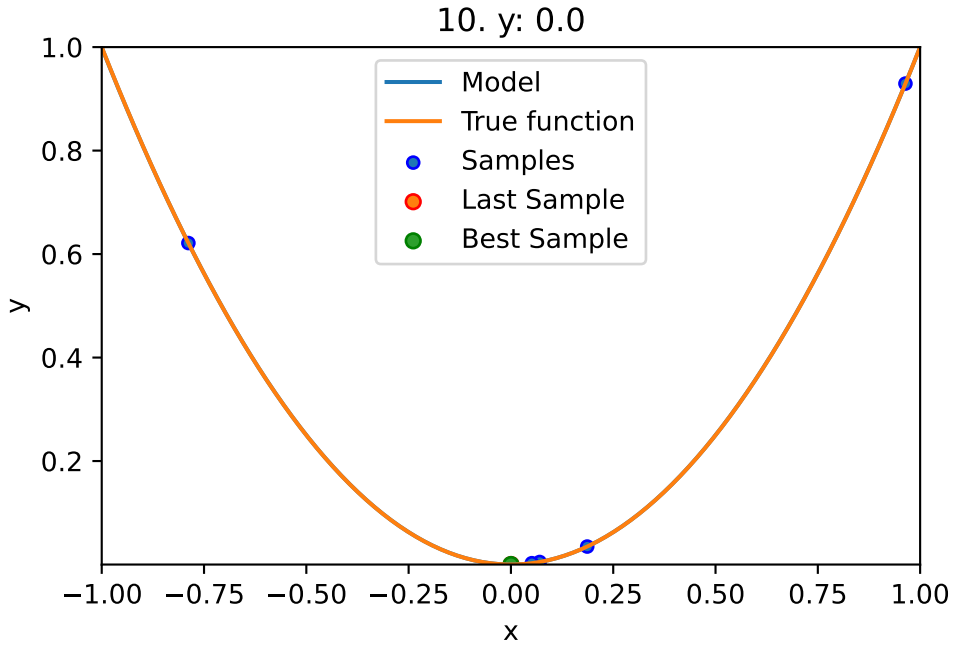
```
min y: 1.6119389142446866e-09
x0: 4.0148959068009304e-05
```

```
[['x0', 4.0148959068009304e-05]]
```

```
spot_1_GP.plot_progress(log_y=True)
```




```
spot_1_GP.plot_model()
```



10.5 Exercises

10.5.1 `DecisionTreeRegressor`

- Describe the surrogate model.
- Use the surrogate as the model for optimization.

10.5.2 `RandomForestRegressor`

- Describe the surrogate model.
- Use the surrogate as the model for optimization.

10.5.3 `linear_model.LinearRegression`

- Describe the surrogate model.
- Use the surrogate as the model for optimization.

10.5.4 `linear_model.Ridge`

- Describe the surrogate model.
- Use the surrogate as the model for optimization.

10.6 Exercise 2

- Compare the performance of the five different surrogates on both objective functions:
  - `spotPython`'s internal Kriging
  - `DecisionTreeRegressor`
  - `RandomForestRegressor`
  - `linear_model.LinearRegression`
  - `linear_model.Ridge`

11 Sequential Parameter Optimization: Gaussian Process Models

This chapter analyzes differences between the `Kriging` implementation in `spotPython` and the `GaussianProcessRegressor` in `scikit-learn`.

```
import numpy as np
from math import inf
from spotPython.fun.objectivefunctions import analytical
from spotPython.design.spacefilling import spacefilling
from spotPython.spot import spot
from spotPython.build.kriging import Kriging
from scipy.optimize import shgo
from scipy.optimize import direct
from scipy.optimize import differential_evolution
import matplotlib.pyplot as plt
import math as m
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF
```

11.1 Gaussian Processes Regression: Basic Introductory `scikit-learn` Example

- This is the example from scikit-learn: https://scikit-learn.org/stable/auto_examples/gaussian_process/pl
- After fitting our model, we see that the hyperparameters of the kernel have been optimized.
- Now, we will use our kernel to compute the mean prediction of the full dataset and plot the 95% confidence interval.

11.1.1 Train and Test Data

```
X = np.linspace(start=0, stop=10, num=1_000).reshape(-1, 1)
y = np.squeeze(X * np.sin(X))
rng = np.random.RandomState(1)
training_indices = rng.choice(np.arange(y.size), size=6, replace=False)
X_train, y_train = X[training_indices], y[training_indices]
```

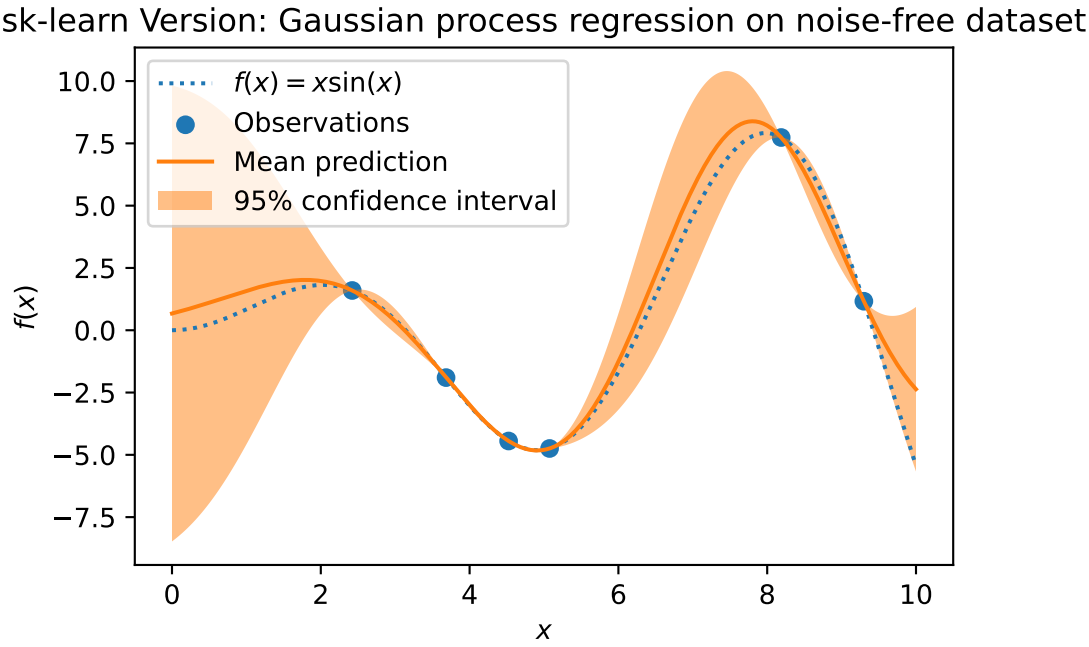
11.1.2 Building the Surrogate With `Sklearn`

- The model building with `sklearn` consisits of three steps:
    1. Instantiating the model, then
    2. fitting the model (using `fit`), and
    3. making predictions (using `predict`)

```
kernel = 1 * RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e2))
gaussian_process = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=9)
gaussian_process.fit(X_train, y_train)
mean_prediction, std_prediction = gaussian_process.predict(X, return_std=True)
```

11.1.3 Plotting the `Sklearn`Model

```
plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, mean_prediction, label="Mean prediction")
plt.fill_between(
    X.ravel(),
    mean_prediction - 1.96 * std_prediction,
    mean_prediction + 1.96 * std_prediction,
    alpha=0.5,
    label=r"95% confidence interval",
)
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("sk-learn Version: Gaussian process regression on noise-free dataset")
```



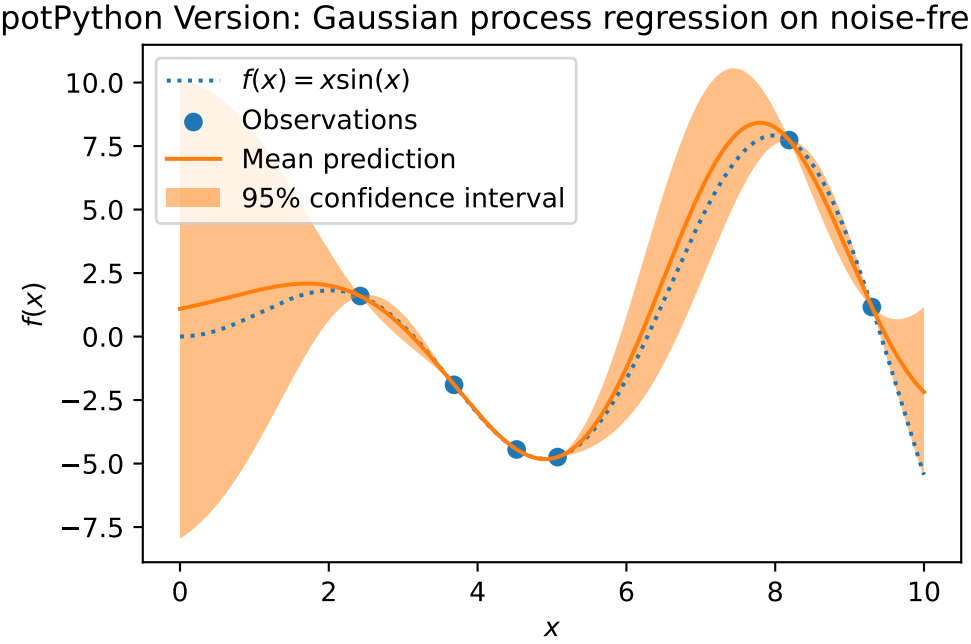
11.1.4 The `spotPython` Version

- The `spotPython` version is very similar:
  1. Instantiating the model, then
  2. fitting the model and
  3. making predictions (using `predict`).

```
S = Kriging(name='kriging',  seed=123, log_level=50, cod_type="norm")
S.fit(X_train, y_train)
S_mean_prediction, S_std_prediction, S_ei = S.predict(X, return_val="all")
```

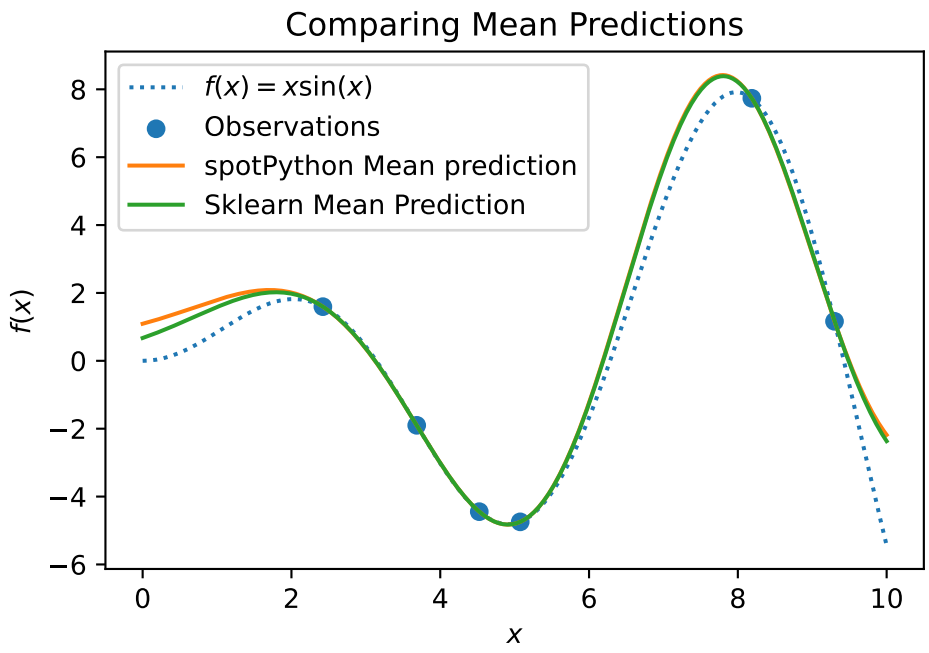
```
plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, S_mean_prediction, label="Mean prediction")
plt.fill_between(
    X.ravel(),
    S_mean_prediction - 1.96 * S_std_prediction,
    S_mean_prediction + 1.96 * S_std_prediction,
    alpha=0.5,
    label=r"95% confidence interval",
```

```
)
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("spotPython Version: Gaussian process regression on noise-free dataset")
```



11.1.5 Visualizing the Differences Between the `spotPython` and the `sklearn` Model Fits

```
plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, S_mean_prediction, label="spotPython Mean prediction")
plt.plot(X, mean_prediction, label="Sklearn Mean Prediction")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Comparing Mean Predictions")
```



11.2 Exercises

11.2.1 Schonlau Example Function

- The Schonlau Example Function is based on sample points only (there is no analytical function description available):

```
X = np.linspace(start=0, stop=13, num=1_000).reshape(-1, 1)
X_train = np.array([1., 2., 3., 4., 12.]).reshape(-1,1)
y_train = np.array([0., -1.75, -2, -0.5, 5.])
```

- Describe the function.
- Compare the two models that were build using the `spotPython` and the `sklearn` surrogate.
- Note: Since there is no analytical function available, you might be interested in adding some points and describe the effects.

11.2.2 Forrester Example Function

- The Forrester Example Function is defined as follows:

f(x) = (6x- 2)^2 sin(12x-4) for x in [0,1].

- Data points are generated as follows:

```
X = np.linspace(start=-0.5, stop=1.5, num=1_000).reshape(-1, 1)
X_train = np.array([0.0, 0.175, 0.225, 0.3, 0.35, 0.375, 0.5,1]).reshape(-1,1)
fun = analytical().fun_forrester
fun_control = {"sigma": 0.1,
               "seed": 123}
y = fun(X, fun_control=fun_control)
y_train = fun(X_train, fun_control=fun_control)
```

- Describe the function.
- Compare the two models that were build using the `spotPython` and the `sklearn` surrogate.
- Note: Modify the noise level (`"sigma"`), e.g., use a value of `0.2`, and compare the two models.

```
fun_control = {"sigma": 0.2}
```

11.2.3 `fun_runge` Function (1-dim)

- The Runge function is defined as follows:

  f(x) = 1/ (1 + sum(x_i))^2

- Data points are generated as follows:

```
gen = spacefilling(1)
rng = np.random.RandomState(1)
lower = np.array([-10])
upper = np.array([10])
fun = analytical().fun_runge
fun_control = {"sigma": 0.025,
               "seed": 123}
X_train = gen.scipy_lhd(10, lower=lower, upper = upper).reshape(-1,1)
y_train = fun(X, fun_control=fun_control)
X = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
y = fun(X, fun_control=fun_control)
```

- Describe the function.

- Compare the two models that were build using the `spotPython` and the `sklearn` surrogate.
- Note: Modify the noise level (`"sigma"`), e.g., use a value of `0.05`, and compare the two models.

```
fun_control = {"sigma": 0.5}
```

11.2.4 `fun_cubed (1-dim)`

- The Cubed function is defined as follows:

  `np.sum(X[i]** 3)`

- Data points are generated as follows:

```
gen = spacefilling(1)
rng = np.random.RandomState(1)
lower = np.array([-10])
upper = np.array([10])
fun = analytical().fun_cubed
fun_control = {"sigma": 0.025,
               "seed": 123}
X_train = gen.scipy_lhd(10, lower=lower, upper = upper).reshape(-1,1)
y_train = fun(X, fun_control=fun_control)
X = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
y = fun(X, fun_control=fun_control)
```

- Describe the function.
- Compare the two models that were build using the `spotPython` and the `sklearn` surrogate.
- Note: Modify the noise level (`"sigma"`), e.g., use a value of `0.05`, and compare the two models.

```
fun_control = {"sigma": 0.05}
```

11.2.5 The Effect of Noise

How does the behavior of the `spotPython` fit changes when the argument `noise` is set to `True`, i.e.,

```
S = Kriging(name='kriging',  seed=123, n_theta=1, noise=True)
```

is used?

12 Expected Improvement

This chapter describes, analyzes, and compares different infill criterion. An infill criterion defines how the next point x_{n+1} is selected from the surrogate model S. Expected improvement is a popular infill criterion in Bayesian optimization.

12.1 Example: `Spot` and the 1-dim Sphere Function

```
import numpy as np
from math import inf
from spotPython.fun.objectivefunctions import analytical
from spotPython.spot import spot
import matplotlib.pyplot as plt
```

12.1.1 The Objective Function: 1-dim Sphere

- The `spotPython` package provides several classes of objective functions.
- We will use an analytical objective function, i.e., a function that can be described by a (closed) formula:

$$f(x) = x^2$$

```
fun = analytical().fun_sphere
```

```
fun = analytical().fun_sphere
```

- The size of the `lower` bound vector determines the problem dimension.
- Here we will use `np.array([-1])`, i.e., a one-dim function.

> **TensorBoard**
>
> Similar to the one-dimensional case, which was introduced in Section Section 7.5, we can use TensorBoard to monitor the progress of the optimization. We will use the same code, only the prefix is different:

```
from spotPython.utils.file import get_experiment_name
from spotPython.utils.init import fun_control_init
from spotPython.utils.file import get_spot_tensorboard_path

PREFIX = "07_Y"
experiment_name = get_experiment_name(prefix=PREFIX)
print(experiment_name)

fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    sigma=0,
    seed=123,)
```

```
07_Y_p040025_2023-12-30_22-58-44
```

```
spot_1 = spot.Spot(fun=fun,
                   fun_evals = 25,
                   lower = np.array([-1]),
                   upper = np.array([1]),
                   design_control={"init_size": 10},
                   tolerance_x = np.sqrt(np.spacing(1)),
                   fun_control = fun_control,)

spot_1.run()
```

```
spotPython tuning: 1.1986325668379847e-08 [####------] 44.00%

spotPython tuning: 1.1986325668379847e-08 [#####-----] 48.00%

spotPython tuning: 1.1986325668379847e-08 [#####-----] 52.00%

spotPython tuning: 1.1385311249270152e-08 [######----] 56.00%

spotPython tuning: 5.189351944607845e-10 [######----] 60.00%

spotPython tuning: 5.189351944607845e-10 [######----] 64.00%

spotPython tuning: 5.189351944607845e-10 [#######---] 68.00%
```

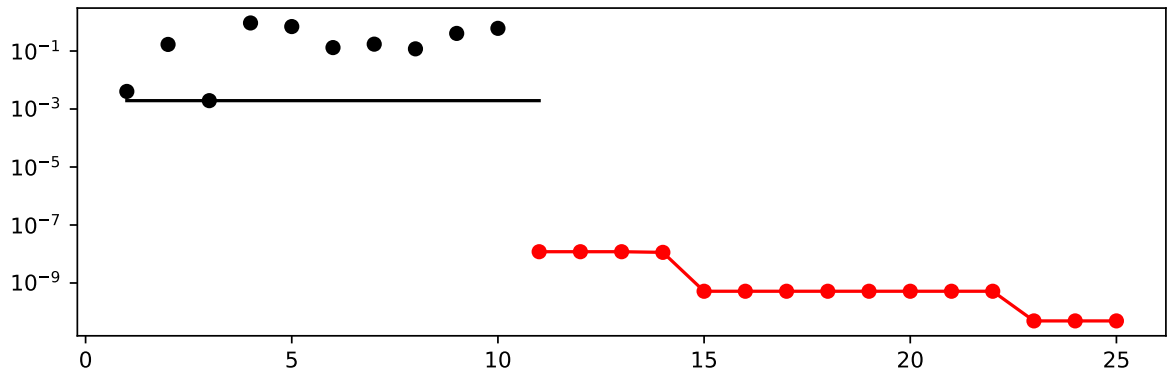
```
spotPython tuning: 5.189351944607845e-10 [#######---] 72.00%

spotPython tuning: 5.189351944607845e-10 [########--] 76.00%

spotPython tuning: 5.189351944607845e-10 [########--] 80.00%

spotPython tuning: 5.189351944607845e-10 [########--] 84.00%

spotPython tuning: 5.189351944607845e-10 [#########-] 88.00%

spotPython tuning: 4.9233547910699585e-11 [#########-] 92.00%

spotPython tuning: 4.9233547910699585e-11 [##########] 96.00%

spotPython tuning: 4.9233547910699585e-11 [##########] 100.00% Done...

<spotPython.spot.spot.Spot at 0x2c7b39210>
```

12.1.2 Results

```
spot_1.print_results()
```

```
min y: 4.9233547910699585e-11
x0: -7.016662163072951e-06

[['x0', -7.016662163072951e-06]]
```

```
spot_1.plot_progress(log_y=True)
```



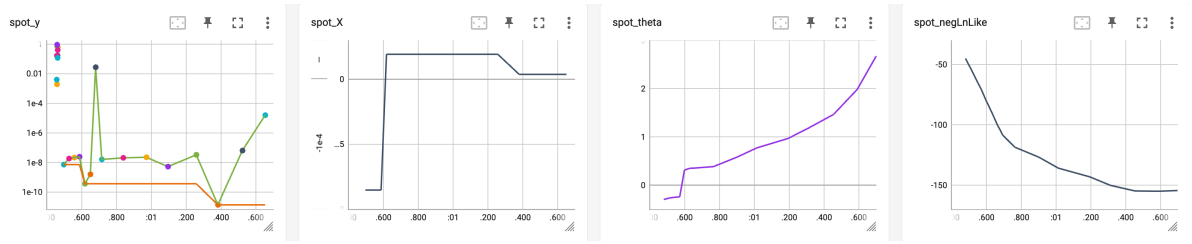


Figure 12.1: TensorBoard visualization of the spotPython optimization process and the surrogate model.

12.2 Same, but with EI as infill_criterion

```
PREFIX = "07_EI_ISO"
experiment_name = get_experiment_name(prefix=PREFIX)
print(experiment_name)
fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    sigma=0,
    seed=123,)
```

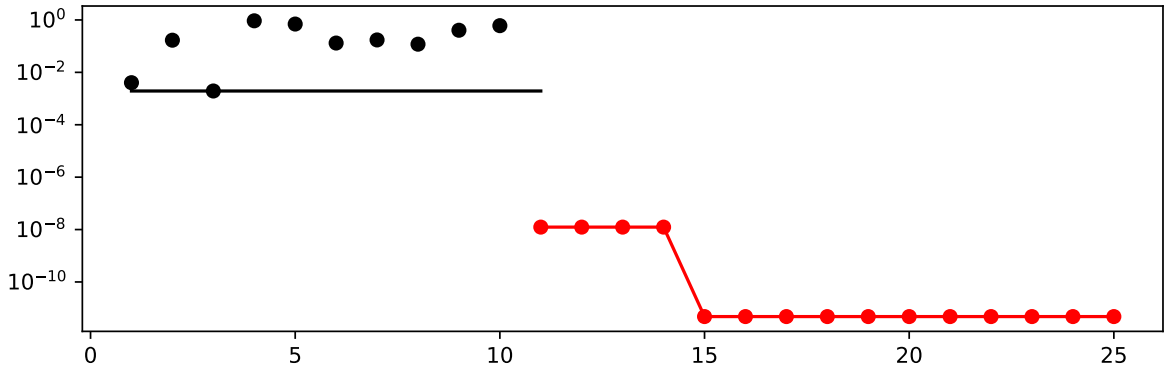
```
07_EI_ISO_p040025_2023-12-30_22-58-47
```

```
spot_1_ei = spot.Spot(fun=fun,
                    lower = np.array([-1]),
                    upper = np.array([1]),
                    fun_evals = 25,
                    tolerance_x = np.sqrt(np.spacing(1)),
                    infill_criterion = "ei",
                    design_control={"init_size": 10},
                    fun_control = fun_control,)
spot_1_ei.run()
```

```
spotPython tuning: 1.2401208585321248e-08 [####------] 44.00%

spotPython tuning: 1.2401208585321248e-08 [#####-----] 48.00%

spotPython tuning: 1.2401208585321248e-08 [#####-----] 52.00%
```

```
spotPython tuning: 1.2401208585321248e-08 [######----] 56.00%

spotPython tuning: 4.773576053400193e-12 [######----] 60.00%

spotPython tuning: 4.773576053400193e-12 [######----] 64.00%

spotPython tuning: 4.773576053400193e-12 [#######---] 68.00%

spotPython tuning: 4.773576053400193e-12 [#######---] 72.00%

spotPython tuning: 4.773576053400193e-12 [########--] 76.00%

spotPython tuning: 4.773576053400193e-12 [########--] 80.00%

spotPython tuning: 4.773576053400193e-12 [########--] 84.00%

spotPython tuning: 4.773576053400193e-12 [#########-] 88.00%

spotPython tuning: 4.773576053400193e-12 [#########-] 92.00%

spotPython tuning: 4.773576053400193e-12 [##########] 96.00%

spotPython tuning: 4.773576053400193e-12 [##########] 100.00% Done...

<spotPython.spot.spot.Spot at 0x2cd5110d0>
```

```
spot_1_ei.plot_progress(log_y=True)
```



```
spot_1_ei.print_results()
```

```
min y: 4.773576053400193e-12
x0: -2.184851494587262e-06

[['x0', -2.184851494587262e-06]]
```

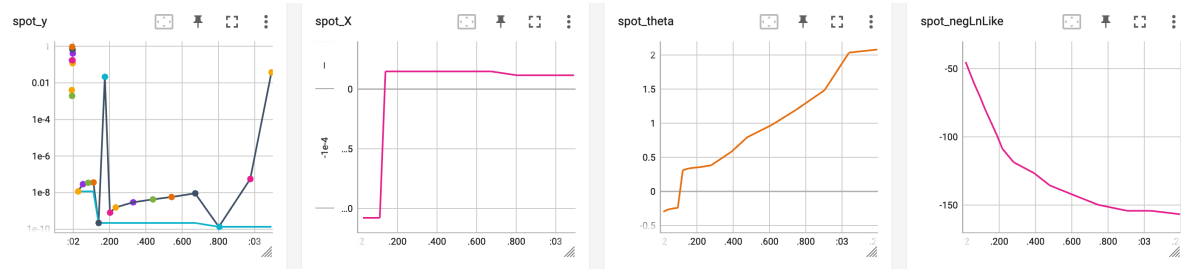


Figure 12.2: TensorBoard visualization of the spotPython optimization process and the surrogate model. Expected improvement, isotropic Kriging.

12.3 Non-isotropic Kriging

```
PREFIX = "07_EI_NONISO"
experiment_name = get_experiment_name(prefix=PREFIX)
print(experiment_name)
fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    sigma=0,
    seed=123,)
```

```
07_EI_NONISO_p040025_2023-12-30_22-58-49
```

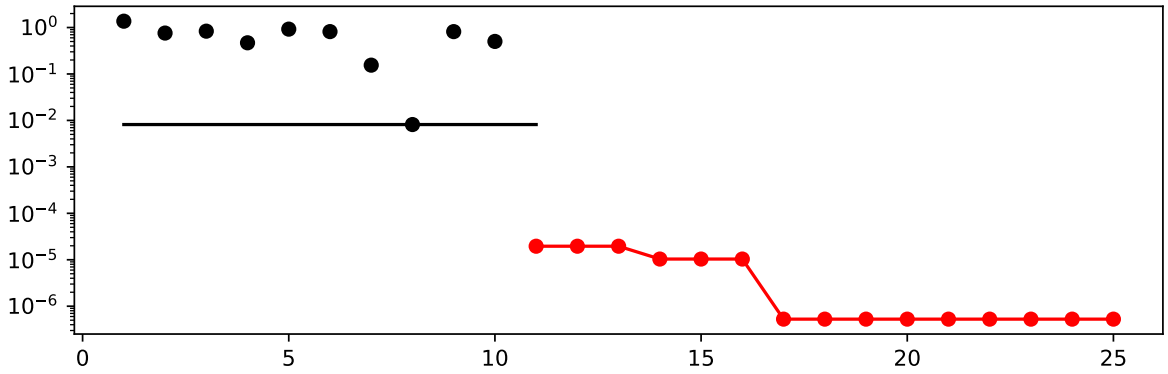
```
spot_2_ei_noniso = spot.Spot(fun=fun,
                   lower = np.array([-1, -1]),
                   upper = np.array([1, 1]),
                   fun_evals = 25,
                   tolerance_x = np.sqrt(np.spacing(1)),
                   infill_criterion = "ei",
```

```
                    show_models=True,
                    design_control={"init_size": 10},
                    surrogate_control={"noise": False,
                                       "cod_type": "norm",
                                       "min_theta": -4,
                                       "max_theta": 3,
                                       "n_theta": 2,
                                       "model_fun_evals": 1000,
                                       },
                     fun_control=fun_control,)
spot_2_ei_noniso.run()
```

```
spotPython tuning: 1.9573671021907253e-05 [####------] 44.00%

spotPython tuning: 1.9573671021907253e-05 [#####-----] 48.00%

spotPython tuning: 1.9573671021907253e-05 [#####-----] 52.00%

spotPython tuning: 1.0363119355843435e-05 [######----] 56.00%

spotPython tuning: 1.0363119355843435e-05 [######----] 60.00%

spotPython tuning: 1.0363119355843435e-05 [######----] 64.00%

spotPython tuning: 5.269369914423776e-07 [#######---] 68.00%

spotPython tuning: 5.269369914423776e-07 [#######---] 72.00%

spotPython tuning: 5.269369914423776e-07 [########--] 76.00%

spotPython tuning: 5.269369914423776e-07 [########--] 80.00%

spotPython tuning: 5.269369914423776e-07 [########--] 84.00%

spotPython tuning: 5.269369914423776e-07 [#########-] 88.00%

spotPython tuning: 5.269369914423776e-07 [#########-] 92.00%
```

```
spotPython tuning: 5.269369914423776e-07 [##########] 96.00%

spotPython tuning: 5.269369914423776e-07 [##########] 100.00% Done...

<spotPython.spot.spot.Spot at 0x2cd60c450>
```

```
spot_2_ei_noniso.plot_progress(log_y=True)
```




```
spot_2_ei_noniso.print_results()
```

```
min y: 5.269369914423776e-07
x0: 0.0006499472418001477
x1: 0.00032327321930335944

[['x0', 0.0006499472418001477], ['x1', 0.00032327321930335944]]
```

```
spot_2_ei_noniso.surrogate.plot()
```

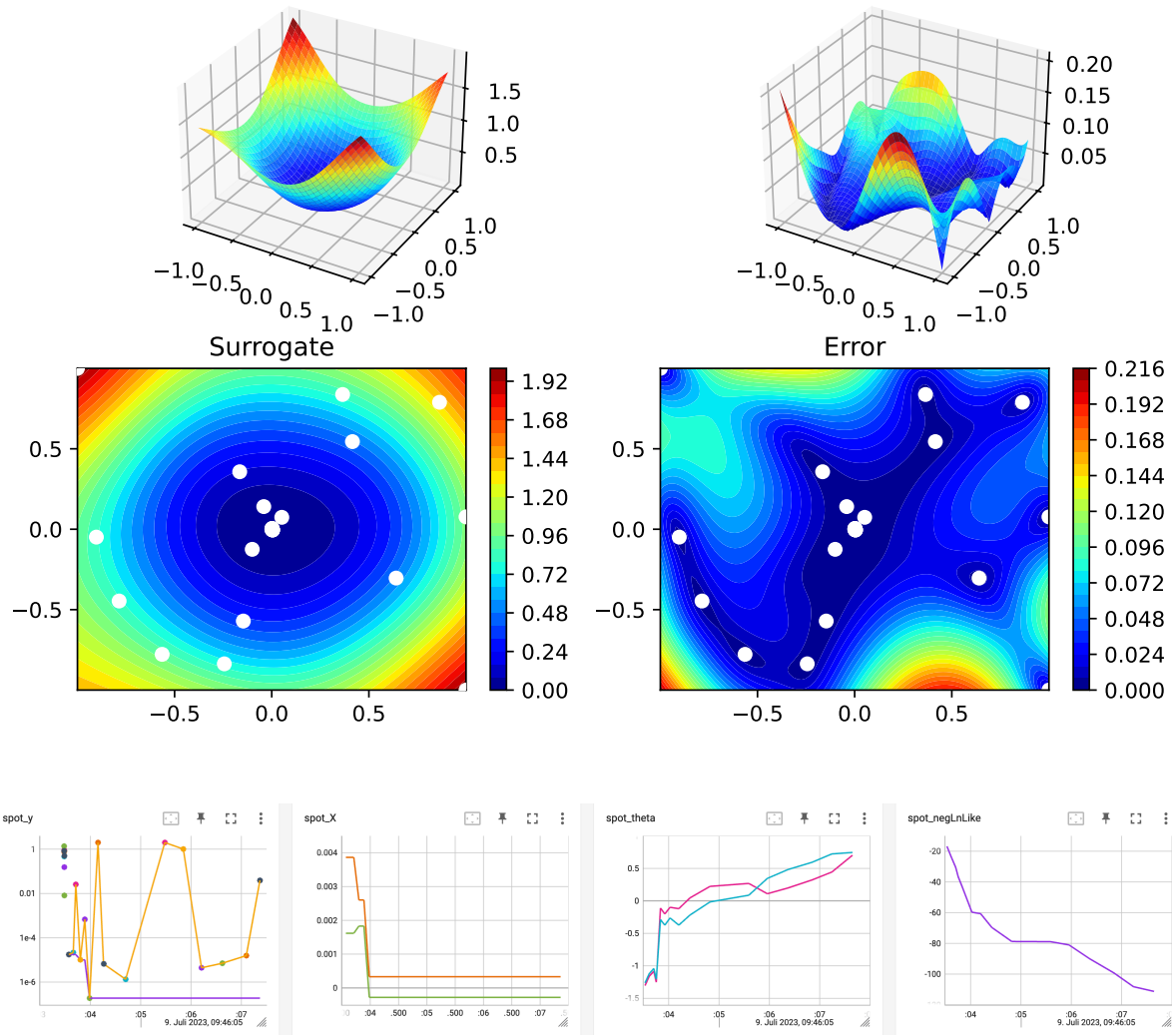


Figure 12.3: TensorBoard visualization of the spotPython optimization process and the surrogate model. Expected improvement, isotropic Kriging.

12.4 Using `sklearn` Surrogates

12.4.1 The spot Loop

The `spot` loop consists of the following steps:

1. Init: Build initial design X
2. Evaluate initial design on real objective f: $y = f(X)$

3. Build surrogate: $S = S(X, y)$
4. Optimize on surrogate: $X_0 = \text{optimize}(S)$
5. Evaluate on real objective: $y_0 = f(X_0)$
6. Impute (Infill) new points: $X = X \cup X_0$, $y = y \cup y_0$.
7. Got 3.

The `spot` loop is implemented in `R` as follows:

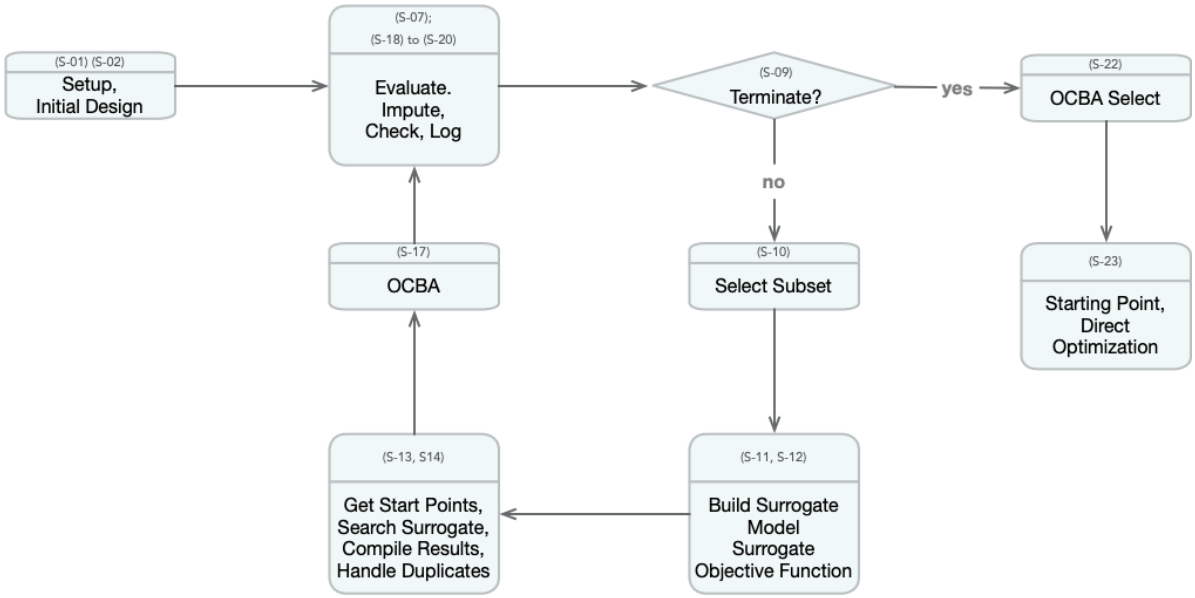


Figure 12.4: Visual representation of the model based search with SPOT. Taken from: Bartz-Beielstein, T., and Zaefferer, M. Hyperparameter tuning approaches. In Hyperparameter Tuning for Machine and Deep Learning with R - A Practical Guide, E. Bartz, T. Bartz-Beielstein, M. Zaefferer, and O. Mersmann, Eds. Springer, 2022, ch. 4, pp. 67–114.

12.4.2 spot: The Initial Model

12.4.2.1 Example: Modifying the initial design size

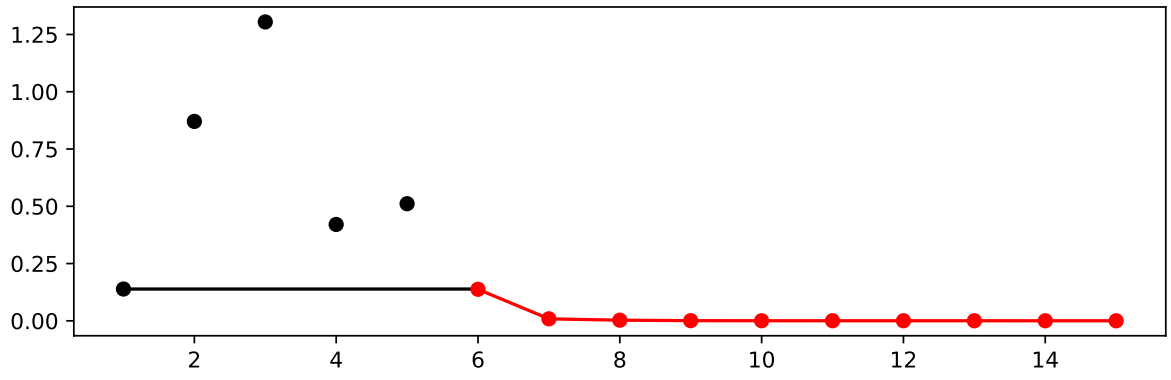
This is the "Example: Modifying the initial design size" from Chapter 4.5.1 in [bart21i].

```
spot_ei = spot.Spot(fun=fun,
                lower = np.array([-1,-1]),
                upper= np.array([1,1]),
                design_control={"init_size": 5})
```

```
spot_ei.run()
```

```
spotPython tuning: 0.13771718056980337 [####------] 40.00%

spotPython tuning: 0.00873860889878173 [#####-----] 46.67%

spotPython tuning: 0.0028322099789017184 [#####-----] 53.33%

spotPython tuning: 0.0008112636408310097 [######----] 60.00%

spotPython tuning: 0.00036558411628539707 [#######---] 66.67%

spotPython tuning: 0.00035830687535911014 [#######---] 73.33%

spotPython tuning: 0.00035830687535911014 [########--] 80.00%

spotPython tuning: 0.00032833591068864664 [#########-] 86.67%

spotPython tuning: 0.0002757670364359785 [#########-] 93.33%

spotPython tuning: 0.0001601071101026825 [##########] 100.00% Done...

<spotPython.spot.spot.Spot at 0x2cda7b010>
```

```
spot_ei.plot_progress()
```



```
np.min(spot_1.y), np.min(spot_ei.y)
```

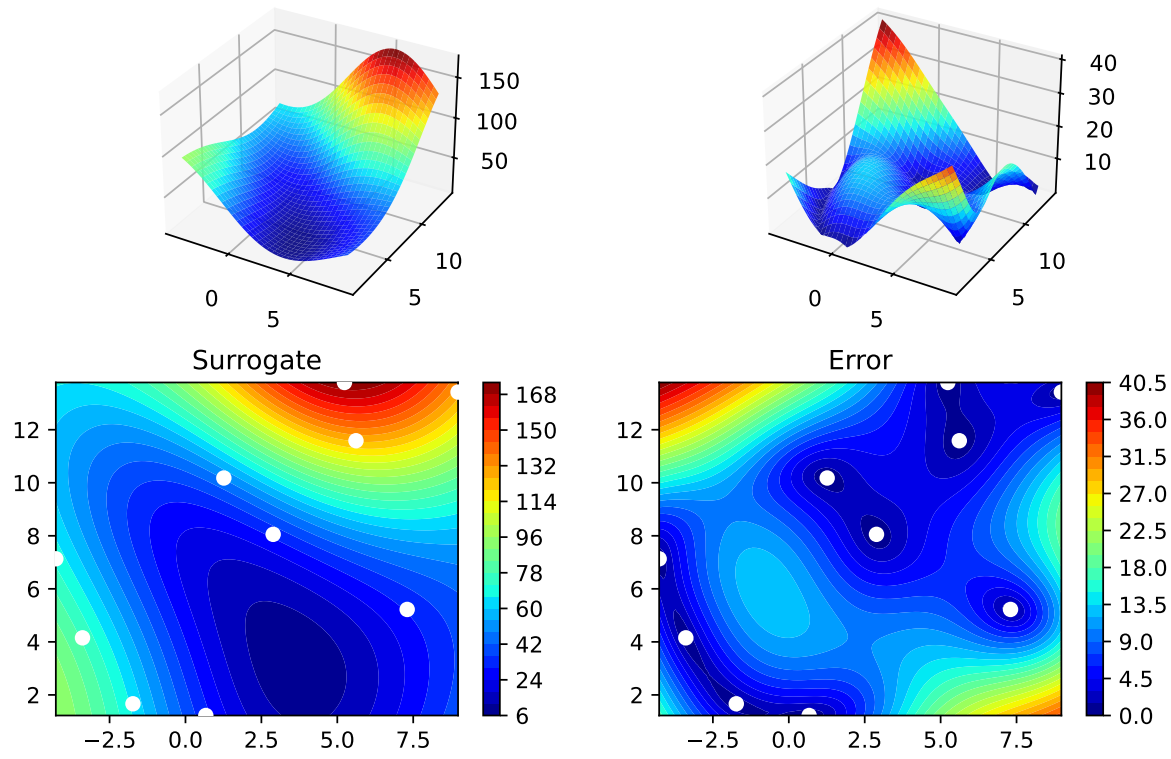
```
(4.9233547910699585e-11, 0.0001601071101026825)
```

12.4.3 Init: Build Initial Design

```
from spotPython.design.spacefilling import spacefilling
from spotPython.build.kriging import Kriging
from spotPython.fun.objectivefunctions import analytical
gen = spacefilling(2)
rng = np.random.RandomState(1)
lower = np.array([-5,-0])
upper = np.array([10,15])
fun = analytical().fun_branin

X = gen.scipy_lhd(10, lower=lower, upper = upper)
print(X)
y = fun(X, fun_control=fun_control)
print(y)
```

```
[[ 8.97647221 13.41926847]
 [ 0.66946019  1.22344228]
 [ 5.23614115 13.78185824]
 [ 5.6149825  11.5851384 ]
 [-1.72963184  1.66516096]
 [-4.26945568  7.1325531 ]
 [ 1.26363761 10.17935555]
 [ 2.88779942  8.05508969]
 [-3.39111089  4.15213772]
 [ 7.30131231  5.22275244]]
[128.95676449  31.73474356 172.89678121 126.71295908  64.34349975
  70.16178611  48.71407916  31.77322887  76.91788181  30.69410529]
```

```
S = Kriging(name='kriging',  seed=123)
S.fit(X, y)
S.plot()
```




```
gen = spacefilling(2, seed=123)
X0 = gen.scipy_lhd(3)
gen = spacefilling(2, seed=345)
X1 = gen.scipy_lhd(3)
X2 = gen.scipy_lhd(3)
gen = spacefilling(2, seed=123)
X3 = gen.scipy_lhd(3)
X0, X1, X2, X3
```

```
(array([[0.77254938, 0.31539299],
        [0.59321338, 0.93854273],
        [0.27469803, 0.3959685 ]]),
 array([[0.78373509, 0.86811887],
        [0.06692621, 0.6058029 ],
        [0.41374778, 0.00525456]]),
 array([[0.121357  , 0.69043832],
        [0.41906219, 0.32838498],
        [0.86742658, 0.52910374]]),
```

```
array([[0.77254938, 0.31539299],
       [0.59321338, 0.93854273],
       [0.27469803, 0.3959685 ]]))
```

12.4.4 Evaluate

12.4.5 Build Surrogate

12.4.6 A Simple Predictor

The code below shows how to use a simple model for prediction.

- Assume that only two (very costly) measurements are available:
  1. f(0) = 0.5
  2. f(2) = 2.5
- We are interested in the value at $x_0 = 1$, i.e., $f(x_0 = 1)$, but cannot run an additional, third experiment.

```
from sklearn import linear_model
X = np.array([[0], [2]])
y = np.array([0.5, 2.5])
S_lm = linear_model.LinearRegression()
S_lm = S_lm.fit(X, y)
X0 = np.array([[1]])
y0 = S_lm.predict(X0)
print(y0)
```

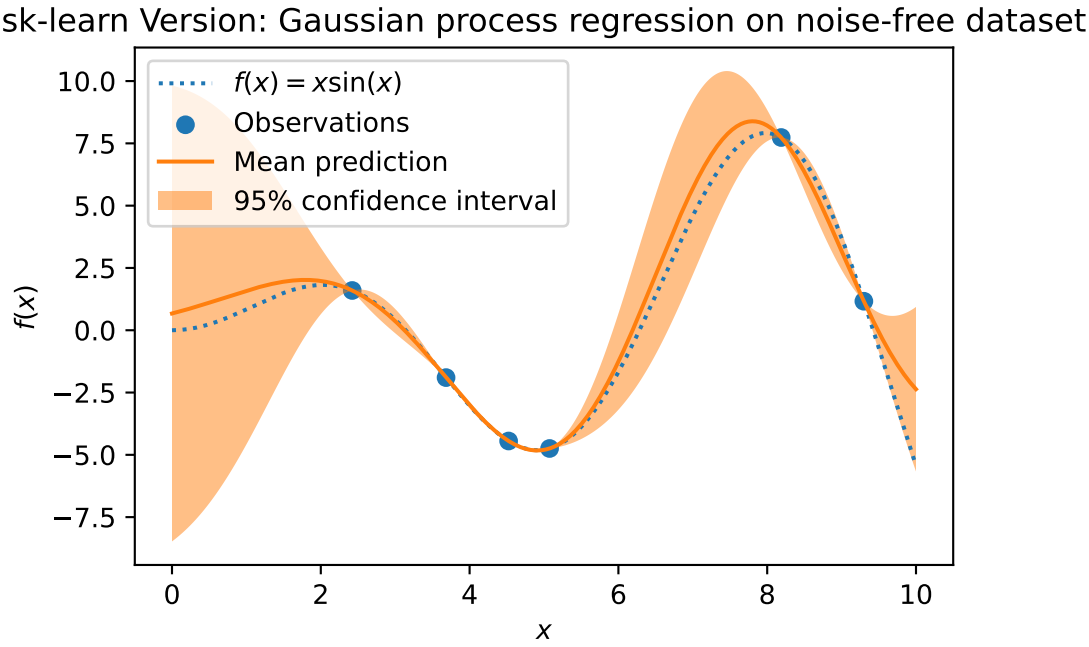
```
[1.5]
```

- Central Idea:
  - Evaluation of the surrogate model `S_lm` is much cheaper (or / and much faster) than running the real-world experiment f.

12.5 Gaussian Processes regression: basic introductory example

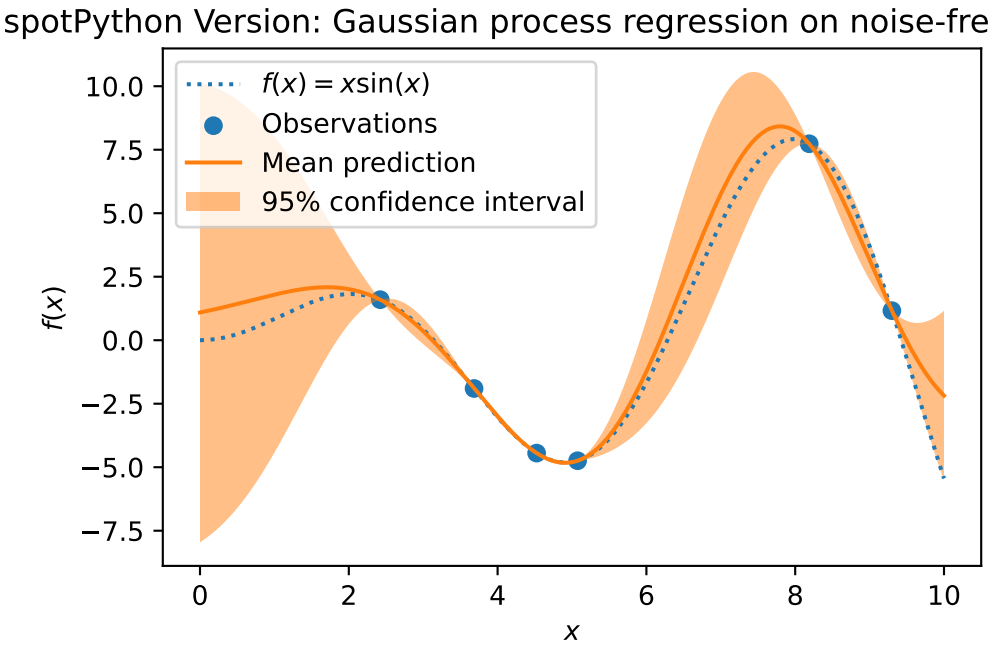
This example was taken from scikit-learn. After fitting our model, we see that the hyperparameters of the kernel have been optimized. Now, we will use our kernel to compute the mean prediction of the full dataset and plot the 95% confidence interval.

```
import numpy as np
import matplotlib.pyplot as plt
import math as m
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF

X = np.linspace(start=0, stop=10, num=1_000).reshape(-1, 1)
y = np.squeeze(X * np.sin(X))
rng = np.random.RandomState(1)
training_indices = rng.choice(np.arange(y.size), size=6, replace=False)
X_train, y_train = X[training_indices], y[training_indices]

kernel = 1 * RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e2))
gaussian_process = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=9)
gaussian_process.fit(X_train, y_train)
gaussian_process.kernel_

mean_prediction, std_prediction = gaussian_process.predict(X, return_std=True)

plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, mean_prediction, label="Mean prediction")
plt.fill_between(
    X.ravel(),
    mean_prediction - 1.96 * std_prediction,
    mean_prediction + 1.96 * std_prediction,
    alpha=0.5,
    label=r"95% confidence interval",
)
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("sk-learn Version: Gaussian process regression on noise-free dataset")
```




```
from spotPython.build.kriging import Kriging
import numpy as np
import matplotlib.pyplot as plt
rng = np.random.RandomState(1)
X = np.linspace(start=0, stop=10, num=1_000).reshape(-1, 1)
y = np.squeeze(X * np.sin(X))
training_indices = rng.choice(np.arange(y.size), size=6, replace=False)
X_train, y_train = X[training_indices], y[training_indices]


S = Kriging(name='kriging',  seed=123, log_level=50, cod_type="norm")
S.fit(X_train, y_train)

mean_prediction, std_prediction, ei = S.predict(X, return_val="all")

std_prediction

plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, mean_prediction, label="Mean prediction")
plt.fill_between(
```

```
    X.ravel(),
    mean_prediction - 1.96 * std_prediction,
    mean_prediction + 1.96 * std_prediction,
    alpha=0.5,
    label=r"95% confidence interval",
)
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("spotPython Version: Gaussian process regression on noise-free dataset")
```



12.6 The Surrogate: Using scikit-learn models

Default is the internal `kriging` surrogate.

```
S_0 = Kriging(name='kriging', seed=123)
```

Models from `scikit-learn` can be selected, e.g., Gaussian Process:

```
# Needed for the sklearn surrogates:
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn import linear_model
from sklearn import tree
import pandas as pd
```

```
kernel = 1 * RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e2))
S_GP = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=9)
```

- and many more:

```
S_Tree = DecisionTreeRegressor(random_state=0)
S_LM = linear_model.LinearRegression()
S_Ridge = linear_model.Ridge()
S_RF = RandomForestRegressor(max_depth=2, random_state=0)
```

- The scikit-learn GP model `S_GP` is selected.

```
S = S_GP
```

```
isinstance(S, GaussianProcessRegressor)
```

```
True
```

```
from spotPython.fun.objectivefunctions import analytical
fun = analytical().fun_branin
lower = np.array([-5,-0])
upper = np.array([10,15])
design_control={"init_size": 5}
surrogate_control={
            "infill_criterion": None,
            "n_points": 1,
        }
spot_GP = spot.Spot(fun=fun, lower = lower, upper= upper, surrogate=S,
                    fun_evals = 15, noise = False, log_level = 50,
                    design_control=design_control,
                    surrogate_control=surrogate_control)
```

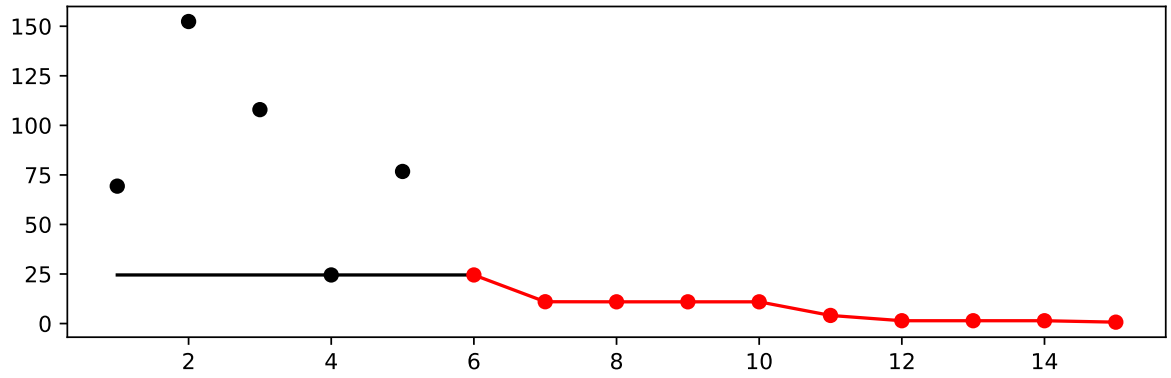
```
spot_GP.run()
```

```
spotPython tuning: 24.51465459019188 [####------] 40.00%

spotPython tuning: 11.003078163486554 [#####-----] 46.67%

spotPython tuning: 10.960665185123245 [#####-----] 53.33%

spotPython tuning: 10.960665185123245 [######----] 60.00%

spotPython tuning: 10.960665185123245 [#######---] 66.67%

spotPython tuning: 4.0894841491438765 [#######---] 73.33%

spotPython tuning: 1.4230377508791392 [########--] 80.00%

spotPython tuning: 1.4230377508791392 [#########-] 86.67%

spotPython tuning: 1.4230377508791392 [#########-] 93.33%

spotPython tuning: 0.6989341031319167 [##########] 100.00% Done...

<spotPython.spot.spot.Spot at 0x2cd57e210>
```

```
spot_GP.y
```

```
array([ 69.32459936, 152.38491454, 107.92560483,  24.51465459,
        76.73500031,  86.304256  ,  11.00307816,  10.96066519,
        16.06668258,  24.08432082,   4.08948415,   1.42303775,
         1.47359526,  16.04703294,   0.6989341 ])
```

```
spot_GP.plot_progress()
```




```
spot_GP.print_results()
```

```
min y: 0.6989341031319167
x0: 3.358292789592623
x1: 2.3886120108545597

[['x0', 3.358292789592623], ['x1', 2.3886120108545597]]
```

12.7 Additional Examples

```
# Needed for the sklearn surrogates:
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn import linear_model
from sklearn import tree
import pandas as pd
```

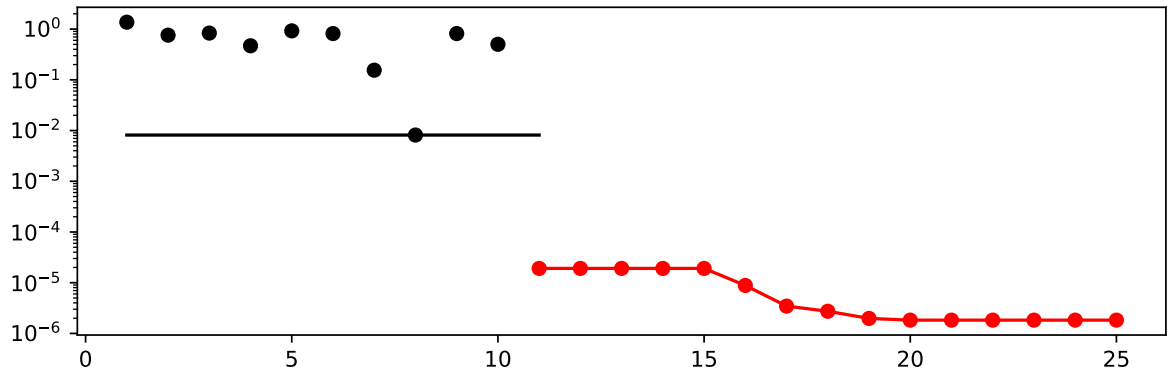
```
kernel = 1 * RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e2))
S_GP = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=9)
```

```
from spotPython.build.kriging import Kriging
import numpy as np
import spotPython
from spotPython.fun.objectivefunctions import analytical
from spotPython.spot import spot

S_K = Kriging(name='kriging',
              seed=123,
              log_level=50,
              infill_criterion = "y",
              n_theta=1,
              noise=False,
              cod_type="norm")
fun = analytical().fun_sphere
lower = np.array([-1,-1])
upper = np.array([1,1])

design_control={"init_size": 10}
surrogate_control={
            "n_points": 1,
        }
spot_S_K = spot.Spot(fun=fun,
                     lower = lower,
                     upper= upper,
                     surrogate=S_K,
                     fun_evals = 25,
                     noise = False,
                     log_level = 50,
                     design_control=design_control,
                     surrogate_control=surrogate_control)

spot_S_K.run()
```

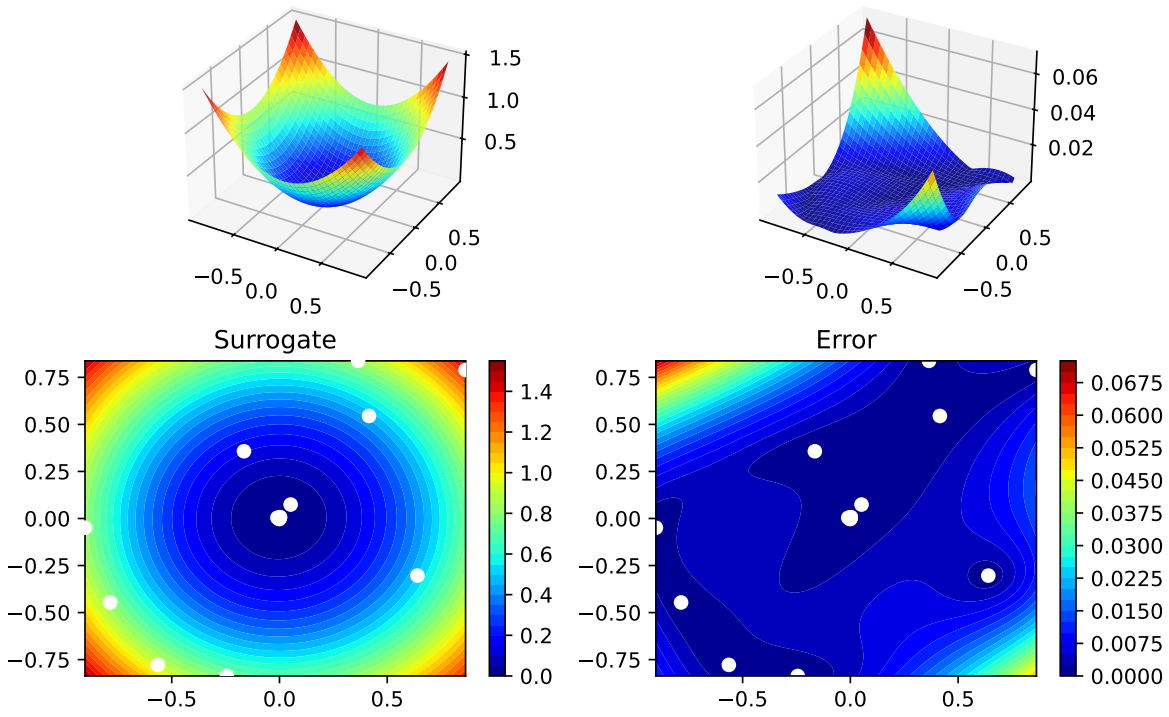
```
spotPython tuning: 1.914791277049731e-05 [####------] 44.00%

spotPython tuning: 1.914791277049731e-05 [#####-----] 48.00%

spotPython tuning: 1.914791277049731e-05 [#####-----] 52.00%

spotPython tuning: 1.914791277049731e-05 [######----] 56.00%
```

```
spotPython tuning: 1.914791277049731e-05 [######----] 60.00%

spotPython tuning: 8.829975201246504e-06 [######----] 64.00%

spotPython tuning: 3.457800382864333e-06 [#######---] 68.00%

spotPython tuning: 2.737454052579376e-06 [#######---] 72.00%

spotPython tuning: 1.9781089716235884e-06 [########--] 76.00%

spotPython tuning: 1.828978093628599e-06 [########--] 80.00%

spotPython tuning: 1.828978093628599e-06 [########--] 84.00%

spotPython tuning: 1.828978093628599e-06 [#########-] 88.00%

spotPython tuning: 1.828978093628599e-06 [#########-] 92.00%

spotPython tuning: 1.828978093628599e-06 [##########] 96.00%

spotPython tuning: 1.828978093628599e-06 [##########] 100.00% Done...

<spotPython.spot.spot.Spot at 0x2d7791090>
```

```
spot_S_K.plot_progress(log_y=True)
```



```
spot_S_K.surrogate.plot()
```




```
spot_S_K.print_results()
```

```
min y: 1.828978093628599e-06
x0: -0.0013506629663128178
x1: 6.846637904591543e-05

[['x0', -0.0013506629663128178], ['x1', 6.846637904591543e-05]]
```

12.7.1 Optimize on Surrogate

12.7.2 Evaluate on Real Objective

12.7.3 Impute / Infill new Points

12.8 Tests

```
import numpy as np
from spotPython.spot import spot
from spotPython.fun.objectivefunctions import analytical

fun_sphere = analytical().fun_sphere
spot_1 = spot.Spot(
    fun=fun_sphere,
    lower=np.array([-1, -1]),
    upper=np.array([1, 1]),
    n_points = 2
)

# (S-2) Initial Design:
spot_1.X = spot_1.design.scipy_lhd(
    spot_1.design_control["init_size"], lower=spot_1.lower, upper=spot_1.upper
)
print(spot_1.X)

# (S-3): Eval initial design:
spot_1.y = spot_1.fun(spot_1.X)
print(spot_1.y)

spot_1.fit_surrogate()
X0 = spot_1.suggest_new_X()
print(X0)
assert X0.size == spot_1.n_points * spot_1.k
```

```
[[ 0.86352963  0.7892358 ]
 [-0.24407197 -0.83687436]
 [ 0.36481882  0.8375811 ]
 [ 0.415331    0.54468512]
 [-0.56395091 -0.77797854]
 [-0.90259409 -0.04899292]]
```

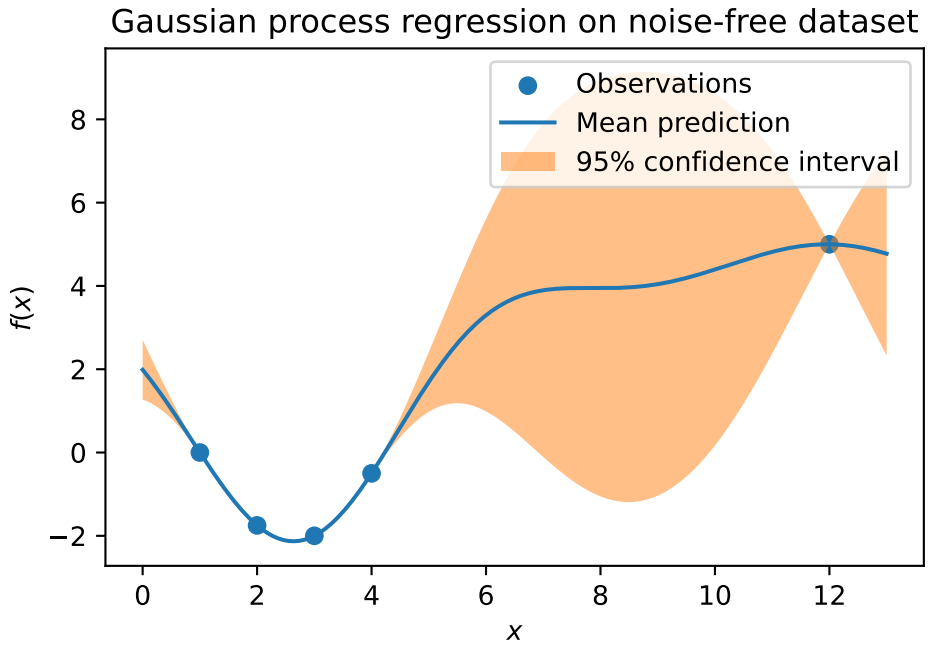
```
 [-0.16484832  0.35724741]
 [ 0.05170659  0.07401196]
 [-0.78548145 -0.44638164]
 [ 0.64017497 -0.30363301]]
[1.36857656 0.75992983 0.83463487 0.46918172 0.92329124 0.8170764
 0.15480068 0.00815134 0.81623768 0.502017  ]
[[0.00165526 0.00410847]
 [0.00165526 0.00410847]]
```

12.9 EI: The Famous Schonlau Example

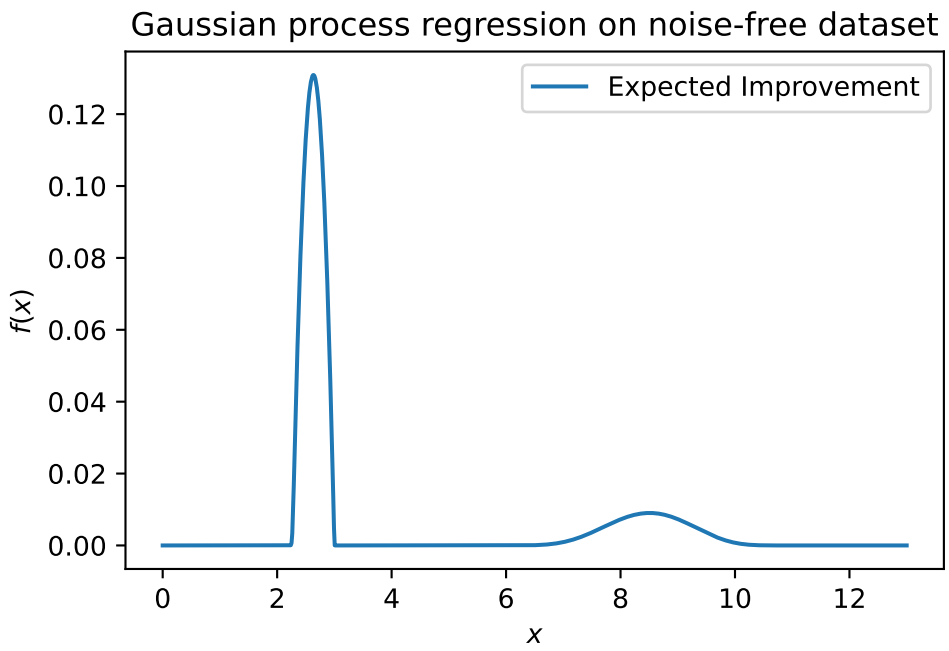
```
X_train0 = np.array([1, 2, 3, 4, 12]).reshape(-1,1)
X_train = np.linspace(start=0, stop=10, num=5).reshape(-1, 1)
```

```
from spotPython.build.kriging import Kriging
import numpy as np
import matplotlib.pyplot as plt

X_train = np.array([1., 2., 3., 4., 12.]).reshape(-1,1)
y_train = np.array([0., -1.75, -2, -0.5, 5.])

S = Kriging(name='kriging',  seed=123, log_level=50, n_theta=1, noise=False, cod_type="nor
S.fit(X_train, y_train)

X = np.linspace(start=0, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, mean_prediction, label="Mean prediction")
if True:
    plt.fill_between(
        X.ravel(),
        mean_prediction - 2 * std_prediction,
        mean_prediction + 2 * std_prediction,
        alpha=0.5,
        label=r"95% confidence interval",
    )
plt.legend()
plt.xlabel("$x$")
```

```
plt.ylabel("$f(x)$")
_ = plt.title("Gaussian process regression on noise-free dataset")
```




```
#plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
# plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, -ei, label="Expected Improvement")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Gaussian process regression on noise-free dataset")
```



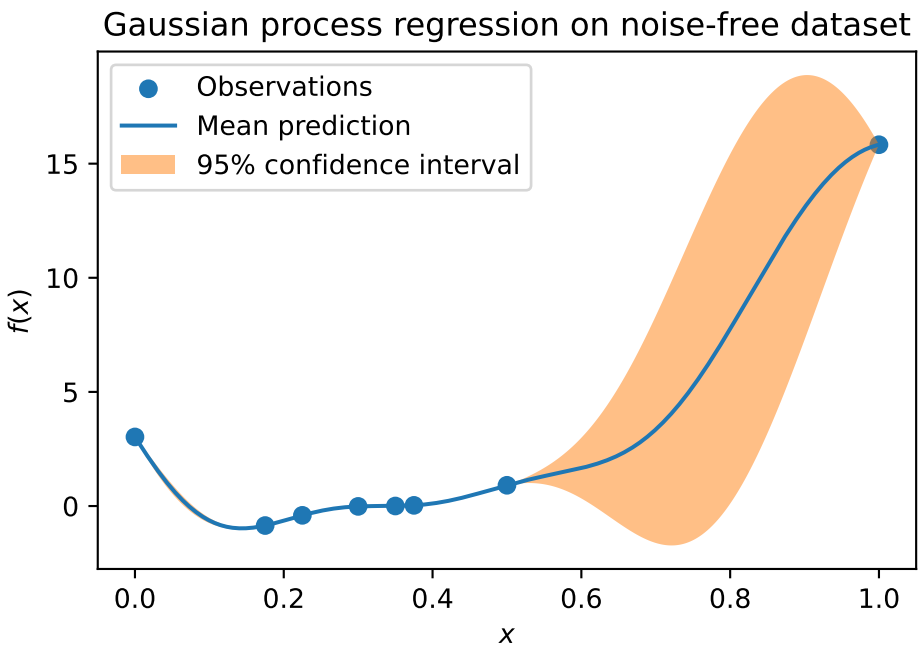

```
S.log
```

```
{'negLnLike': array([1.20788205]),
 'theta': array([-0.9900252]),
 'p': [],
 'Lambda': []}
```

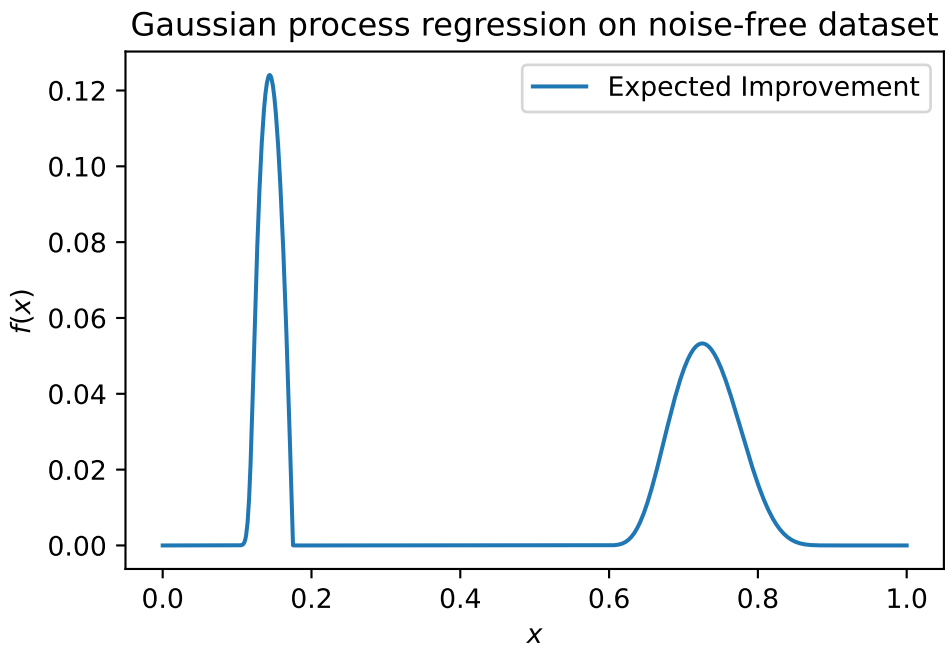
12.10 EI: The Forrester Example

```
from spotPython.build.kriging import Kriging
import numpy as np
import matplotlib.pyplot as plt
import spotPython
from spotPython.fun.objectivefunctions import analytical
from spotPython.spot import spot

# exact x locations are unknown:
X_train = np.array([0.0, 0.175, 0.225, 0.3, 0.35, 0.375, 0.5,1]).reshape(-1,1)
```

```
fun = analytical().fun_forrester
fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    sigma=1.0,
    seed=123,)
y_train = fun(X_train, fun_control=fun_control)

S = Kriging(name='kriging',  seed=123, log_level=50, n_theta=1, noise=False, cod_type="nor
S.fit(X_train, y_train)

X = np.linspace(start=0, stop=1, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, mean_prediction, label="Mean prediction")
if True:
    plt.fill_between(
        X.ravel(),
        mean_prediction - 2 * std_prediction,
        mean_prediction + 2 * std_prediction,
        alpha=0.5,
        label=r"95% confidence interval",
    )
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Gaussian process regression on noise-free dataset")
```




```
#plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
# plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, -ei, label="Expected Improvement")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Gaussian process regression on noise-free dataset")
```

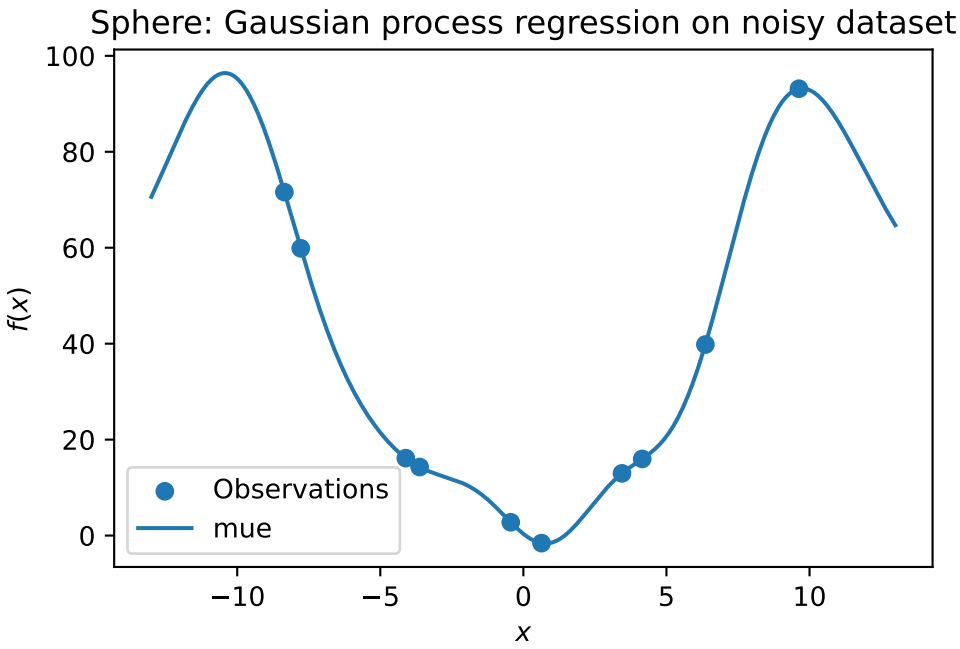


12.11 Noise

```
import numpy as np
import spotPython
from spotPython.fun.objectivefunctions import analytical
from spotPython.spot import spot
from spotPython.design.spacefilling import spacefilling
from spotPython.build.kriging import Kriging
import matplotlib.pyplot as plt

gen = spacefilling(1)
rng = np.random.RandomState(1)
lower = np.array([-10])
upper = np.array([10])
fun = analytical().fun_sphere
fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    sigma=2.0,
    seed=123,)
X = gen.scipy_lhd(10, lower=lower, upper = upper)
```

```
print(X)
y = fun(X, fun_control=fun_control)
print(y)
y.shape
X_train = X.reshape(-1,1)
y_train = y

S = Kriging(name='kriging',
            seed=123,
            log_level=50,
            n_theta=1,
            noise=False)
S.fit(X_train, y_train)

X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

#plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
#plt.plot(X, ei, label="Expected Improvement")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Sphere: Gaussian process regression on noisy dataset")
```

```
[[ 0.63529627]
 [-4.10764204]
 [-0.44071975]
 [ 9.63125638]
 [-8.3518118 ]
 [-3.62418901]
 [ 4.15331   ]
 [ 3.4468512 ]
 [ 6.36049088]
 [-7.77978539]]
[-1.57464135 16.13714981  2.77008442 93.14904827 71.59322218 14.28895359
 15.9770567  12.96468767 39.82265329 59.88028242]
```

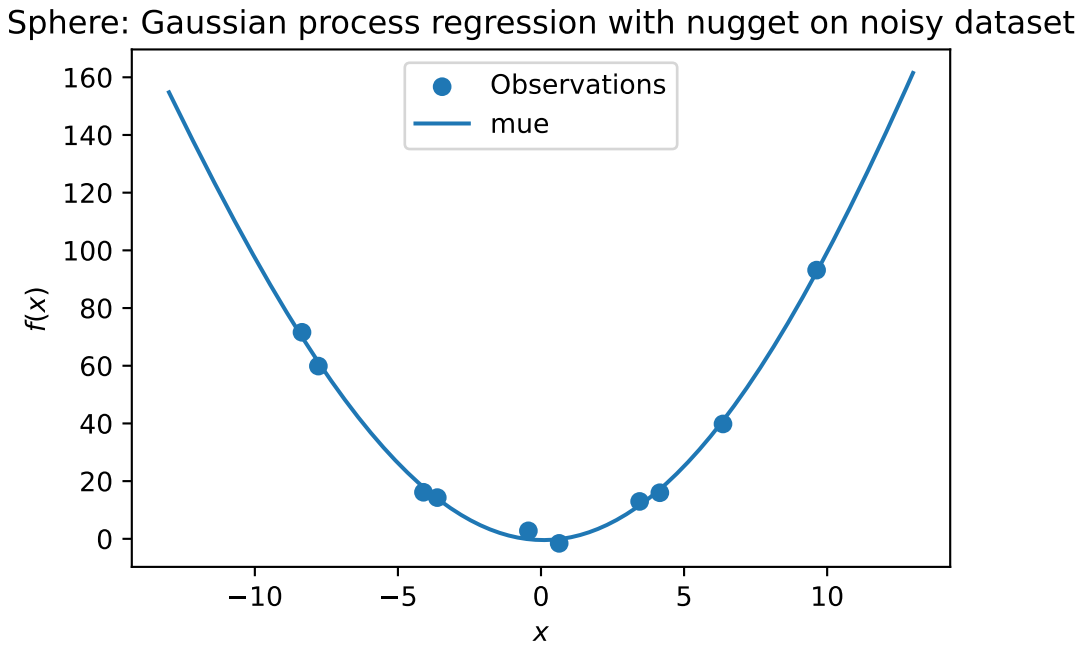



```
S.log
```

```
{'negLnLike': array([26.18505386]),
 'theta': array([-1.10547474]),
 'p': [],
 'Lambda': []}
```

```
S = Kriging(name='kriging',
            seed=123,
            log_level=50,
            n_theta=1,
            noise=True)
S.fit(X_train, y_train)

X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

#plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
#plt.plot(X, ei, label="Expected Improvement")
plt.plot(X_axis, mean_prediction, label="mue")
```

```
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Sphere: Gaussian process regression with nugget on noisy dataset")
```




```
S.log
```

```
{'negLnLike': array([21.82059174]),
 'theta': array([-2.96946062]),
 'p': [],
 'Lambda': array([4.28985898e-05])}
```

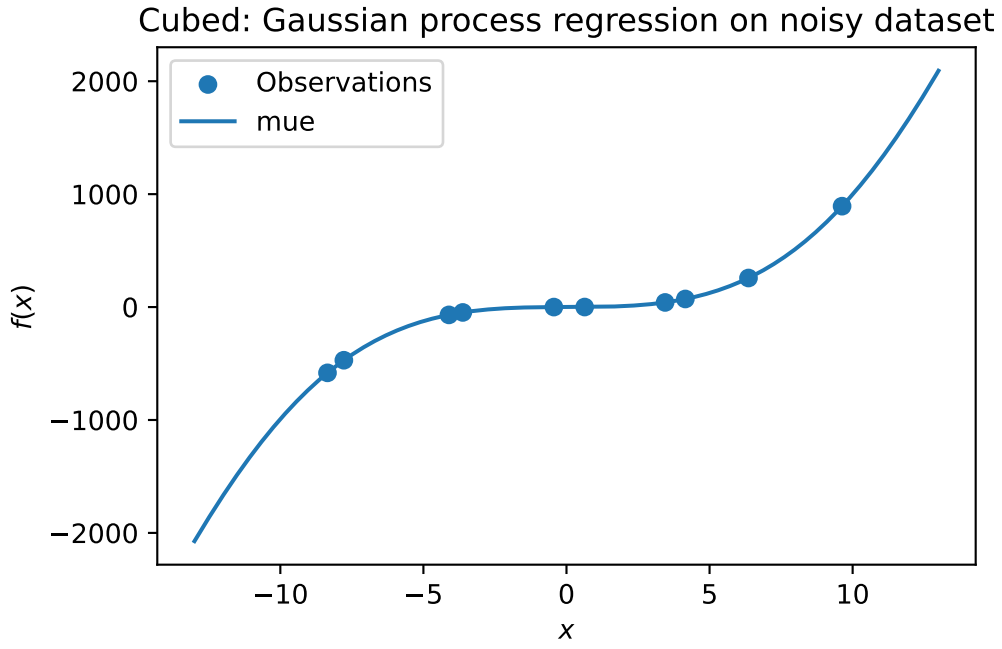
12.12 Cubic Function

```
import numpy as np
import spotPython
from spotPython.fun.objectivefunctions import analytical
from spotPython.spot import spot
from spotPython.design.spacefilling import spacefilling
```

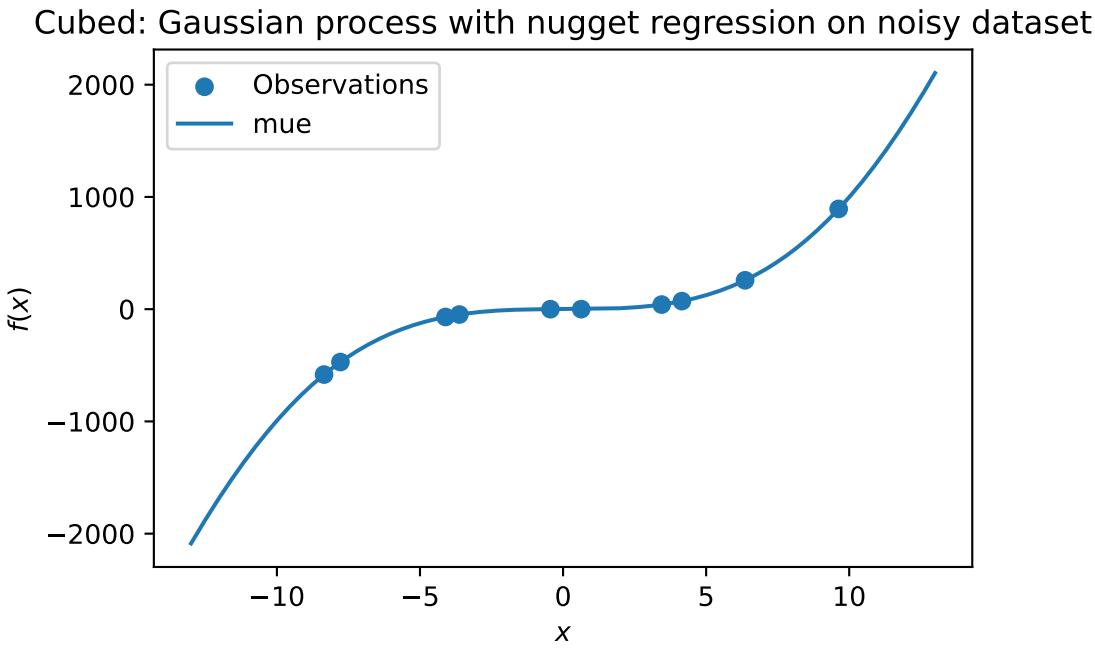
```
from spotPython.build.kriging import Kriging
import matplotlib.pyplot as plt

gen = spacefilling(1)
rng = np.random.RandomState(1)
lower = np.array([-10])
upper = np.array([10])
fun = analytical().fun_cubed
fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    sigma=10.0,
    seed=123,)

X = gen.scipy_lhd(10, lower=lower, upper = upper)
print(X)
y = fun(X, fun_control=fun_control)
print(y)
y.shape
X_train = X.reshape(-1,1)
y_train = y

S = Kriging(name='kriging',  seed=123, log_level=50, n_theta=1, noise=False)
S.fit(X_train, y_train)

X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
#plt.plot(X, ei, label="Expected Improvement")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Cubed: Gaussian process regression on noisy dataset")
```

```
[[ 0.63529627]
 [-4.10764204]
 [-0.44071975]
 [ 9.63125638]
 [-8.3518118 ]
 [-3.62418901]
 [ 4.15331   ]
```

```
 [ 3.4468512 ]
 [ 6.36049088]
 [-7.77978539]]
[ 2.56406437e-01 -6.93071067e+01 -8.56027124e-02  8.93405931e+02
 -5.82561927e+02 -4.76028022e+01  7.16445311e+01  4.09512920e+01
  2.57319028e+02 -4.70871982e+02]
```



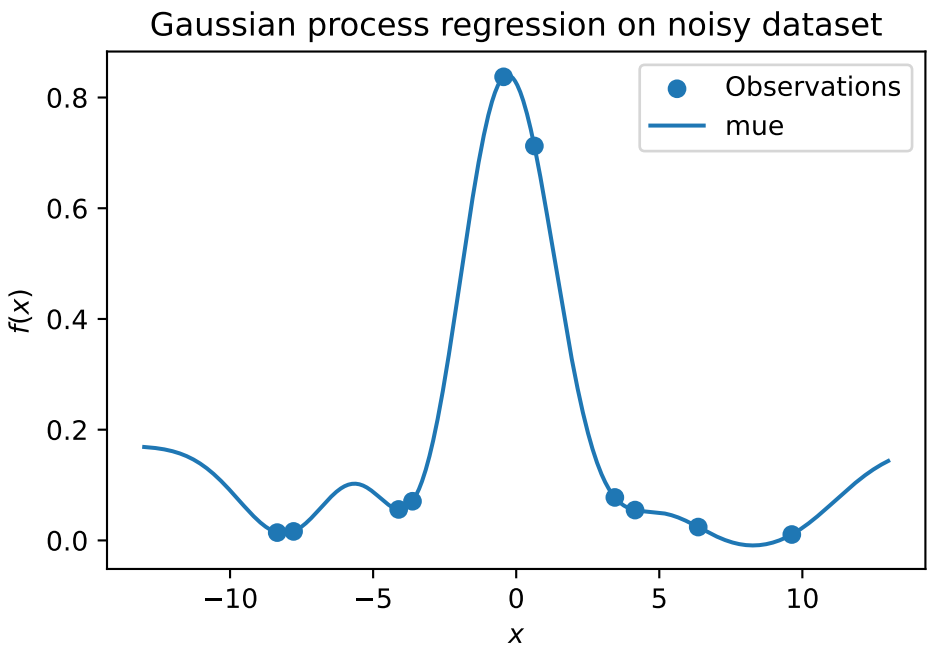

```
S = Kriging(name='kriging',  seed=123, log_level=0, n_theta=1, noise=True)
S.fit(X_train, y_train)

X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
#plt.plot(X, ei, label="Expected Improvement")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Cubed: Gaussian process with nugget regression on noisy dataset")
```



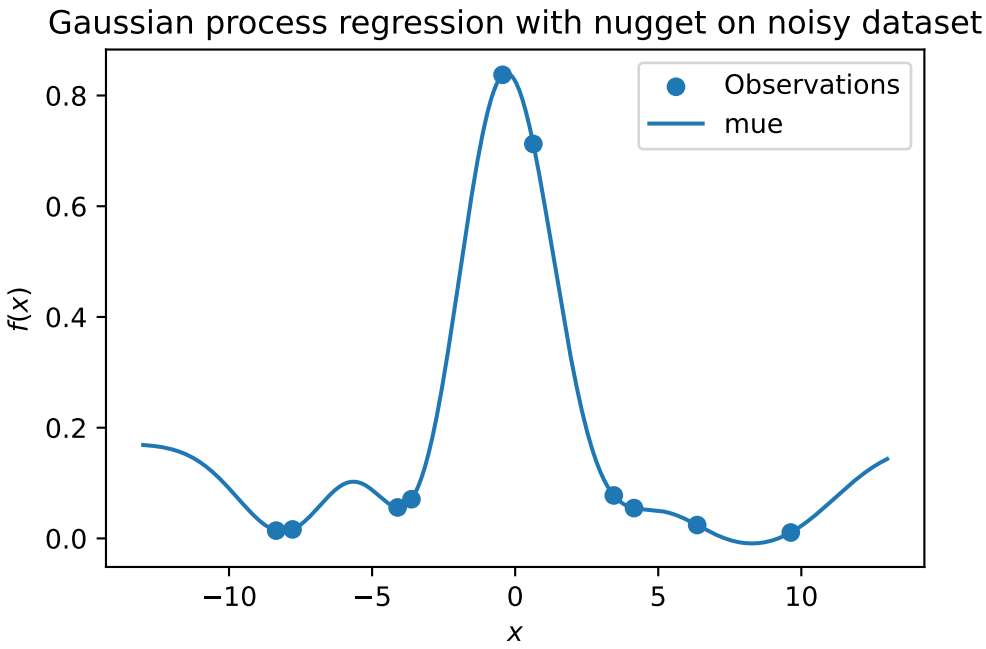

```
import numpy as np
import spotPython
from spotPython.fun.objectivefunctions import analytical
from spotPython.spot import spot
from spotPython.design.spacefilling import spacefilling
from spotPython.build.kriging import Kriging
import matplotlib.pyplot as plt

gen = spacefilling(1)
rng = np.random.RandomState(1)
lower = np.array([-10])
upper = np.array([10])
fun = analytical().fun_runge
fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    sigma=0.25,
    seed=123,)

X = gen.scipy_lhd(10, lower=lower, upper = upper)
print(X)
y = fun(X, fun_control=fun_control)
```

```
print(y)
y.shape
X_train = X.reshape(-1,1)
y_train = y

S = Kriging(name='kriging',  seed=123, log_level=50, n_theta=1, noise=False)
S.fit(X_train, y_train)

X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
#plt.plot(X, ei, label="Expected Improvement")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Gaussian process regression on noisy dataset")
```

```
[[ 0.63529627]
 [-4.10764204]
 [-0.44071975]
 [ 9.63125638]
 [-8.3518118 ]
 [-3.62418901]
 [ 4.15331   ]
 [ 3.4468512 ]
 [ 6.36049088]
 [-7.77978539]]
[0.712453   0.05595118 0.83735691 0.0106654  0.01413372 0.07074765
 0.05479457 0.07763503 0.02412205 0.01625354]
```




```
S = Kriging(name='kriging',
            seed=123,
            log_level=50,
            n_theta=1,
            noise=True)
S.fit(X_train, y_train)

X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
#plt.plot(X, ei, label="Expected Improvement")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Gaussian process regression with nugget on noisy dataset")
```



12.13 Factors

```
["num"] * 3
```

```
['num', 'num', 'num']
```

```
from spotPython.design.spacefilling import spacefilling
from spotPython.build.kriging import Kriging
from spotPython.fun.objectivefunctions import analytical
import numpy as np
```

```
gen = spacefilling(2)
n = 30
rng = np.random.RandomState(1)
lower = np.array([-5,-0])
upper = np.array([10,15])
fun = analytical().fun_branin_factor
#fun = analytical(sigma=0).fun_sphere
```

```
X0 = gen.scipy_lhd(n, lower=lower, upper = upper)
X1 = np.random.randint(low=1, high=3, size=(n,))
X = np.c_[X0, X1]
y = fun(X)
S = Kriging(name='kriging',  seed=123, log_level=50, n_theta=3, noise=False, var_type=["nu
S.fit(X, y)
Sf = Kriging(name='kriging',  seed=123, log_level=50, n_theta=3, noise=False, var_type=["n
Sf.fit(X, y)
n = 50
X0 = gen.scipy_lhd(n, lower=lower, upper = upper)
X1 = np.random.randint(low=1, high=3, size=(n,))
X = np.c_[X0, X1]
y = fun(X)
s=np.sum(np.abs(S.predict(X)[0] - y))
sf=np.sum(np.abs(Sf.predict(X)[0] - y))
sf - s
```

```
-40.49050199646808
```

```
# vars(S)
```

```
# vars(Sf)
```

13 Handling Noise

This chapter demonstrates how noisy functions can be handled by `Spot` and how noise can be simulated, i.e., added to the objective function.

13.1 Example: `Spot` and the Noisy Sphere Function

```
import numpy as np
from math import inf
from spotPython.fun.objectivefunctions import analytical
from spotPython.spot import spot
import matplotlib.pyplot as plt
from spotPython.utils.file import get_experiment_name
from spotPython.utils.init import fun_control_init
from spotPython.utils.file import get_spot_tensorboard_path

PREFIX = "08"
experiment_name = get_experiment_name(prefix=PREFIX)
print(experiment_name)
```

```
08_p040025_2023-12-30_22-59-24
```

13.1.1 The Objective Function: Noisy Sphere

The `spotPython` package provides several classes of objective functions, which return a one-dimensional output $y = f(x)$ for a given input x (independent variable). Several objective functions allow one- or multidimensional input, some also combinations of real-valued and categorial input values.

An objective function is considered as "analytical" if it can be described by a closed mathematical formula, e.g.,

$$f(x, y) = x^2 + y^2.$$

To simulate measurement errors, adding artificial noise to the function value y is a common practice, e.g.,:

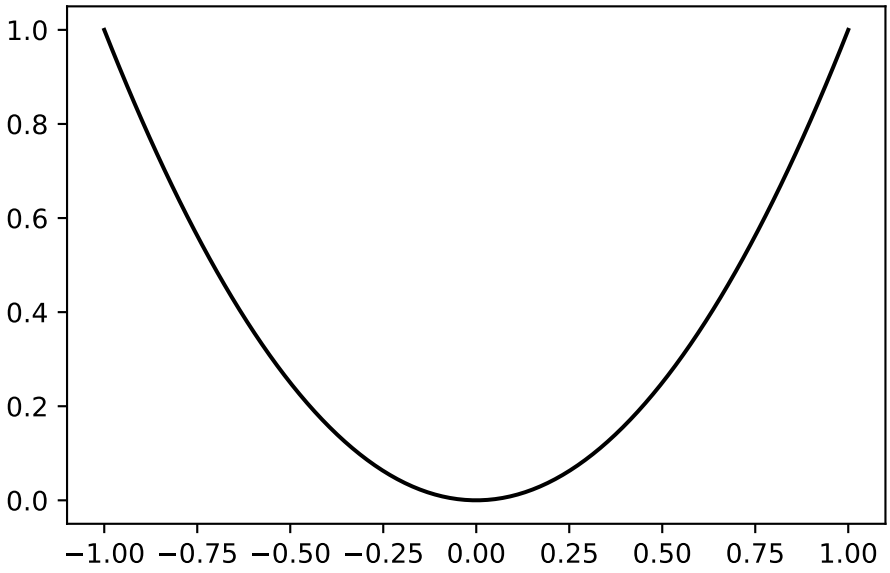
$$f(x, y) = x^2 + y^2 + \epsilon.$$

Usually, noise is assumed to be normally distributed with mean $\mu = 0$ and standard deviation σ. spotPython uses numpy's `scale` parameter, which specifies the standard deviation (spread or "width") of the distribution is used. This must be a non-negative value, see https://numpy.org/doc/stable/reference/random/generated/numpy.random.normal.html.

Example: The sphere function without noise

The default setting does not use any noise.

```
from spotPython.fun.objectivefunctions import analytical
fun = analytical().fun_sphere
x = np.linspace(-1,1,100).reshape(-1,1)
y = fun(x)
plt.figure()
plt.plot(x,y, "k")
plt.show()
```



Example: The sphere function with noise

Noise can be added to the sphere function as follows:

```
from spotPython.fun.objectivefunctions import analytical
fun = analytical(seed=123, sigma=0.02).fun_sphere
x = np.linspace(-1,1,100).reshape(-1,1)
y = fun(x)
plt.figure()
plt.plot(x,y, "k")
plt.show()
```

13.1.2 Reproducibility: Noise Generation and Seed Handling

spotPython provides two mechanisms for generating random noise:

1. The seed is initialized once, i.e., when the objective function is instantiated. This can be done using the following call: `fun = analytical(sigma=0.02, seed=123).fun_sphere`.
2. The seed is set every time the objective function is called. This can be done using the following call: `y = fun(x, sigma=0.02, seed=123)`.

These two different ways lead to different results as explained in the following tables:

Example: Noise added to the sphere function

Since `sigma` is set to `0.02`, noise is added to the function:

```
from spotPython.fun.objectivefunctions import analytical
fun = analytical(sigma=0.02, seed=123).fun_sphere
x = np.array([1]).reshape(-1,1)
for i in range(3):
    print(f"{i}: {fun(x)}")
```

```
0: [0.98021757]
1: [0.99264427]
2: [1.02575851]
```

The seed is set once. Every call to `fun()` results in a different value. The whole experiment can be repeated, the initial seed is used to generate the same sequence as shown below:

Example: Noise added to the sphere function

Since `sigma` is set to `0.02`, noise is added to the function:

```
from spotPython.fun.objectivefunctions import analytical
fun = analytical(sigma=0.02, seed=123).fun_sphere
x = np.array([1]).reshape(-1,1)
for i in range(3):
    print(f"{i}: {fun(x)}")
```

```
0: [0.98021757]
1: [0.99264427]
2: [1.02575851]
```

If `spotPython` is used as a hyperparameter tuner, it is important that only one realization of the noise function is optimized. This behaviour can be accomplished by passing the same seed via the dictionary `fun_control` to every call of the objective function `fun` as shown below:

Example: The same noise added to the sphere function

Since `sigma` is set to `0.02`, noise is added to the function:

```
from spotPython.fun.objectivefunctions import analytical
fun = analytical().fun_sphere
fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    sigma=0.02,
    seed=123,)
y = fun(x, fun_control=fun_control)
x = np.array([1]).reshape(-1,1)
for i in range(3):
    print(f"{i}: {fun(x)}")
```

```
0: [0.98021757]
1: [0.98021757]
2: [0.98021757]
```

13.2 spotPython's Noise Handling Approaches

The following setting will be used for the next steps:

```
fun = analytical().fun_sphere
fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    sigma=0.02,
    seed=123,)
```

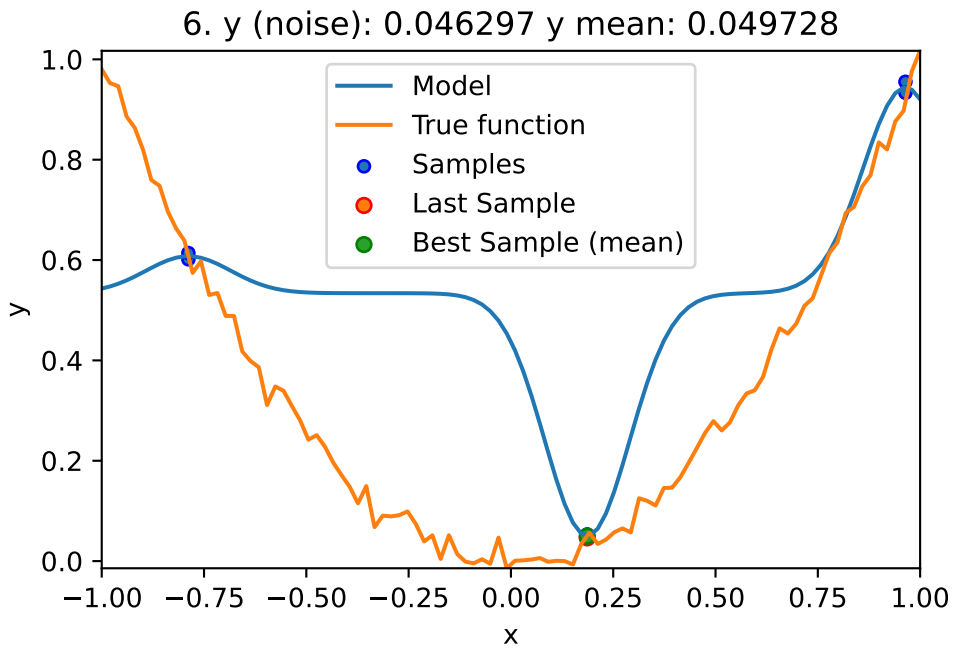
`spotPython` is adopted as follows to cope with noisy functions:

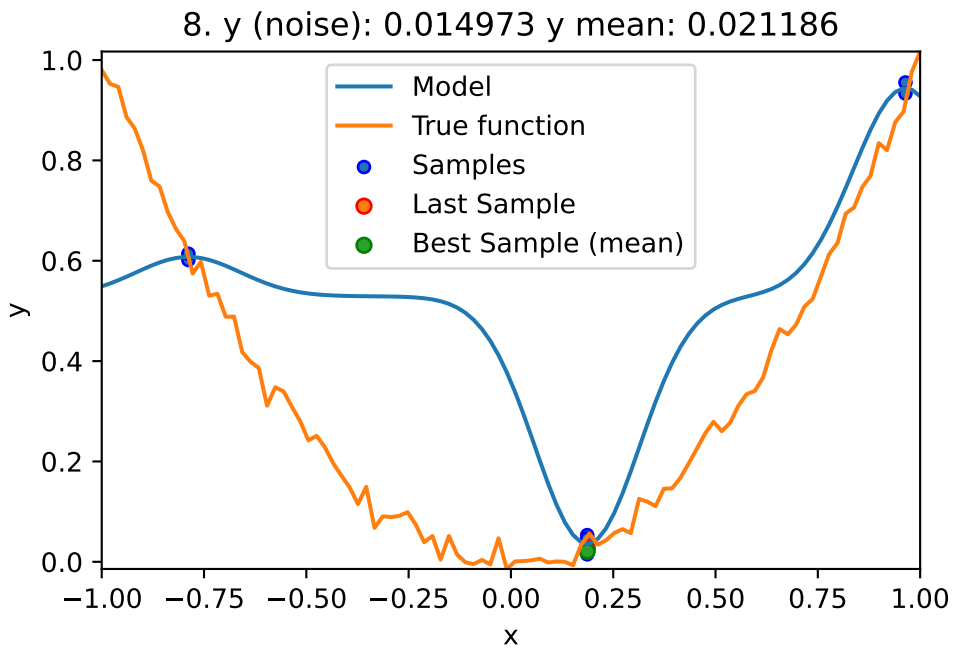
1. `fun_repeats` is set to a value larger than 1 (here: 2)
2. `noise` is set to `true`. Therefore, a nugget (`Lambda`) term is added to the correlation matrix
3. `init size` (of the `design_control` dictionary) is set to a value larger than 1 (here: 3)

```
spot_1_noisy = spot.Spot(fun=fun,
                    lower = np.array([-1]),
                    upper = np.array([1]),
                    fun_evals = 20,
                    fun_repeats = 2,
                    noise = True,
                    seed=123,
```

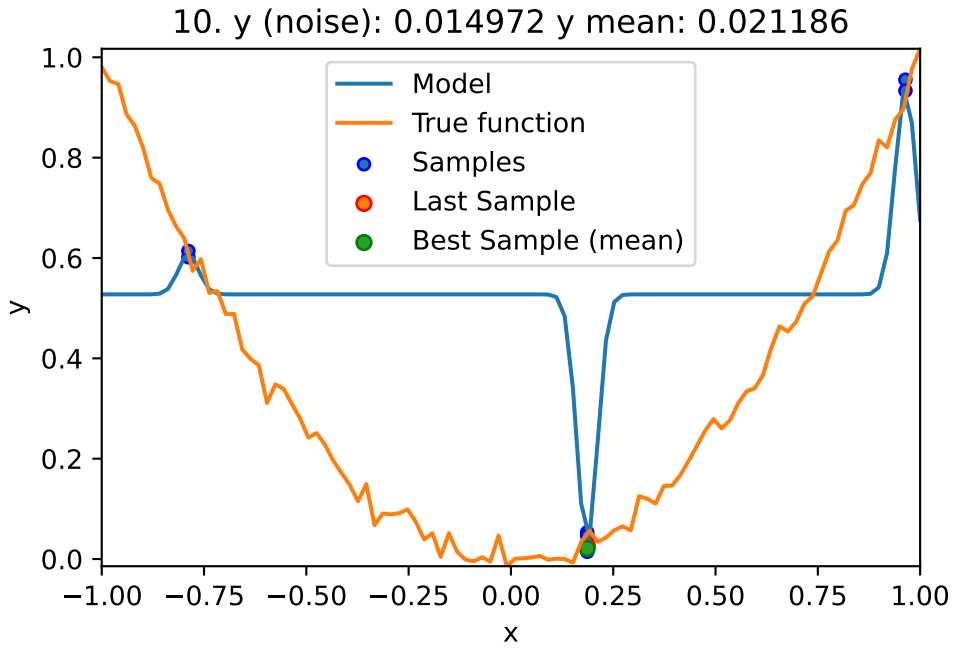
```
                    show_models=True,
                    design_control={"init_size": 3,
                                    "repeats": 2},
                    surrogate_control={"noise": True},
                    fun_control=fun_control,)
```

```
spot_1_noisy.run()
```

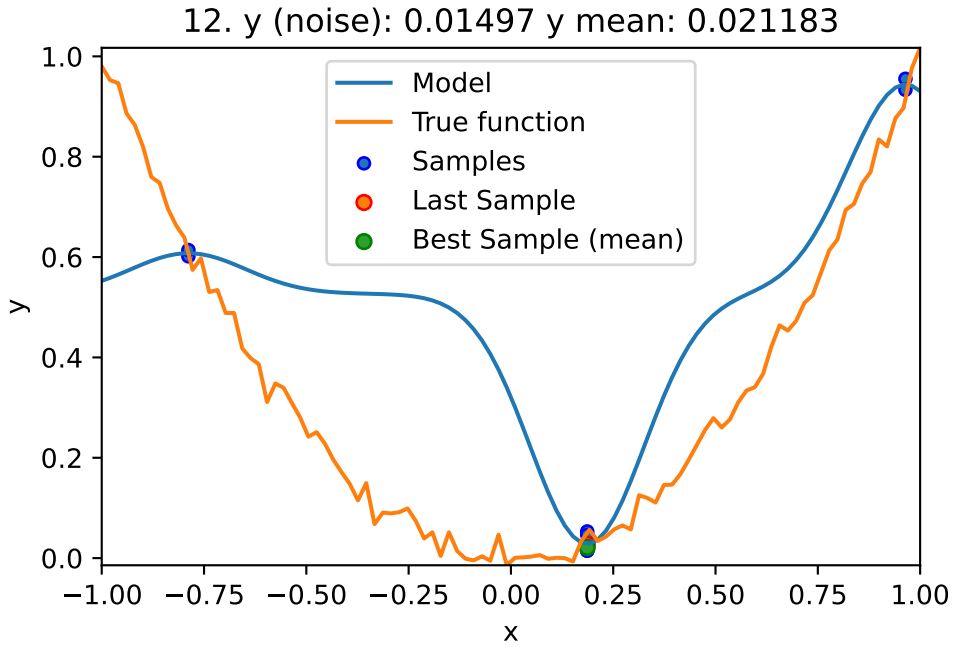




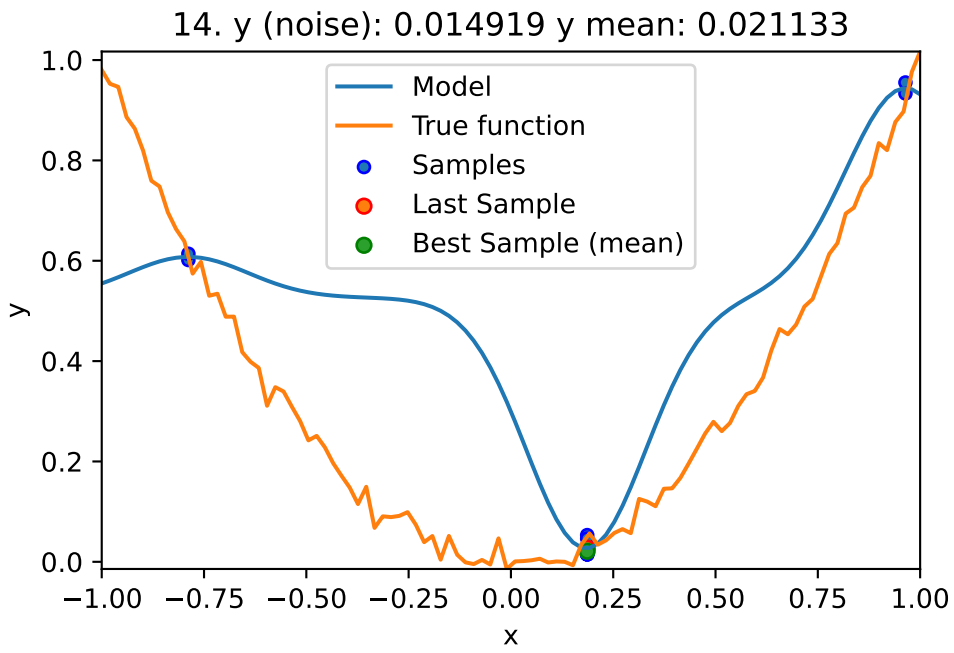

```
spotPython tuning: 0.01497250386441857 [####------] 40.00%
```



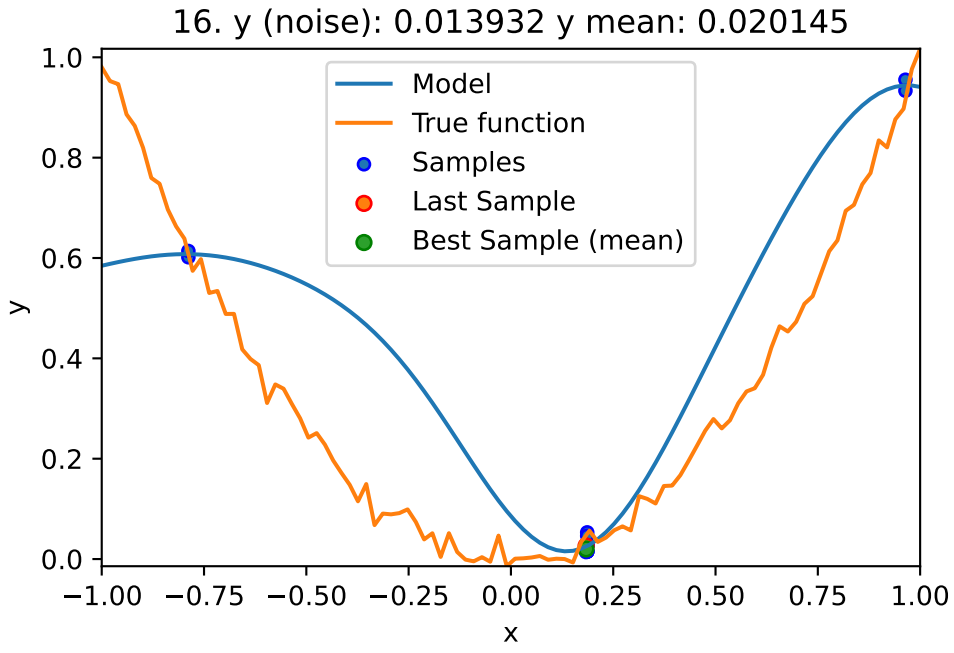
```
spotPython tuning: 0.014972409228673794 [#####-----] 50.00%
```



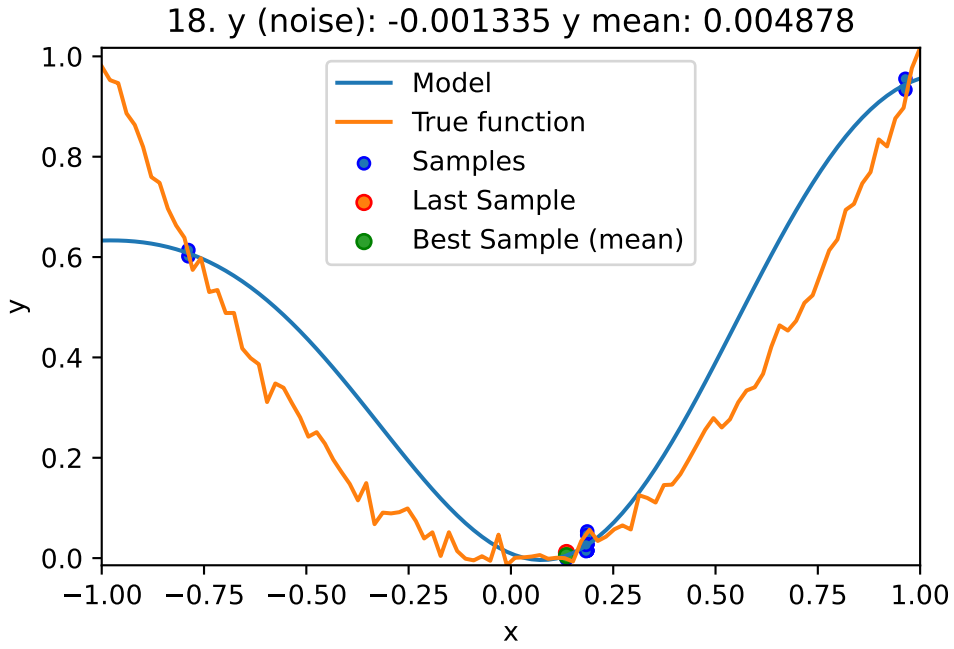

```
spotPython tuning: 0.014970051809890264 [######----] 60.00%
```



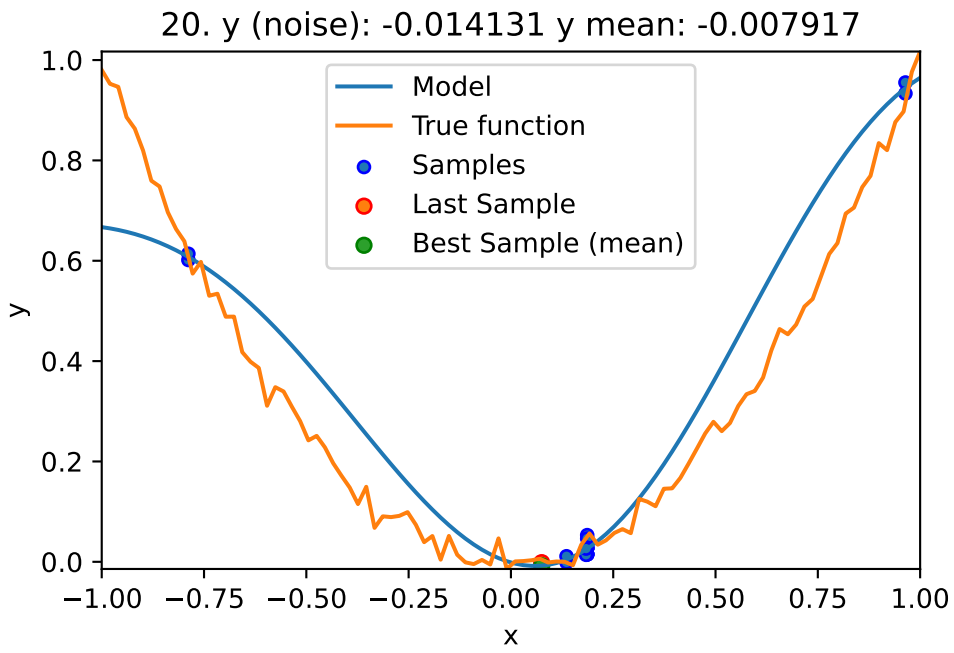

```
spotPython tuning: 0.014919266139021774 [#######---] 70.00%
```



```
spotPython tuning: 0.013931945695333382 [########--] 80.00%
```




```
spotPython tuning: -0.0013351470199611352 [#########-] 90.00%
```




```
spotPython tuning: -0.014130542563158436 [##########] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2cece2850>
```

13.3 Print the Results

```
spot_1_noisy.print_results()
```

```
min y: -0.014130542563158436
x0: 0.07517901598051534
min mean y: -0.00791719557435876
x0: 0.07517901598051534
```

```
[['x0', 0.07517901598051534], ['x0', 0.07517901598051534]]
```

```
spot_1_noisy.plot_progress(log_y=False,
    filename="./figures/" + experiment_name+"_progress.png")
```

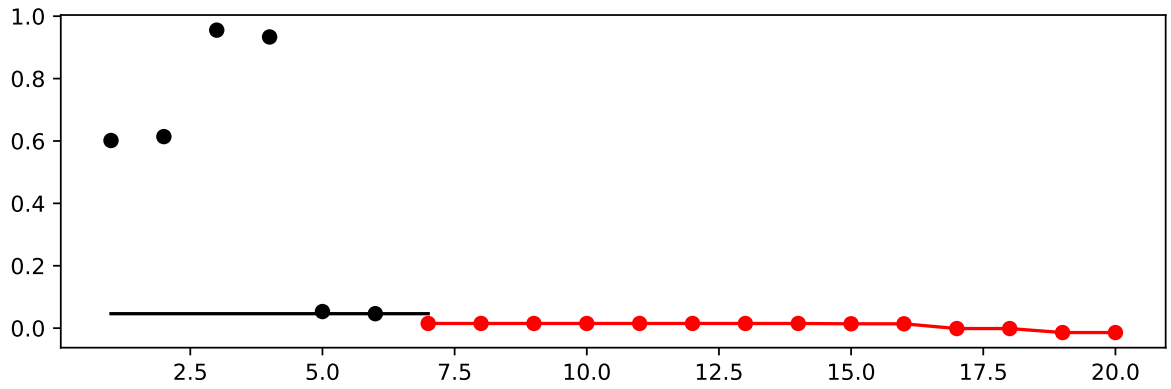


Figure 13.1: Progress plot. *Black* dots denote results from the initial design. *Red* dots illustrate the improvement found by the surrogate model based optimization.

13.4 Noise and Surrogates: The Nugget Effect

13.4.1 The Noisy Sphere

13.4.1.1 The Data

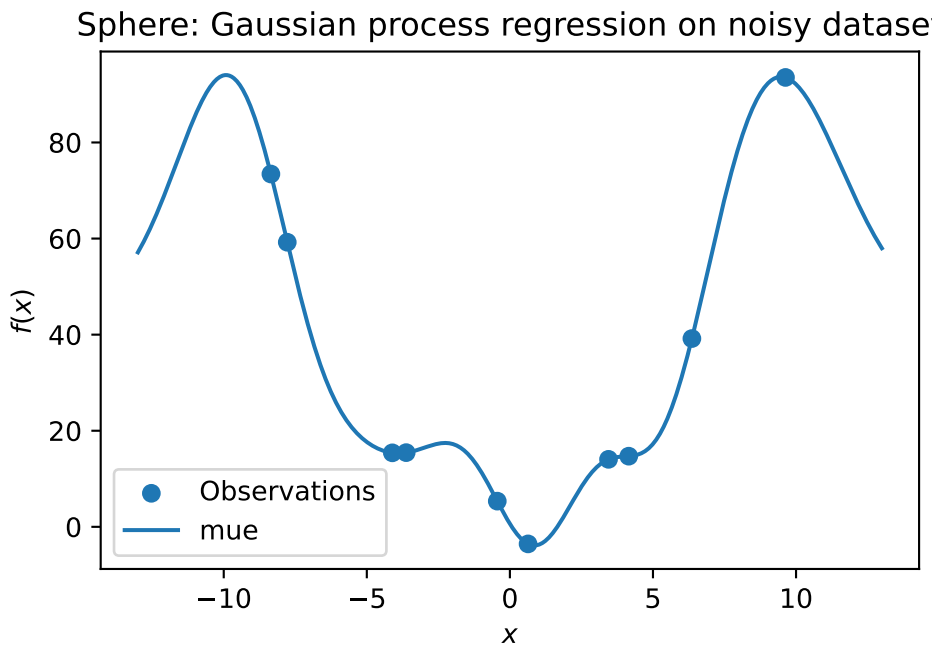
- We prepare some data first:

```
import numpy as np
import spotPython
from spotPython.fun.objectivefunctions import analytical
from spotPython.spot import spot
from spotPython.design.spacefilling import spacefilling
from spotPython.build.kriging import Kriging
import matplotlib.pyplot as plt

gen = spacefilling(1)
rng = np.random.RandomState(1)
lower = np.array([-10])
upper = np.array([10])
fun = analytical().fun_sphere
fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    sigma=4,
    seed=123,)
```

```
X = gen.scipy_lhd(10, lower=lower, upper = upper)
y = fun(X, fun_control=fun_control)
X_train = X.reshape(-1,1)
y_train = y
```

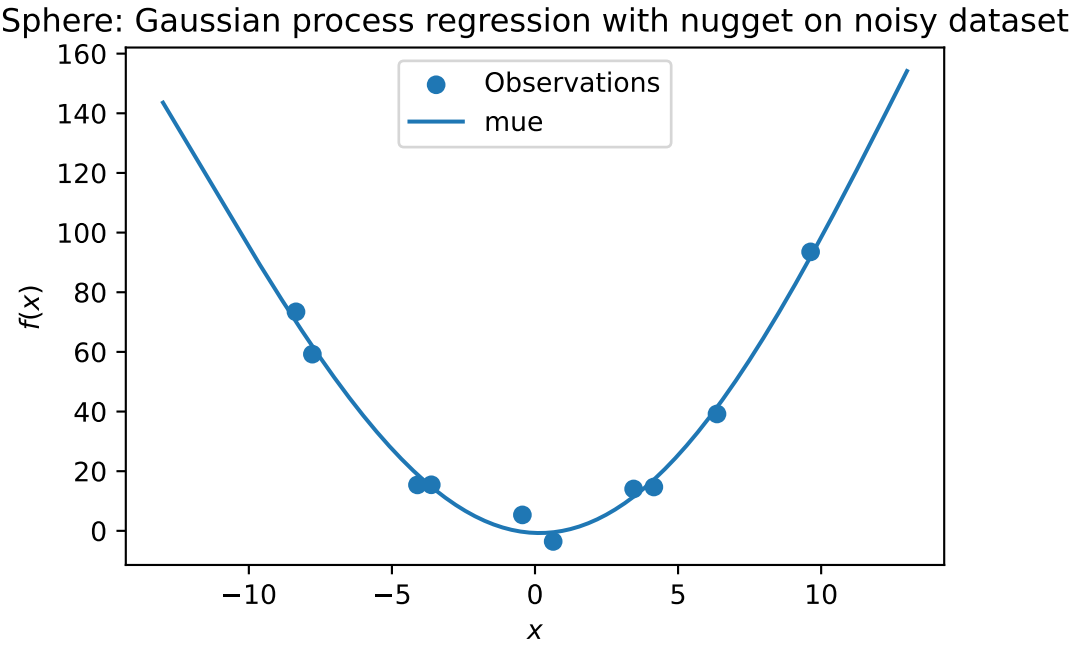
- A surrogate without nugget is fitted to these data:

```
S = Kriging(name='kriging',
            seed=123,
            log_level=50,
            n_theta=1,
            noise=False)
S.fit(X_train, y_train)

X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Sphere: Gaussian process regression on noisy dataset")
```



- In comparison to the surrogate without nugget, we fit a surrogate with nugget to the data:

```
S_nug = Kriging(name='kriging',
            seed=123,
            log_level=50,
            n_theta=1,
            noise=True)
S_nug.fit(X_train, y_train)
X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S_nug.predict(X_axis, return_val="all")
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Sphere: Gaussian process regression with nugget on noisy dataset")
```



- The value of the nugget term can be extracted from the model as follows:

```
S.Lambda
```

```
S_nug.Lambda
```

```
0.0005592460043643936
```

- We see:
  - the first model `S` has no nugget,
  - whereas the second model has a nugget value (`Lambda`) larger than zero.

13.5 Exercises

13.5.1 Noisy `fun_cubed`

- Analyse the effect of noise on the `fun_cubed` function with the following settings:

```
fun = analytical().fun_cubed
fun_control = fun_control_init(
```

```
    sigma=10,
    seed=123,)
lower = np.array([-10])
upper = np.array([10])
```

13.5.2 `fun_runge`

- Analyse the effect of noise on the `fun_runge` function with the following settings:

```
lower = np.array([-10])
upper = np.array([10])
fun = analytical().fun_runge
fun_control = fun_control_init(
    sigma=0.25,
    seed=123,)
```

13.5.3 `fun_forrester`

- Analyse the effect of noise on the `fun_forrester` function with the following settings:

```
lower = np.array([0])
upper = np.array([1])
fun = analytical().fun_forrester
fun_control = fun_control_init(
    sigma=5,
    seed=123,)
```

13.5.4 `fun_xsin`

- Analyse the effect of noise on the `fun_xsin` function with the following settings:

```
lower = np.array([-1.])
upper = np.array([1.])
fun = analytical().fun_xsin
fun_control = fun_control_init(
    sigma=0.5,
    seed=123,)
```

14 Optimal Computational Budget Allocation in Spot

This chapter demonstrates how noisy functions can be handled with Optimal Computational Budget Allocation (OCBA) by Spot.

14.1 Example: Spot, OCBA, and the Noisy Sphere Function

```
import numpy as np
from math import inf
from spotPython.fun.objectivefunctions import analytical
from spotPython.spot import spot
import matplotlib.pyplot as plt
from spotPython.utils.file import get_experiment_name
from spotPython.utils.init import fun_control_init
from spotPython.utils.file import get_spot_tensorboard_path

PREFIX = "09"
experiment_name = get_experiment_name(prefix=PREFIX)
print(experiment_name)
```

```
09_p040025_2023-12-30_22-59-41
```

14.1.1 The Objective Function: Noisy Sphere

The spotPython package provides several classes of objective functions. We will use an analytical objective function with noise, i.e., a function that can be described by a (closed) formula:

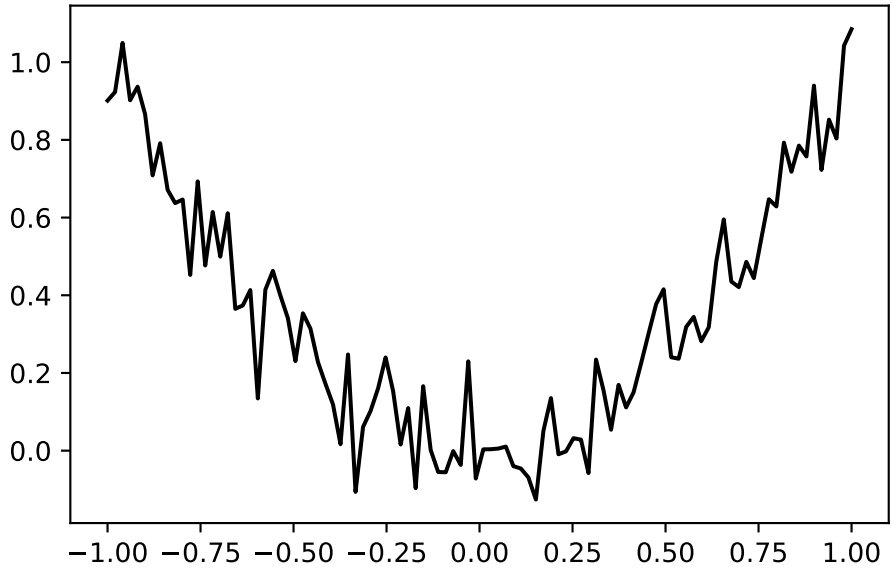
$$f(x) = x^2 + \epsilon$$

Since sigma is set to 0.1, noise is added to the function:

```
fun = analytical().fun_sphere
fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    sigma=0.1,
    seed=123,)
```

A plot illustrates the noise:

```
x = np.linspace(-1,1,100).reshape(-1,1)
y = fun(x, fun_control=fun_control)
plt.figure()
plt.plot(x,y, "k")
plt.show()
```



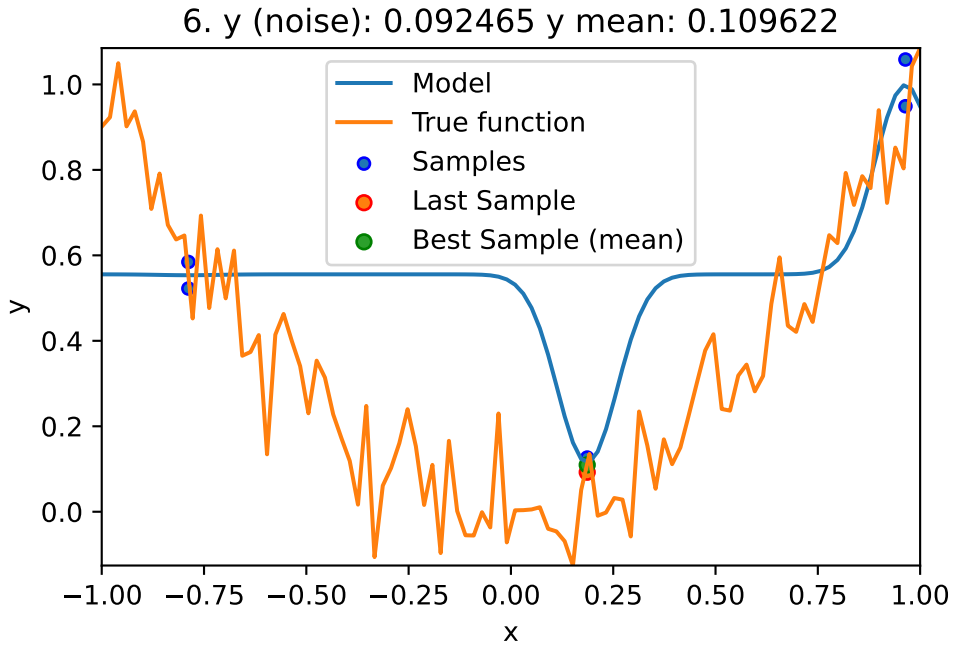
`Spot` is adopted as follows to cope with noisy functions:

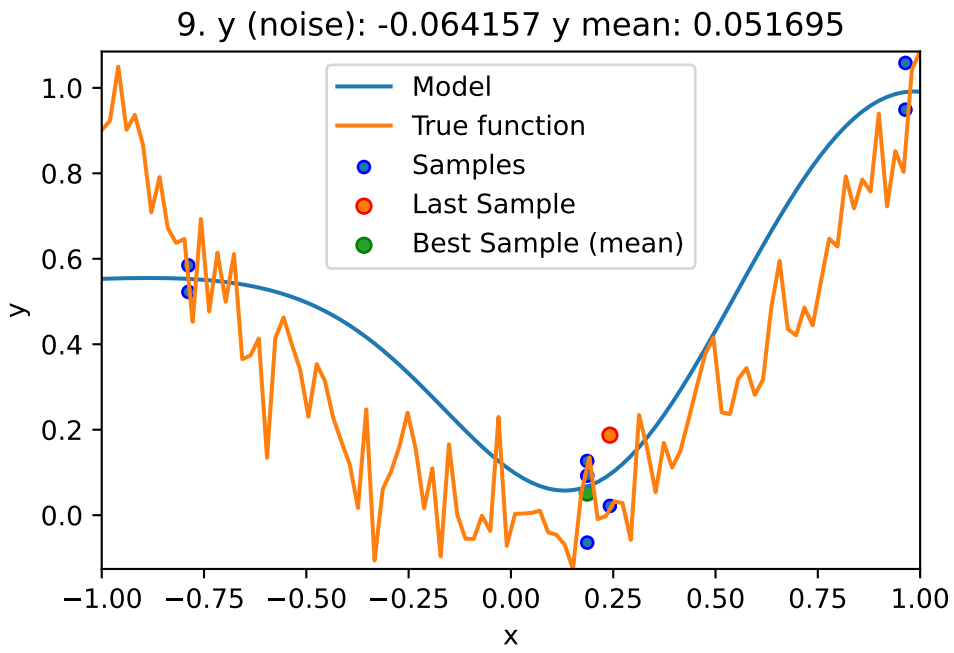
1. `fun_repeats` is set to a value larger than 1 (here: 2)
2. `noise` is set to `true`. Therefore, a nugget (`Lambda`) term is added to the correlation matrix
3. `init size` (of the `design_control` dictionary) is set to a value larger than 1 (here: 2)

```
spot_1_noisy = spot.Spot(fun=fun,
                    lower = np.array([-1]),
```

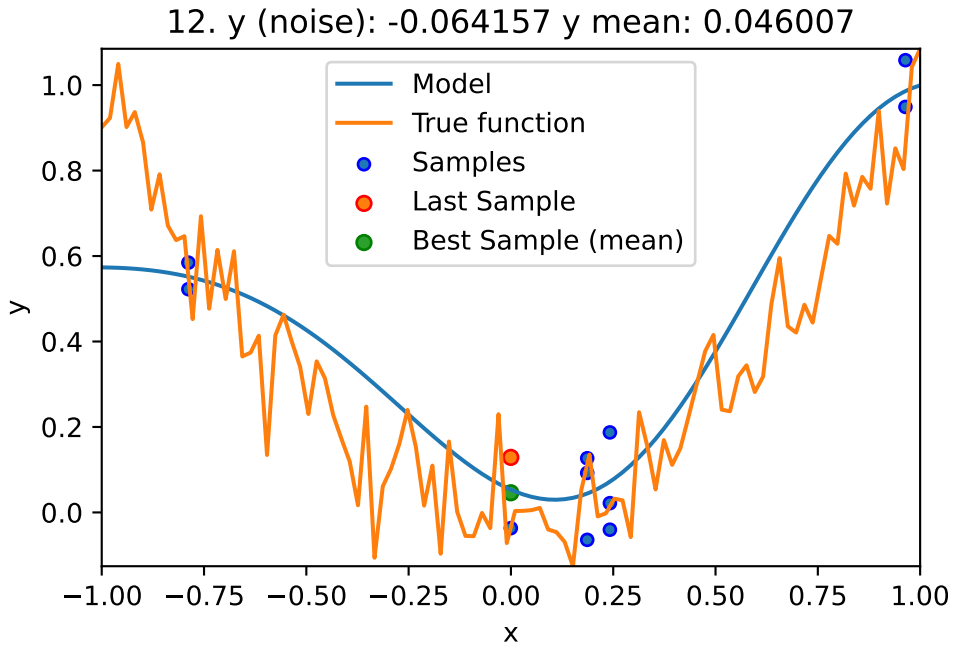
```
                    upper = np.array([1]),
                    fun_evals = 20,
                    fun_repeats = 2,
                    infill_criterion="ei",
                    noise = True,
                    tolerance_x=0.0,
                    ocba_delta = 1,
                    seed=123,
                    show_models=True,
                    fun_control = fun_control,
                    design_control={"init_size": 3,
                                    "repeats": 2},
                    surrogate_control={"noise": True})
```

```
spot_1_noisy.run()
```

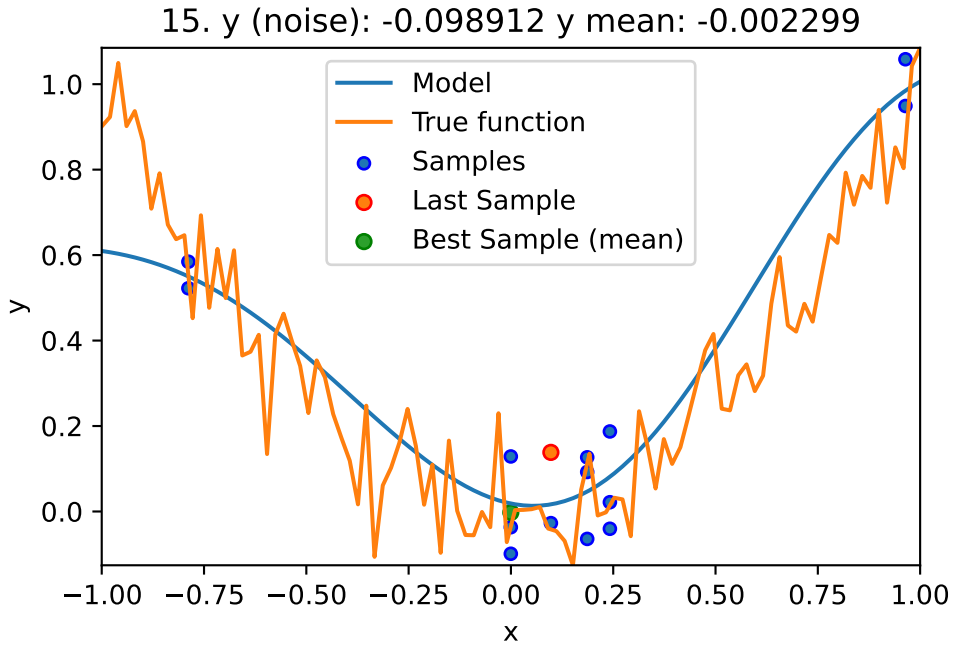




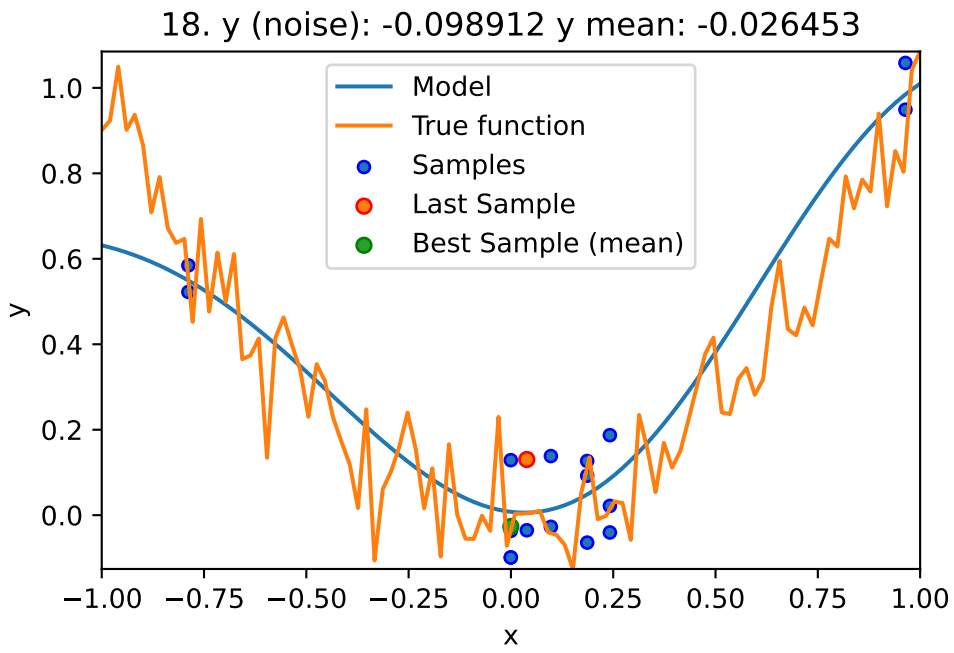

```
spotPython tuning: -0.0641572013655628 [####------] 45.00%
```



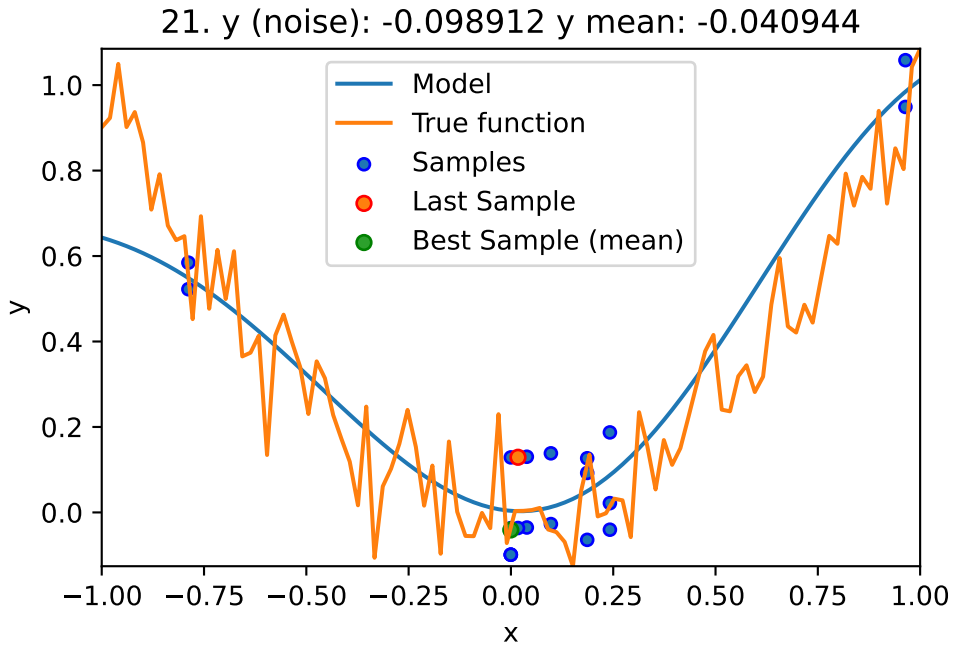
```
spotPython tuning: -0.0641572013655628 [######----] 60.00%
```




```
spotPython tuning: -0.09891205574531109 [########--] 75.00%
```




```
spotPython tuning: -0.09891205574531109 [#########-] 90.00%
```



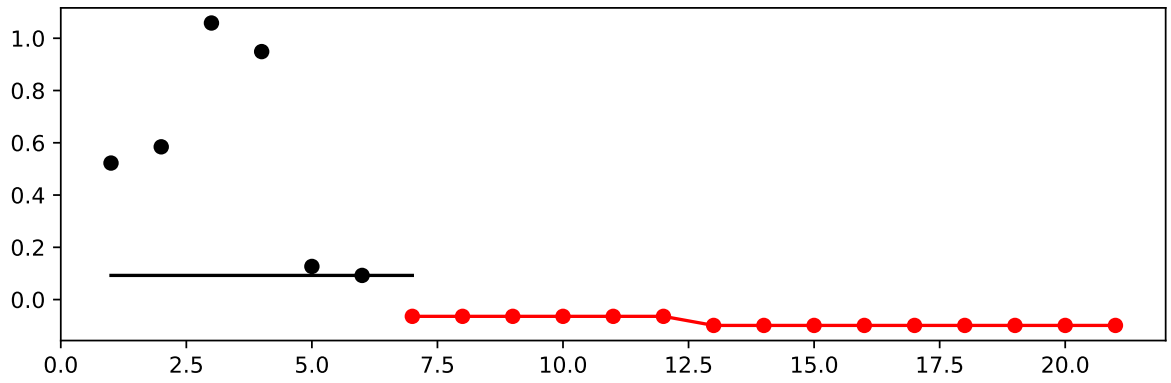
```
spotPython tuning: -0.09891205574531109 [##########] 100.00% Done...

<spotPython.spot.spot.Spot at 0x2b5725590>
```

14.2 Print the Results

```
spot_1_noisy.print_results()
```

```
min y: -0.09891205574531109
x0: -0.00028158386672407626
min mean y: -0.04094442953496975
x0: -0.00028158386672407626

[['x0', -0.00028158386672407626], ['x0', -0.00028158386672407626]]
```

```
spot_1_noisy.plot_progress(log_y=False)
```



14.3 Noise and Surrogates: The Nugget Effect

14.3.1 The Noisy Sphere

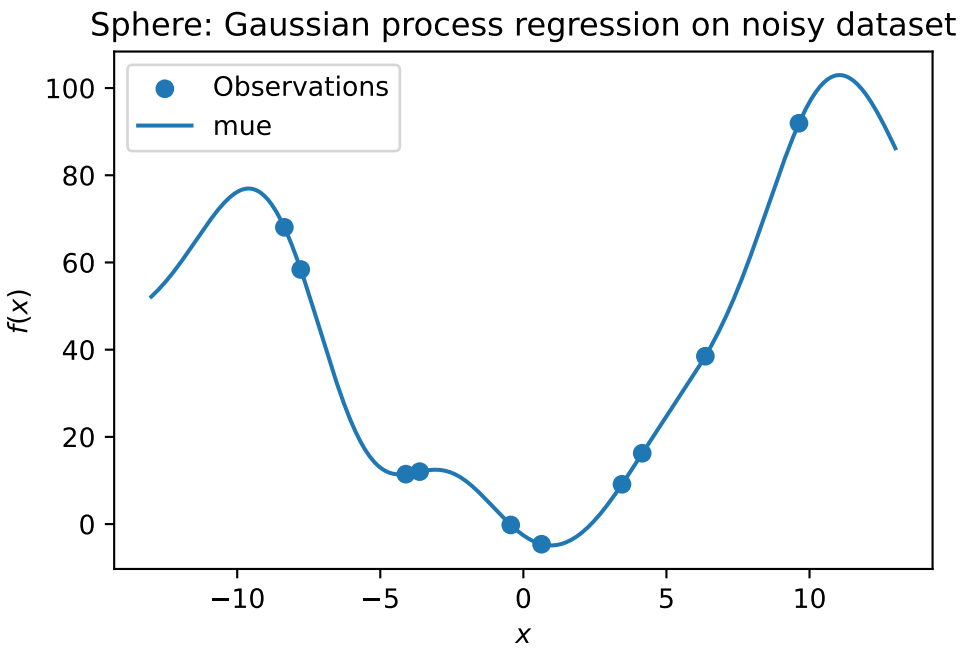
14.3.1.1 The Data

We prepare some data first:

```
import numpy as np
import spotPython
from spotPython.fun.objectivefunctions import analytical
from spotPython.spot import spot
from spotPython.design.spacefilling import spacefilling
from spotPython.build.kriging import Kriging
import matplotlib.pyplot as plt

gen = spacefilling(1)
rng = np.random.RandomState(1)
lower = np.array([-10])
upper = np.array([10])
fun = analytical().fun_sphere
fun_control = fun_control_init(
    sigma=2,
    seed=125)
X = gen.scipy_lhd(10, lower=lower, upper = upper)
y = fun(X, fun_control=fun_control)
X_train = X.reshape(-1,1)
y_train = y
```

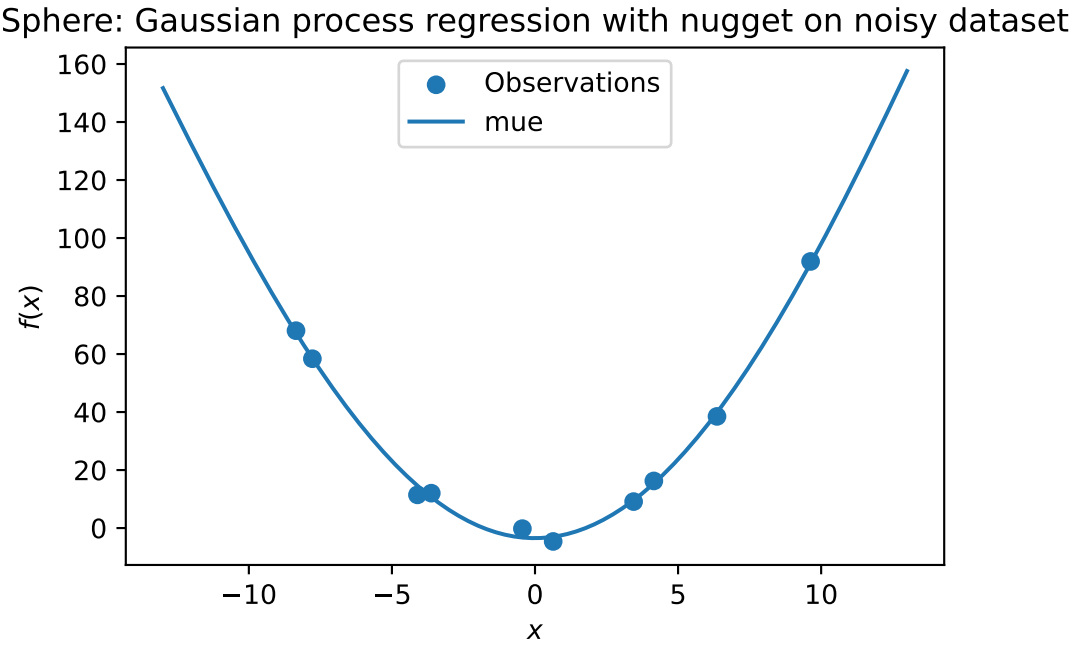
A surrogate without nugget is fitted to these data:

```
S = Kriging(name='kriging',
            seed=123,
            log_level=50,
            n_theta=1,
            noise=False)
S.fit(X_train, y_train)

X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Sphere: Gaussian process regression on noisy dataset")
```



In comparison to the surrogate without nugget, we fit a surrogate with nugget to the data:

```
S_nug = Kriging(name='kriging',
            seed=123,
            log_level=50,
            n_theta=1,
            noise=True)
S_nug.fit(X_train, y_train)
X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S_nug.predict(X_axis, return_val="all")
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Sphere: Gaussian process regression with nugget on noisy dataset")
```



The value of the nugget term can be extracted from the model as follows:

```
S.Lambda
```

```
S_nug.Lambda
```

```
8.374496269458742e-05
```

We see:

- the first model `S` has no nugget,
- whereas the second model has a nugget value (`Lambda`) larger than zero.

14.4 Exercises

14.4.1 Noisy `fun_cubed`

Analyse the effect of noise on the `fun_cubed` function with the following settings:

```
fun = analytical().fun_cubed
fun_control = fun_control_init(
    sigma=10,
    seed=123)
lower = np.array([-10])
upper = np.array([10])
```

14.4.2 `fun_runge`

Analyse the effect of noise on the `fun_runge` function with the following settings:

```
lower = np.array([-10])
upper = np.array([10])
fun = analytical().fun_runge
fun_control = fun_control_init(
    sigma=0.25,
    seed=123)
```

14.4.3 `fun_forrester`

Analyse the effect of noise on the `fun_forrester` function with the following settings:

```
lower = np.array([0])
upper = np.array([1])
fun = analytical().fun_forrester
fun_control = {"sigma": 5,
               "seed": 123}
```

14.4.4 `fun_xsin`

Analyse the effect of noise on the `fun_xsin` function with the following settings:

```
lower = np.array([-1.])
upper = np.array([1.])
fun = analytical().fun_xsin
fun_control = fun_control_init(
    sigma=0.5,
    seed=123)
```

A Introduction to Jupyter Notebook

Jupyter Notebook is a widely used tool in the Data Science community. It is easy to use and the produced code can be run per cell. This has a huge advantage, because with other tools e.g. (pycharm, vscode, etc.) the whole script is executed. This can be a time consuming process, especially when working with huge data sets.

A.1 Different Notebook cells

There are different cells that the notebook is currently supporting:

- code cells
- markdown cells
- raw cells

As a default, every cells in jupyter is set to "code"

A.1.1 Code cells

The code cells are used to execute the code. They are following the logic of the choosen kernel. Therefore, it is important to keep in mind which programming language is currently used. Otherwise one might yield an error because of the wrong syntax.

The code cells are executed my be **Run** button (can be found in the header of the notebook).

A.1.2 Markdown cells

The markdown cells are a usefull tool to comment the written code. Especially with the help of headers can the code be brought in a more readable format. If you are not familiar with the markdown syntax, you can find a usefull cheat sheet here: Markdown Cheat Sheeet

A.1.3 Raw cells

The "Raw NBConvert" cell type can be used to render different code formats into HTML or LaTeX by Sphinx. This information is stored in the notebook metadata and converted appropriately.

A.1.3.1 Usage

To select a desired format from within Jupyter, select the cell containing your special code and choose options from the following dropdown menus:

1. Select "Raw NBConvert"
2. Switch the Cell Toolbar to "Raw Cell Format" (The cell toolbar can be found under View)
3. Chose the appropriate "Raw NBConvert Format" within the cell

Data Science is fun

A.2 Install Packages

Because python is a heavily used programming language, there are many different packags that can make your life easier. Sadly, there are only a few standard packages that are already included in your python enviroment. If you have the need to install a new package in your enviroment, you can simply do that by exectuing the following code snippet in a **code cell**

```
!pip install numpy
```

- The *!* is used to run the cell as a shell command
- *pip* is package manager for python packages.
- *numpy* is the the package you want to install

Hint: It is often usefull to restart the kernel after installing a package, otherwise loading the package could lead to an error.

A.3 Load Packages

After successfully installing the package it is necessary to import them before you can work with them. The import of the packages is done in the following way:

```
import numpy as np
```

The imported packages are often abbreviated. This is because you need to specify where the function is coming from.

The most common abbreviations for data science packages are:

Table A.1: Abbreviations for data science packages

Abbreviation	Package	Import
np	numpy	import numpy as np
pd	pandas	import pandas as pd
plt	matplotlib	import matplotlib.pyplot as plt
px	plotly	import plotly.exprss as px
tf	tensorflow	import tensorflow as tf
sns	seaborn	import seaborn as sns
dt	datetime	import datetime as dt
pkl	pickle	import pickle as pkl

A.4 Functions in Python

Because python is not using Semicolon's it is import to keep track of indentation in your code. The indentation works as a placeholder for the semicolons. This is especially important if your are defining loops, functions, etc. …

Example: We are defining a function that calculates the squared sum of its input parameters

```
def squared_sum(x,y):
    z = x**2 + y**2
    return z
```

If you are working with something that needs indentation, it will be already done by the notebook.

Hint: Keep in mind that is good practice to use the *return* parameter. If you are not using *return* and a function has multiple paramaters that you would like to return, it will only return the last one defined.

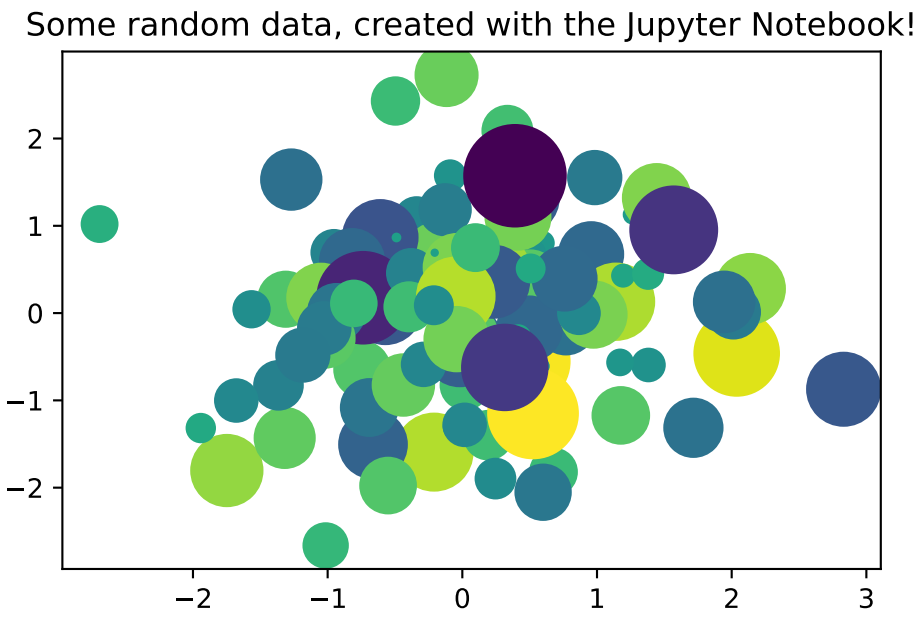
A.5 List of Useful Jupyter Notebook Shortcuts

Table A.2: List of useful Jupyter Notebook Shortcuts

Function	Keyboard Shortcut	Menu Tools
Save notebook	Esc + s	File → Save and Checkpoint
Create new Cell	Esc + a (above), Esc + b (below)	Insert → Cell above; Insert → Cell below
Run Cell	Ctrl + enter	Cell → Run Cell
Copy Cell	c	Copy Key
Paste Cell	v	Paste Key
Interrupt Kernel	Esc + i i	Kernel → Interrupt
Restart Kernel	Esc + 0 0	Kernel → Restart

If you combine everything you can create beautiful graphics

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

# Generate 100 random data points along 3 dimensions
x, y, scale = np.random.randn(3, 100)
fig, ax = plt.subplots()

# Map each onto a scatterplot we'll create with Matplotlib
ax.scatter(x=x, y=y, c=scale, s=np.abs(scale)*500)
ax.set(title="Some random data, created with the Jupyter Notebook!")
plt.show()
```


Some random data, created with the Jupyter Notebook!
2
1
0
−1
−2
−2
−1
0
1
2
3

B Git Introduction

B.1 Learning Objectives

In this learning unit, you will learn how to set up Git as a version control system for a project. The most important Git commands will be explained. You will learn how to track and manage changes to your projects with Git. Specifically:

- Initializing a repository: `git init`
- Ignoring files: `.gitignore`
- Adding files to the staging area: `git add`
- Checking status changes: `git status`
- Reviewing history: `git log`
- Creating a new branch: `git branch`
- Switching to the current branch: `git switch` and `git checkout`
- Merging two branches: `git merge`
- Resolving conflicts
- Reverting changes: `git revert`
- Uploading changes to GitLab: `git push`
- Downloading changes from GitLab: `git pull`
- Advanced: `git rebase`

B.2 Basics of Git

B.2.1 Initializing a Repository: `git init`

To set up Git as a version control system for your project, you need to initialize a new Git repository at the top-level folder, which is the working directory of your project. This is done using the `git init` command.

All files in this folder and its subfolders will automatically become part of the repository. Creating a Git repository is similar to adding an all-powerful passive observer of all things to your project. Git sits there, observes, and takes note of even the smallest changes, such as a single character in a file within a repository with hundreds of files. And it will tell you where these changes occurred if you forget. Once Git is initialized, it monitors all changes made

within the working directory, and it tracks the history of events from that point forward. For this purpose, a historical timeline is created for your project, referred to as a "branch," and the initial branch is named `main`. So, when someone says they are on the `main branch` or working on the `main branch`, it means they are in the historical main timeline of the project. The Git repository, often abbreviated as `repo`, is a virtual representation of your project, including its history and branches, a book, if you will, where you can look up and retrieve the entire history of the project: you work in your working directory, and the Git repository tracks and stores your work.

B.2.2 Ignoring Files: `.gitignore`

It's useful that Git watches and keeps an eye on everything in your project. However, in most projects, there are files and folders that you don't need or want to keep an eye on. These may include system files, local project settings, libraries with dependencies, and so on.

You can exclude any file or folder from your Git repository by including them in the `.gitignore` file. In the `.gitignore` file, you create a list of file names, folder names, and other items that Git should not track, and Git will ignore these items. Hence the name "gitignore." Do you want to track a file that you previously ignored? Simply remove the mention of the file in the gitignore file, and Git will start tracking it again.

B.2.3 Adding Changes to the Staging Area: `git add`

The interesting thing about Git as an all-powerful, passive observer of all things is that it's very passive. As long as you don't tell Git what to remember, it will passively observe the changes in the project folder but do nothing.

When you make a change to your project that you want Git to include in the project's history to take a snapshot of so you can refer back to it later, your personal checkpoint, if you will, you need to first stage the changes in the staging area. What is the staging area? The staging area is where you collect changes to files that you want to include in the project's history.

This is done using the `git add` command. You can specify which files you want to add by naming them, or you can add all of them using `-A`. By doing this, you're telling Git that you've made changes and want it to remember these particular changes so you can recall them later if needed. This is important because you can choose which changes you want to stage, and those are the changes that will eventually be transferred to the history.

Note: When you run `git add`, the changes are not transferred to the project's history. They are only transferred to the staging area.

Example of git add from the beginning

```
# Create a new directory for your
# repository and navigate to that directory:

mkdir my-repo
cd my-repo

# Initialize the repository with git init:

git init

# Create a .gitignore file for Python code.
# You can use a template from GitHub:

curl https://raw.githubusercontent.com/github/gitignore/master/Python.gitignore -o .giti

# Add your files to the repository using git add:

git add .
```

This adds all files in the current directory to the repository, except for the files listed in the .gitignore file.

B.2.4 Transferring Changes to Memory: `git commit`

The power of Git becomes evident when you start transferring changes to the project history. This is done using the `git commit` command. When you run `git commit`, you inform Git that the changes in the staging area should be added to the history of the project so that they can be referenced or retrieved later.

Additionally, you can add a commit message with the `-m` option to explain what changes were made. So when you look back at the project history, you can see that you added a new feature.

`git commit` creates a snapshot, an image of the current state of your project at that specific time, and adds it to the branch you are currently working on.

As you work on your project and transfer more snapshots, the branch grows and forms a timeline of events. This means you can now look back at every transfer in the branch and see what your code looked like at that time.

You can compare any phase of your code with any other phase of your code to find errors, restore deleted code, or do things that would otherwise not be possible, such as resetting the project to a previous state or creating a new timeline from any point.

So how often should you add these commits? My rule of thumb is not to commit too often. It's better to have a Git repository with too many commits than one with too few commits.

> ℹ Continuing the example from above:
>
> After adding your files with `git add`, you can create a commit to save your changes. Use the `git commit` command with the `-m` option to specify your commit message:
>
> ```
> git commit -m "My first commit message"
> ```
>
> This creates a new commit with the added files and the specified commit message.

B.2.5 Check the Status of Your Repository: `git status`

If you're wondering what you've changed in your project since the last commit snapshot, you can always check the Git status. Git will list every modified file and the current status of each file.

This status can be either:

- Unchanged (`unmodified`), meaning nothing has changed since you last transferred it, or
- It's been changed (`changed`) but not staged (`staged`) to be transferred into the history, or
- Something has been added to staging (`staged`) and is ready to be transferred into the history.

When you run `git status`, you get an overview of the current state of your project.

> ℹ Continuing the example from above:
>
> The `git status` command displays the status of your working directory and the staging area. It shows you which files have been modified, which files are staged for commit, and which files are not yet being tracked:
>
> ```
> git status
> ```
>
> `git status` is a useful tool to keep track of your changes and ensure that you have added all the desired files for commit.

B.2.6 Review Your Repository's History: `git log`

> Continuing the example from above:
>
> You can view the history of your commits with the `git log` command. This command displays a list of all the commits in the current branch, along with information such as the author, date, and commit message:
>
> ```
> git log
> ```
>
> There are many options to customize the output of `git log`. For example, you can use the `--pretty` option to change the format of the output:
>
> ```
> git log --pretty=oneline
> ```
>
> This displays each commit in a single line.

B.3 Branches (Timelines)

B.3.1 Creating an Alternative Timeline: `git branch`

In the course of developing a project, you often reach a point where you want to add a new feature, but doing so might require changing the existing code in a way that could be challenging to undo later.

Or maybe you just want to experiment and be able to discard your work if the experiment fails. In such cases, Git allows you to create an alternative timeline called a `branch` to work in.

This new `branch` has its own name and exists in parallel with the `main branch` and all other branches in your project.

During development, you can switch between branches and work on different versions of your code concurrently. This way, you can have a stable codebase in the `main branch` while developing an experimental feature in a separate `branch`. When you switch from one `branch` to another, the code you're working on is automatically reset to the latest commit of the branch you're currently in.

If you're working in a team, different team members can work on their own branches, creating an entire universe of alternative timelines for your project. When features are completed, they can be seamlessly merged back into the `main branch`.

Continuing the example from above:

To create a new `branch`, you can use the `git branch` command with the name of the new `branch` as an argument:

```
git branch my-tests
```

B.3.2 The Pointer to the Current Branch: `HEAD`

How does Git know where you are on the timeline, and how can you keep track of your position?

You're always working at the tip (`HEAD`) of the currently active branch. The `HEAD` pointer points there quite literally. In a new project archive with just a single `main branch` and only new commits being added, `HEAD` always points to the latest commit in the `main branch`. That's where you are.

However, if you're in a repository with multiple branches, meaning multiple alternative timelines, `HEAD` will point to the latest commit in the branch you're currently working on.

B.3.3 Switching to an Alternative Timeline: `git switch`

As your project grows, and you have multiple branches, you need to be able to switch between these branches. This is where the `switch` command comes into play.

At any time, you can use the `git switch` command with the name of the branch you want to switch to, and `HEAD` moves from your current branch to the one you specified.

If you've made changes to your code before switching, Git will attempt to carry those changes over to the branch you're switching to. However, if these changes conflict with the target branch, the switch will be canceled.

To resolve this issue without losing your changes, return to the original branch, add and commit your recent changes, and then perform the `switch`.

B.3.4 Switching to an Alternative Timeline and Making Changes: `git checkout`

To switch between branches, you can also use the `git checkout` command. It works similarly to `git switch` for this purpose: you pass the name of the branch you want to switch to, and `HEAD` moves to the beginning of that branch.

But `checkout` can do more than just switch to another timeline. With `git checkout`, you can also move to any commit point in any timeline. In other words, you can travel back in time and work on code from the past.

To do this, use `git checkout` and provide the commit ID. This is an automatically generated, random combination of letters and numbers that identifies each commit. You can retrieve the commit ID using `git log`. When you run `git log`, you get a list of all the commits in your repository, starting with the most recent ones.

When you use `git checkout` with an older commit ID, you check out a commit in the middle of a branch. This disrupts the timeline, as you're actively attempting to change history. Git doesn't want you to do that because, much like in a science fiction movie, altering the past might also alter the future. In our case, it would break the version control branch's coherence.

To prevent you from accidentally disrupting time and altering history, checking out an earlier commit in any branch results in the warning "Detached Head," which sounds rather ominous. The "Detached Head" warning is appropriate because it accurately describes what's happening. Git literally detaches the head from the branch and sets it aside.

Now, you're working outside of time in a space unbound to any timeline, which again sounds rather threatening but is perfectly fine in reality.

To continue working on this past code, all you need to do is reattach it to the timeline. You can use `git branch` to create a new branch, and the detached head will automatically attach to this new branch.

Instead of breaking the history, you've now created a new alternative timeline that starts in the past, allowing you to work safely. You can continue working on the branch as usual.

> Continuing the example from above:
>
> To switch to a new branch, you can use the `git checkout` command:
>
> ```
> git checkout meine-tests
> ```
>
> Now you're using the new branch and can make changes independently from the original branch.

B.3.5 The Difference Between `checkout` and `switch`

What is the difference between `git switch` and `git checkout`? `git switch` and `git checkout` are two different commands that both serve the purpose of switching between branches. You can use both to switch between branches, but they have an important distinction. `git switch` is a new command introduced with Git 2.23. `git checkout` is an older command that has existed since Git 1.6.0. So, `git switch` and `git checkout` have

different origins. `git switch` was introduced to separate the purposes of `git checkout`. `git checkout` has two different purposes: 1. It can be used to switch between branches, and 2. It can be used to reset files to the state of the last commit.

Here's an example: In my project, I made a change since the last commit, but I haven't staged it yet. Then, I realized that I actually don't want this change. I want to reset the file to the state before the last commit. As long as I haven't committed my changes, I can do this with `git checkout` by targeting the specific file. So, if that file is named `main.js`, I can say: `git checkout main.js`. And the file will be reset to the state of the last commit, which makes sense. I'm checking out the file from the last commit.

But that's quite different from switching between the beginning of one branch to another. `git switch` and `git restore` were introduced to separate these two operations. `git switch` is for switching between branches, and `git restore` is for resetting the specified file to the state of the last commit. If you try to restore a file with `git switch`, it simply won't work. It's not intended for that. As I mentioned earlier, it's about separating concerns.

:::{.callout-note} #### Difference Between `checkout` and `switch` `git checkout` and `git switch` are both commands for switching between branches in a Git repository. The main difference between the two commands is that `git switch` is a newer command specifically designed for branch switching, while `git checkout` is an older command that can be used for various tasks, including branch switching.

Here's an example demonstrating how to initialize a repository and switch between branches:

```
# Create a new directory for your repository
# and navigate to that directory:
mkdir my-repo
cd my-repo

# Initialize the repository with git init:
git init

# Create a new branch with git branch:
git branch my-new-branch

# Switch to the new branch using git switch:
git switch my-new-branch

# Alternatively, you can also use git checkout
# to switch to the new branch:

git checkout my-new-branch
```

Both commands lead to the same result: You are now on the new branch.

B.4 Merging Branches and Resolving Conflicts

B.4.1 git `merge`: Merging Two Timelines

Git allows you to split your development work into as many branches or alternative timelines as you like, enabling you to work on many different versions of your code simultaneously without losing or overwriting any of your work.

This is all well and good, but at some point, you need to bring those various versions of your code back together into one branch. That's where `git merge` comes in.

Consider an example where you have two branches, a `main branch` and an experimental branch called `experimental-branch`. In the experimental branch, there is a new feature. To merge these two branches, you set `HEAD` to the branch where you want to incorporate the code and execute `git merge` followed by the name of the branch you want to merge. `HEAD` is a special pointer that points to the current branch. When you run `git merge`, it combines the code from the branch associated with `HEAD` with the code from the branch specified by the branch name you provide.

```
# Initialize the repository
git init

# Create a new branch called "experimental-branch"
git branch experimental-branch

# Switch to the "experimental-branch"
git checkout experimental-branch

# Add the new feature here and
# make a commit
# ...

# Switch back to the "main" branch
git checkout main

# Perform the merge
git merge experimental-branch
```

During the merge, matching pieces of code in the branches overlap, and any new code from the branch being merged is added to the project. So now, the main branch also contains the code from the experimental branch, and the events of the two separate timelines have been merged into a single one. What's interesting is that even though the experimental branch was merged

with the main branch, the last commit of the experimental branch remains intact, allowing you to continue working on the experimental branch separately if you wish.

B.4.2 Resolving Conflicts When Merging

Merging branches where there are no code changes at the same place in both branches is a straightforward process. It's also a rare process. In most cases, there will be some form of conflict between the branches – the same code or the same code area has been modified differently in the different branches. Merging two branches with such conflicts will not work, at least not automatically.

In this case, Git doesn't know how to merge this code. So, when such a situation occurs, it's marked as a conflict, and the merging process is halted. This might sound more dramatic than it is. When you get a conflict warning, Git is saying there are two different versions here, and Git needs to know which one you want to keep. To help you figure out the conflict, Git combines all the code into a single file and automatically marks the conflicting code as the current change, which is the original code from the branch you're working on, or as the incoming change, which is the code from the file you're trying to merge.

To resolve this conflict, you'll edit the file to literally resolve the code conflict. This might mean accepting either the current or incoming change and discarding the other. It could mean combining both changes or something else entirely. It's up to you. So, you edit the code to resolve the conflict. Once you've resolved the conflict by editing the code, you add the new conflict-free version to the staging area with `git add` and then commit the merged code with `git commit`. That's how the conflict is resolved.

A merge conflict occurs when Git struggles to automatically merge changes from two different branches. This usually happens when changes were made to the same line in the same file in both branches. To resolve a merge conflict, you must manually edit the affected files and choose the desired changes. Git marks the conflict areas in the file with special markings like `<<<<<<<`, `=======`, and `>>>>>>>`. You can search for these markings and manually select the desired changes. After resolving the conflicts, you can add the changes with `git add` and create a new commit with `git commit` to complete the merge.

Here's an example:

```
# Perform the merge (this will cause a conflict)
git merge experimenteller-branch

# Open the affected file in an editor and manually resolve the conflicts
# ...

# Add the modified file
```

```
git add <filename>

# Create a new commit
git commit -m "Resolved conflicts"
```

B.4.3 git `revert`: Undoing Something

One of the most powerful features of any software tool is the "Undo" button. Make a mistake, press "Undo," and it's as if it never happened. However, that's not quite as simple when an all-powerful, passive observer is watching and recording your project's history. How do you undo something that you've added to the history without rewriting the history?

The answer is that you can overwrite the history with the `git reset` command, but that's quite risky and not a good practice.

A better solution is to work with the historical timeline and simply place an older version of your code at the top of the branch. This is done with `git revert`. To make this work, you need to know the commit ID of the commit you want to go back to.

The commit ID is a machine-generated set of random numbers and letters, also known as a hash. To get a list of all the commits in the repository, including the commit ID and commit message, you can run `git log`.

```
# Show the list of all operations in the repository
git log
```

By the way, it's a good idea to leave clear and informative commit messages for this reason. This way, you know what happened in your previous commits. Once you've found the commit you want to revert to, call that commit ID with `git revert`, and then the ID. This will create a new commit at the top of the branch with the code from the reference commit. To transfer the code to the branch, add a commit message and save it. Now, the last commit in your branch matches the commit you're reverting to, and your project's history remains intact.

An example with `git revert`

```
# Initialize a new repository
git init

# Create a new file
echo "Hello, World" > file.txt

# Add the file to the repository
git add file.txt

# Create a new commit
git commit -m "First commit"

# Modify the file
echo "Goodbye, World" > file.txt

# Add the modified file
git add file.txt

# Create a new commit
git commit -m "Second commit"

# Use git log to find the commit ID of the second commit
git log

# Use git revert to undo the changes from the second commit
git revert <commit-id>
```

To download the `students` branch from the repository `git@git-ce.rwth-aachen.de:spotseven-lab/numerisc` to your local machine, add a file, and upload the changes, you can follow these steps:

An example with `git clone`, `git checkout`, `git add`, `git commit`, `git push`

```
# Clone the repository to your local machine:
git clone git@git-ce.rwth-aachen.de:spotseven-lab/numerische-mathematik-sommersemester20

# Change to the cloned repository:
cd numerische-mathematik-sommersemester2023

# Switch to the students branch:
git checkout students

# Create the Test folder if it doesn't exist:
mkdir Test

# Create the Testdatei.txt file in the Test folder:
touch Test/Testdatei.txt

# Add the file with git add:
git add Test/Testdatei.txt

# Commit the changes with git commit:
git commit -m "Added Testdatei.txt"

# Push the changes with git push:
git push origin students
```

This will upload the changes to the server and update the students branch in the repository.

B.5 Downloading from GitLab

To download changes from a GitLab repository to your local machine, you can use the `git pull` command. This command downloads the latest changes from the specified remote repository and merges them with your local repository.

Here is an example:

An example with `git pull`

```
# Navigate to the local repository
# linked to the GitHub repository:
cd my-local-repository

# Make sure you are in the correct branch:
git checkout main

# Download the latest changes from GitHub:
git pull origin main
```

This downloads the latest changes from the main branch of the remote repository named "origin" and merges them with your local repository.

If there are conflicts between the downloaded changes and your local changes, you will need to resolve them manually before proceeding.

B.6 Advanced

B.6.1 git `rebase`: Moving the Base of a Branch

In some cases, you may need to "rewrite history." A common scenario is that you've been working on a new feature in a feature branch, and you realize that the work should have actually happened in the `main branch`.

To resolve this issue and make it appear as if the work occurred in the `main branch`, you can reset the experimental branch. "Rebase" literally means detaching the base of the experimental branch and moving it to the beginning of another branch, giving the branch a new base, thus "rebasing."

This operation is performed from the branch you want to "rebase." You use `git rebase` and specify the branch you want to use as the new base. If there are no conflicts between the experimental branch and the branch you want to rebase onto, this process happens automatically.

If there are conflicts, Git will guide you through the conflict resolution process for each commit from the rebase branch.

This may sound like a lot, but there's a good reason for it. You are literally rewriting history by transferring commits from one branch to another. To maintain the coherence of the new version history, there should be no conflicts within the commits. So, you need to resolve

them one by one until the history is clean. It goes without saying that this can be a fairly labor-intensive process. Therefore, you should not use `git rebase` frequently.

An example with `git rebase`

`git rebase` is a command used to change the base of a branch. This means that commits from the branch are applied to a new base, which is usually another branch. It can be used to clean up the repository history and avoid merge conflicts.
Here is an example showing how to use `git rebase`:

- In this example, we initialize a new Git repository and create a new file. We add the file to the repository and make an initial commit. Then, we create a new branch called "feature" and switch to that branch. We make changes to the file in the feature branch and create a new commit.
- Then, we switch back to the main branch and make changes to the file again. We add the modified file and make another commit.
- To rebase the feature branch onto the main branch, we first switch to the feature branch and then use the `git rebase` command with the name of the main branch as an argument. This applies the commits from the feature branch to the main branch and changes the base of the feature branch.

```
# Initialize a new repository
git init
# Create a new file
echo "Hello World" > file.txt
# Add the file to the repository
git add file.txt
# Create an initial commit
git commit -m "Initial commit"
# Create a new branch called "feature"
git branch feature
# Switch to the "feature" branch
git checkout feature
# Make changes to the file in the "feature" branch
echo "Hello Feature World" > file.txt
# Add the modified file
git add file.txt
# Create a new commit in the "feature" branch
git commit -m "Feature commit"
# Switch back to the "main" branch
git checkout main
# Make changes to the file in the "main" branch
echo "Hello Main World" > file.txt
# Add the modified file
git add file.txt
# Create a new commit in the "main" branch
git commit -m "Main commit"
# Use git rebase to rebase the "feature" branch
# onto the "main" branch
git checkout feature
git rebase main
```

B.7 Exercises

In order to be able to carry out this exercise, we provide you with a functional working environment. This can be accessed here. You can log in using your GMID. If you do not have one, you can generate one here. Once you have successfully logged in to the server, you must open a terminal instance. You are now in a position to carry out the exercise.

Alternatively, you can also carry out the exercise locally on your computer, but then you will need to install git.

B.7.1 Create project folder

First create the `test-repo` folder via the command line and then navigate to this folder using the corresponding command.

B.8 Initialize repo

Now initialize the repository so that the future project, which will be saved in the `test-repo` folder, and all associated files are versioned.

B.8.1 Do not upload / ignore certain file types

In order to carry out this exercise, you must first download a file which you then have git ignore. To do this, download the current examination regulations for the Bachelor's degree program in Electrical Engineering using the following command `curl -o pruefungsordnung.pdf https://www.th-koeln.de/mam/downloads/deutsch/studium/studiengaenge/f07/ordnungen_plaene/f07_`

The PDF file has been stored in the root directory of your repo and you must now exclude it from being uploaded so that no changes to this file are tracked. Please note that not only this one PDF file should be ignored, but all PDF files in the repo.

B.8.2 Create file and stage it

In order to be able to commit a change later and thus make it traceable, it must first be staged. However, as we only have a PDF file so far, which is to be ignored by git, we cannot stage anything. Therefore, in this task, a file `test.txt` with some string as content is to be created and then staged.

B.8.3 Create another file and check status

To understand the status function, you should create the file `test2.txt` and then call the status function of git.

B.8.4 Commit changes

After the changes to the `test.txt` file have been staged and these are now to be transferred to the project process, they must be committed. Therefore, in this step you should perform a corresponding commit in the current branch with the message `test-commit`. Finally, you should also display the history of the commits.

B.8.5 Create a new branch and switch to it

In this task, you are to create a new branch with the name `change-text` in which you will later make changes. You should then switch to this branch.

B.8.6 Commit changes in the new branch

To be able to merge the new branch into the main branch later, you must first make changes to the `test.txt` file. To do this, open the file and simply change the character string in this file before saving the changes and closing the file. Before you now commit the file, you should reset the file to the status of the last commit for practice purposes and thus undo the change. After you have done this, open the file `test.txt` again and change the character string again before saving and closing the file. This time you should commit the file `test.txt` and then commit it with the message `test-commit2`.

B.8.7 Merge branch into main

After you have committed the change to the `test.txt` file, you should merge the `change-text` branch including the change into the main branch so that it is also available there.

B.8.8 Resolve merge conflict

To simulate a merge conflict, you must first change the content of the `test.txt` file before you commit the change. Then switch to the branch `change-text` and change the file `test.txt` there as well before you commit the change. Now you should try to merge the branch `change-text` into the main branch and solve the problems that occur in order to be able to perform the merge successfully.

C Python Introduction

C.1 Recommendations

Beginner's Guide to Python

D Documentation of the Sequential Parameter Optimization

This document describes the `Spot` features. The official `spotPython` documentation can be found here: https://sequential-parameter-optimization.github.io/spotPython/.

D.1 Example: spot

```
import numpy as np
from math import inf
from spotPython.fun.objectivefunctions import analytical
from spotPython.spot import spot
from scipy.optimize import shgo
from scipy.optimize import direct
from scipy.optimize import differential_evolution
import matplotlib.pyplot as plt
```

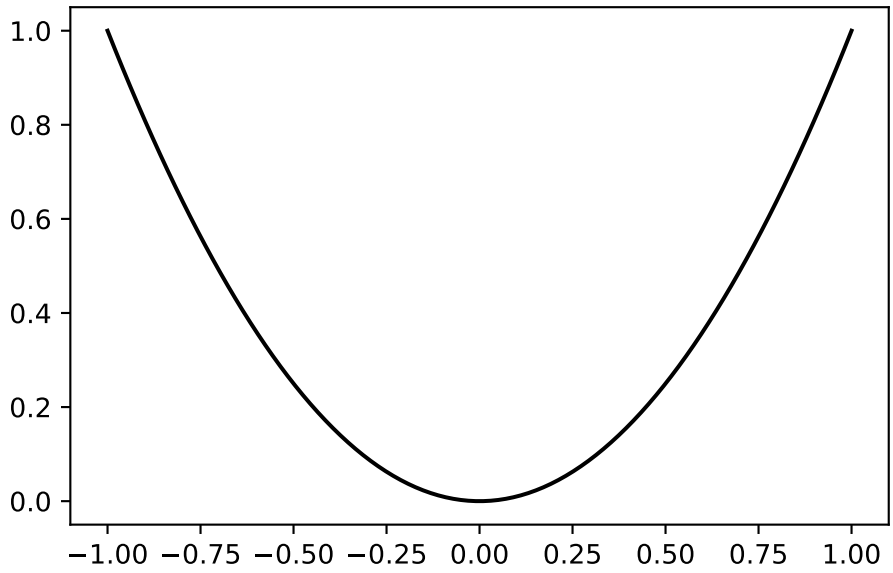
D.1.1 The Objective Function

The `spotPython` package provides several classes of objective functions. We will use an analytical objective function, i.e., a function that can be described by a (closed) formula:

$$f(x) = x^2$$

```
fun = analytical().fun_sphere
```

```
x = np.linspace(-1,1,100).reshape(-1,1)
y = fun(x)
plt.figure()
plt.plot(x,y, "k")
plt.show()
```




```
spot_1 = spot.Spot(fun=fun,
                   lower = np.array([-10]),
                   upper = np.array([100]),
                   fun_evals = 7,
                   fun_repeats = 1,
                   max_time = inf,
                   noise = False,
                   tolerance_x = np.sqrt(np.spacing(1)),
                   var_type=["num"],
                   infill_criterion = "y",
                   n_points = 1,
                   seed=123,
                   log_level = 50,
                   show_models=True,
                   fun_control = {},
                   design_control={"init_size": 5,
                                   "repeats": 1},
                   surrogate_control={"noise": False,
                                      "cod_type": "norm",
                                      "min_theta": -4,
                                      "max_theta": 3,
                                      "n_theta": 1,
                                      "model_optimizer": differential_evolution,
                                      "model_fun_evals": 1000,
```

```
                                                    })
```

`spot`'s `__init__` method sets the control parameters. There are two parameter groups:

1. external parameters can be specified by the user
2. internal parameters, which are handled by `spot`.

D.1.2 External Parameters

external parameter	type	description	default	mandatory
`fun`	object	objective function		yes
`lower`	array	lower bound		yes
`upper`	array	upper bound		yes
`fun_evals`	int	number of function evaluations	15	no
`fun_evals`	int	number of function evaluations	15	no
`fun_control`	dict	noise etc.	{}	n
`max_time`	int	max run time budget	`inf`	no
`noise`	bool	if repeated evaluations of `fun` results in different values, then `noise` should be set to `True`.	`False`	no

external parameter	type	description	default	mandatory
`tolerance_x`	float	tolerance for new x solutions. Minimum distance of new solutions, generated by `suggest_new_X`, to already existing solutions. If zero (which is the default), every new solution is accepted.	`0`	no
`var_type`	list	list of type information, can be either `"num"` or `"factor"`	`["num"]`	no
`infill_criterion`	string	Can be `"y"`, `"s"`, `"ei"` (negative expected improvement), or `"all"`	`"y"`	no
`n_points`	int	number of infill points	1	no
`seed`	int	initial seed. If `Spot.run()` is called twice, different results will be generated. To reproduce results, the `seed` can be used.	`123`	no

external parameter	type	description	default	mandatory
`log_level`	int	log level with the following settings: `NOTSET` (`0`), `DEBUG` (`10`: Detailed information, typically of interest only when diagnosing problems.), `INFO` (`20`: Confirmation that things are working as expected.), `WARNING` (`30`: An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.), `ERROR` (`40`: Due to a more serious problem, the software has not been able to perform some function.), and `CRITICAL` (`50`: A serious error, indicating that the program itself may be unable to continue running.)	`50`	no

external parameter	type	description	default	mandatory
`show_models`	bool	Plot model. Currently only 1-dim functions are supported	`False`	no
`design`	object	experimental design	`None`	no
`design_control`	dict	control parameters	see below	no
`surrogate`		surrogate model	`kriging`	no
`surrogate_control`	dict	control parameters	see below	no
`optimizer`	object	optimizer	see below	no
`optimizer_control`	dict	control parameters	see below	no

- Besides these single parameters, the following parameter dictionaries can be specified by the user:
  - `fun_control`
  - `design_control`
  - `surrogate_control`
  - `optimizer_control`

D.2 The `fun_control` Dictionary

external parameter	type	description	default	mandatory
`sigma`	float	noise: standard deviation	`0`	yes
`seed`	int	seed for rng	`124`	yes

D.3 The `design_control` Dictionary

external parameter	type	description	default	mandatory
`init_size`	int	initial sample size	`10`	yes

external parameter	type	description	default	mandatory
`repeats`	int	number of repeats of the initial sammples	`1`	yes

D.4 The `surrogate_control` Dictionary

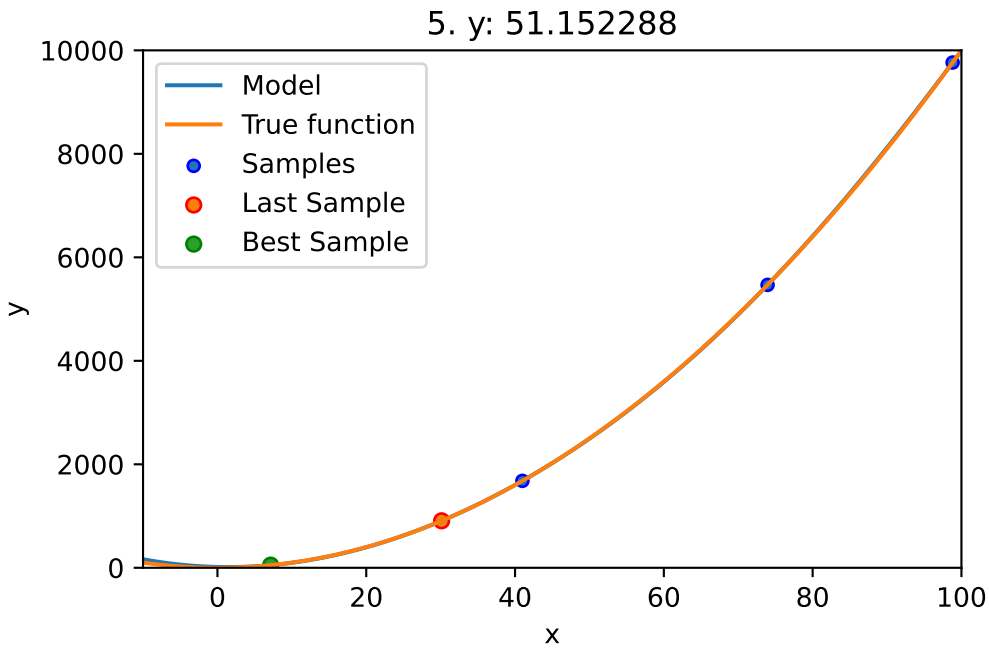
external parameter	type	description	default	mandatory
`noise`				
`model_optimizer`	object	optimizer	`differential_evolution`	
`model_fun_evals`				
`min_theta`			`-3.`	
`max_theta`			`3.`	
`n_theta`			`1`	
`n_p`			`1`	
`optim_p`			`False`	
`cod_type`			`"norm"`	
`var_type`				
`use_cod_y`	bool		`False`	

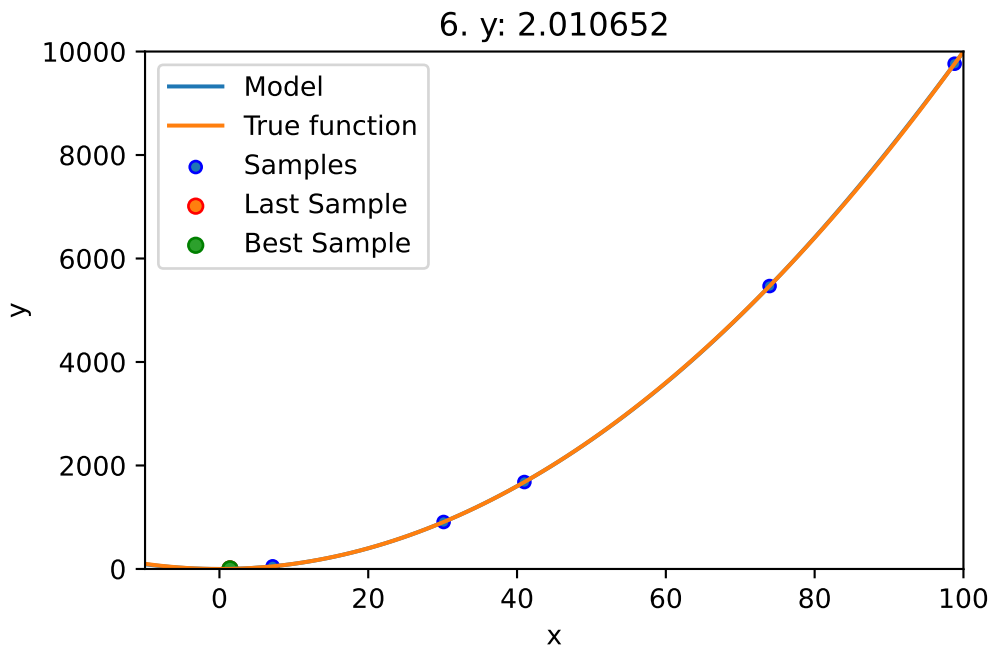
D.5 The `optimizer_control` Dictionary

external parameter	type	description	default	mandatory
`max_iter`	int	max number of iterations. Note: these are the cheap evaluations on the surrogate.	`1000`	no

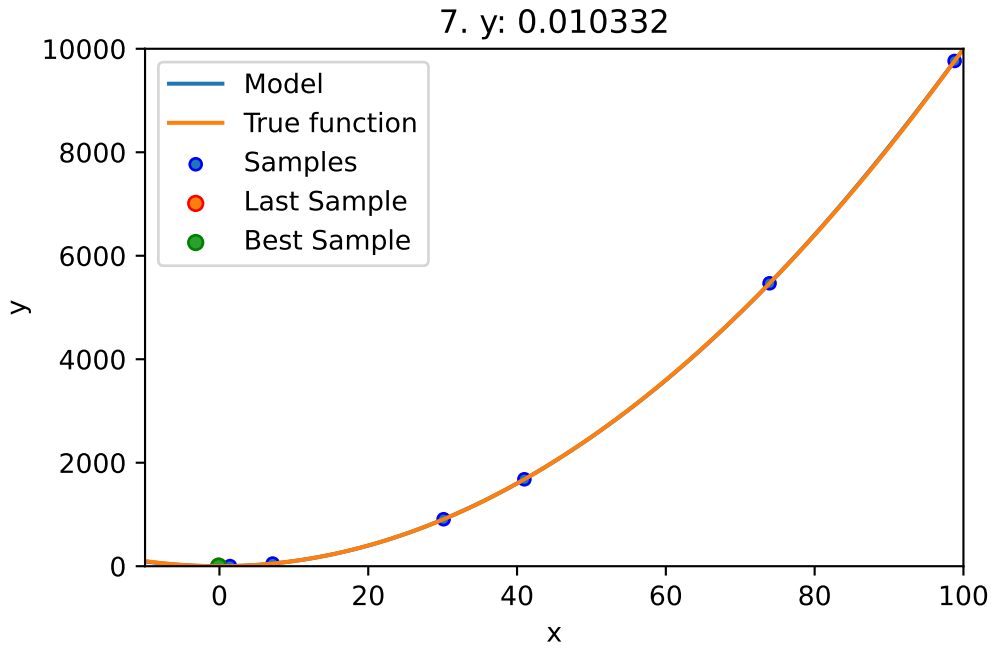
D.6 Run

```
spot_1.run()
```






```
spotPython tuning: 2.0106521524877827 [#########-] 85.71%
```



```
spotPython tuning: 0.01033163973935242 [##########] 100.00% Done...

<spotPython.spot.spot.Spot at 0x108b83250>
```

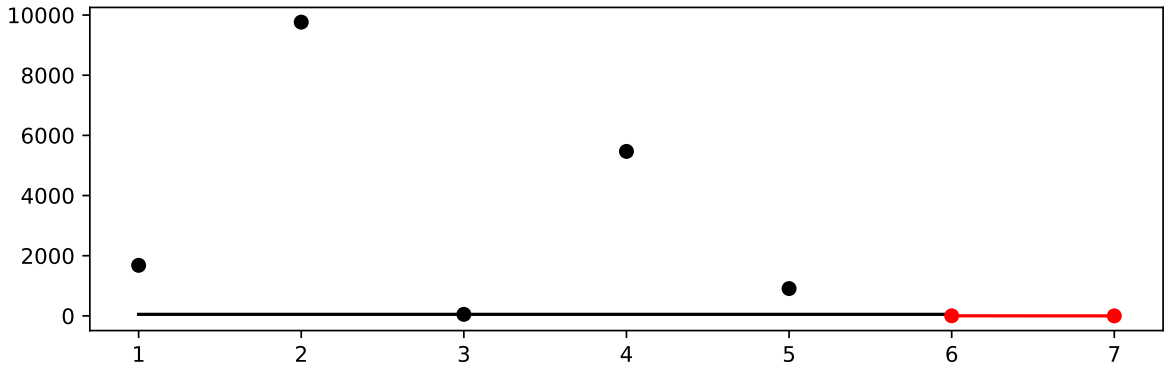
D.7 Print the Results

```
spot_1.print_results()
```

```
min y: 0.01033163973935242
x0: -0.10164467393499976

[['x0', -0.10164467393499976]]
```

D.8 Show the Progress

```
spot_1.plot_progress()
```

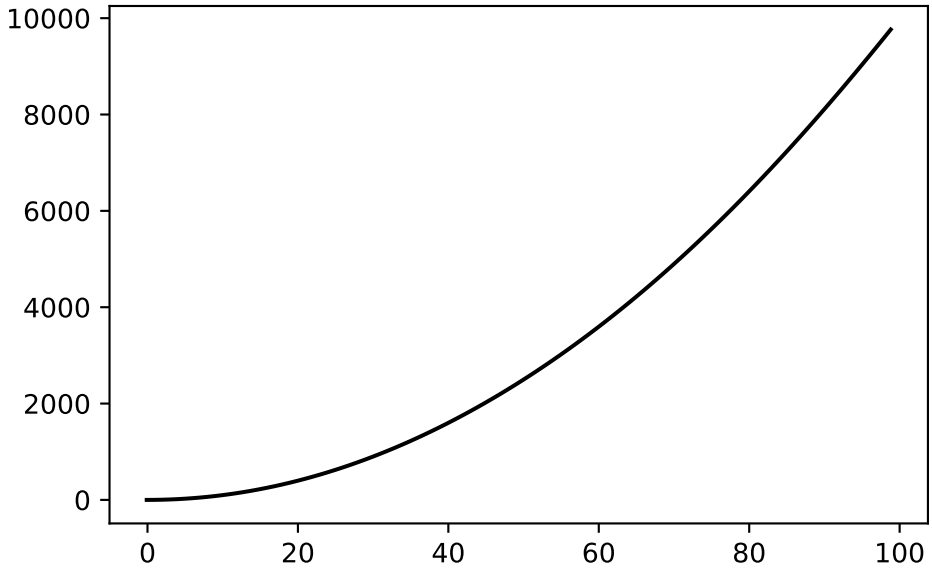


D.9 Visualize the Surrogate

- The plot method of the `kriging` surrogate is used.
- Note: the plot uses the interval defined by the ranges of the natural variables.

```
spot_1.surrogate.plot()
```

```
<Figure size 2700x1800 with 0 Axes>
```

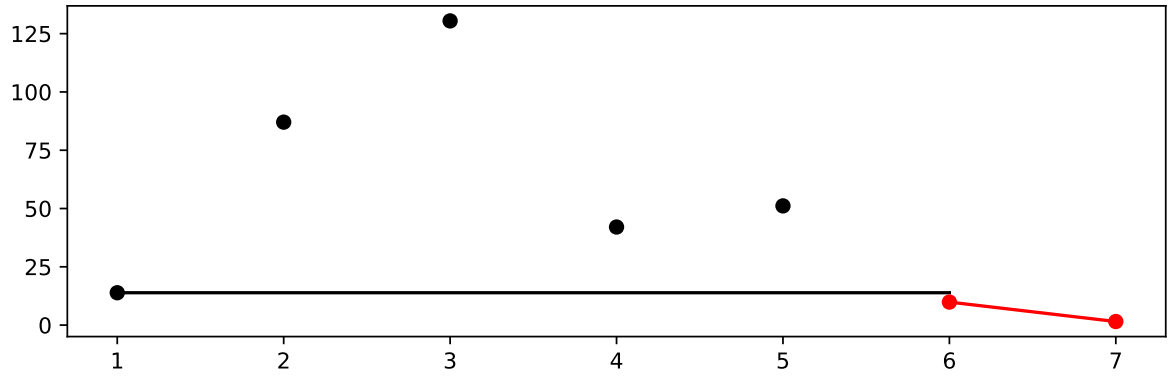


D.10 Run With a Specific Start Design

```
spot_x0 = spot.Spot(fun=fun,
                    lower = np.array([-10, -10]),
                    upper = np.array([10, 10]),
                    fun_evals = 7,
                    fun_repeats = 1,
                    max_time = inf,
                    noise = False,
                    tolerance_x = np.sqrt(np.spacing(1)),
                    var_type=["num"],
                    infill_criterion = "y",
                    n_points = 1,
                    seed=123,
                    log_level = 50,
                    show_models=False,
                    fun_control = {},
                    design_control={"init_size": 5,
                                    "repeats": 1},
                    surrogate_control={"noise": False,
```

```
                                    "cod_type": "norm",
                                    "min_theta": -4,
                                    "max_theta": 3,
                                    "n_theta": 2,
                                    "model_optimizer": differential_evolution,
                                    "model_fun_evals": 1000,
                                    })
spot_x0.run(X_start=np.array([0.5, -0.5]))
spot_x0.plot_progress()
```

```
spotPython tuning: 9.869941006907977 [#########-] 85.71%

spotPython tuning: 1.5261881935843578 [##########] 100.00% Done...
```



D.11 Init: Build Initial Design

```
from spotPython.design.spacefilling import spacefilling
from spotPython.build.kriging import Kriging
from spotPython.fun.objectivefunctions import analytical
gen = spacefilling(2)
rng = np.random.RandomState(1)
lower = np.array([-5,-0])
upper = np.array([10,15])
fun = analytical().fun_branin
fun_control = {"sigma": 0,
               "seed": 123}
```

```
  X = gen.scipy_lhd(10, lower=lower, upper = upper)
  print(X)
  y = fun(X, fun_control=fun_control)
  print(y)
```

```
[[ 8.97647221 13.41926847]
 [ 0.66946019  1.22344228]
 [ 5.23614115 13.78185824]
 [ 5.6149825  11.5851384 ]
 [-1.72963184  1.66516096]
 [-4.26945568  7.1325531 ]
 [ 1.26363761 10.17935555]
 [ 2.88779942  8.05508969]
 [-3.39111089  4.15213772]
 [ 7.30131231  5.22275244]]
[128.95676449  31.73474356 172.89678121 126.71295908  64.34349975
  70.16178611  48.71407916  31.77322887  76.91788181  30.69410529]
```

D.12 Replicability

Seed

```
  gen = spacefilling(2, seed=123)
  X0 = gen.scipy_lhd(3)
  gen = spacefilling(2, seed=345)
  X1 = gen.scipy_lhd(3)
  X2 = gen.scipy_lhd(3)
  gen = spacefilling(2, seed=123)
  X3 = gen.scipy_lhd(3)
  X0, X1, X2, X3
```

```
(array([[0.77254938, 0.31539299],
        [0.59321338, 0.93854273],
        [0.27469803, 0.3959685 ]]),
 array([[0.78373509, 0.86811887],
        [0.06692621, 0.6058029 ],
        [0.41374778, 0.00525456]]),
 array([[0.121357  , 0.69043832],
        [0.41906219, 0.32838498],
```

```
       [0.86742658, 0.52910374]]),
 array([[0.77254938, 0.31539299],
       [0.59321338, 0.93854273],
       [0.27469803, 0.3959685 ]]))
```

D.13 Surrogates

D.13.1 A Simple Predictor

The code below shows how to use a simple model for prediction. Assume that only two (very costly) measurements are available:

1. f(0) = 0.5
2. f(2) = 2.5

We are interested in the value at $x_0 = 1$, i.e., $f(x_0 = 1)$, but cannot run an additional, third experiment.

```
from sklearn import linear_model
X = np.array([[0], [2]])
y = np.array([0.5, 2.5])
S_lm = linear_model.LinearRegression()
S_lm = S_lm.fit(X, y)
X0 = np.array([[1]])
y0 = S_lm.predict(X0)
print(y0)
```

```
[1.5]
```

Central Idea: Evaluation of the surrogate model `S_lm` is much cheaper (or / and much faster) than running the real-world experiment f.

D.14 Demo/Test: Objective Function Fails

SPOT expects `np.nan` values from failed objective function values. These are handled. Note: SPOT's counter considers only successful executions of the objective function.

```
import numpy as np
from spotPython.fun.objectivefunctions import analytical
```

```
from spotPython.spot import spot
import numpy as np
from math import inf
# number of initial points:
ni = 20
# number of points
n = 30

fun = analytical().fun_random_error
lower = np.array([-1])
upper = np.array([1])
design_control={"init_size": ni}

spot_1 = spot.Spot(fun=fun,
            lower = lower,
            upper= upper,
            fun_evals = n,
            show_progress=False,
            design_control=design_control,)
spot_1.run()
# To check whether the run was successfully completed,
# we compare the number of evaluated points to the specified
# number of points.
assert spot_1.y.shape[0] == n
```

```
[ 0.53176481 -0.9053821          nan -0.21843718  0.78240941 -0.58120945
 -0.3923345   0.67234256  0.31802454 -0.68898927 -0.75129705  0.97550354
  0.41757584  0.0786237   0.82585329  0.23700598         nan -0.82319082
 -0.17991251         nan]
[-1.]

[-0.47259301]

[0.95541987]
[0.17335968]

[-0.58552368]
[-0.20126111]
[-0.60100809]
```

```
[-0.97897336]
[-0.2748985]

[0.8359486]
[0.99035591]

[nan]
[nan]

[0.01641232]
[0.5629346]
```

D.15 PyTorch: Detailed Description of the Data Splitting

D.15.1 Description of the "train_hold_out" Setting

The "train_hold_out" setting is used by default. It uses the loss function specfied in fun_control and the metric specified in fun_control.

1. First, the method HyperTorch().fun_torch is called.
2. fun_torc(), which is implemented in the file hypertorch.py, calls evaluate_hold_out() as follows:

```
df_eval, _ = evaluate_hold_out(
    model,
    train_dataset=fun_control["train"],
    shuffle=self.fun_control["shuffle"],
    loss_function=self.fun_control["loss_function"],
    metric=self.fun_control["metric_torch"],
    device=self.fun_control["device"],
    show_batch_interval=self.fun_control["show_batch_interval"],
    path=self.fun_control["path"],
    task=self.fun_control["task"],
    writer=self.fun_control["writer"],
    writerId=config_id,
)
```

Note: Only the data set fun_control["train"] is used for training and validation. It is used in evaluate_hold_out as follows:

```
trainloader, valloader = create_train_val_data_loaders(
                dataset=train_dataset, batch_size=batch_size_instance, shuffle=shuffle
            )
```

create_train_val_data_loaders() splits the train_dataset into trainloader and valloader using torch.utils.data.random_split() as follows:

```
def create_train_val_data_loaders(dataset, batch_size, shuffle, num_workers=0):
    test_abs = int(len(dataset) * 0.6)
    train_subset, val_subset = random_split(dataset, [test_abs, len(dataset) - test_abs])
    trainloader = torch.utils.data.DataLoader(
        train_subset, batch_size=int(batch_size), shuffle=shuffle, num_workers=num_workers
    )
    valloader = torch.utils.data.DataLoader(
```

```
        val_subset, batch_size=int(batch_size), shuffle=shuffle, num_workers=num_workers
    )
    return trainloader, valloader
```

The optimizer is set up as follows:

```
optimizer_instance = net.optimizer
lr_mult_instance = net.lr_mult
sgd_momentum_instance = net.sgd_momentum
optimizer = optimizer_handler(
    optimizer_name=optimizer_instance,
    params=net.parameters(),
    lr_mult=lr_mult_instance,
    sgd_momentum=sgd_momentum_instance,
)
```

3. `evaluate_hold_out()` sets the `net` attributes such as `epochs`, `batch_size`, `optimizer`, and `patience`. For each epoch, the methods `train_one_epoch()` and `validate_one_epoch()` are called, the former for training and the latter for validation and early stopping. The validation loss from the last epoch (not the best validation loss) is returned from `evaluate_hold_out`.
4. The method `train_one_epoch()` is implemented as follows:

```
def train_one_epoch(
    net,
    trainloader,
    batch_size,
    loss_function,
    optimizer,
    device,
    show_batch_interval=10_000,
    task=None,
):
    running_loss = 0.0
    epoch_steps = 0
    for batch_nr, data in enumerate(trainloader, 0):
        input, target = data
        input, target = input.to(device), target.to(device)
        optimizer.zero_grad()
        output = net(input)
        if task == "regression":
```

```
            target = target.unsqueeze(1)
            if target.shape == output.shape:
                loss = loss_function(output, target)
            else:
                raise ValueError(f"Shapes of target and output do not match:
                 {target.shape} vs {output.shape}")
        elif task == "classification":
            loss = loss_function(output, target)
        else:
            raise ValueError(f"Unknown task: {task}")
        loss.backward()
        torch.nn.utils.clip_grad_norm_(net.parameters(), max_norm=1.0)
        optimizer.step()
        running_loss += loss.item()
        epoch_steps += 1
        if batch_nr % show_batch_interval == (show_batch_interval - 1):
            print(
                "Batch: %5d. Batch Size: %d. Training Loss (running): %.3f"
                % (batch_nr + 1, int(batch_size), running_loss / epoch_steps)
            )
            running_loss = 0.0
    return loss.item()
```

5. The method `validate_one_epoch()` is implemented as follows:

```
def validate_one_epoch(net, valloader, loss_function, metric, device, task):
    val_loss = 0.0
    val_steps = 0
    total = 0
    correct = 0
    metric.reset()
    for i, data in enumerate(valloader, 0):
        # get batches
        with torch.no_grad():
            input, target = data
            input, target = input.to(device), target.to(device)
            output = net(input)
            # print(f"target: {target}")
            # print(f"output: {output}")
            if task == "regression":
                target = target.unsqueeze(1)
```

```
                if target.shape == output.shape:
                    loss = loss_function(output, target)
                else:
                    raise ValueError(f"Shapes of target and output
                        do not match: {target.shape} vs {output.shape}")
                metric_value = metric.update(output, target)
            elif task == "classification":
                loss = loss_function(output, target)
                metric_value = metric.update(output, target)
                _, predicted = torch.max(output.data, 1)
                total += target.size(0)
                correct += (predicted == target).sum().item()
            else:
                raise ValueError(f"Unknown task: {task}")
            val_loss += loss.cpu().numpy()
            val_steps += 1
    loss = val_loss / val_steps
    print(f"Loss on hold-out set: {loss}")
    if task == "classification":
        accuracy = correct / total
        print(f"Accuracy on hold-out set: {accuracy}")
    # metric on all batches using custom accumulation
    metric_value = metric.compute()
    metric_name = type(metric).__name__
    print(f"{metric_name} value on hold-out data: {metric_value}")
    return metric_value, loss
```

D.15.1.1 Description of the `"test_hold_out"` Setting

It uses the loss function specfied in `fun_control` and the metric specified in `fun_control`.

1. First, the method `HyperTorch().fun_torch` is called.
2. `fun_torc()` calls `spotPython.torch.traintest.evaluate_hold_out()` similar to the `"train_hold_out"` setting with one exception: It passes an additional `test` data set to `evaluate_hold_out()` as follows:

```
test_dataset=fun_control["test"]
```

`evaluate_hold_out()` calls `create_train_test_data_loaders` instead of `create_train_val_data_loaders`: The two data sets are used in `create_train_test_data_loaders` as follows:

```
def create_train_test_data_loaders(dataset, batch_size, shuffle, test_dataset,
        num_workers=0):
    trainloader = torch.utils.data.DataLoader(
        dataset, batch_size=int(batch_size), shuffle=shuffle,
        num_workers=num_workers
    )
    testloader = torch.utils.data.DataLoader(
        test_dataset, batch_size=int(batch_size), shuffle=shuffle,
        num_workers=num_workers
    )
    return trainloader, testloader
```

3. The following steps are identical to the `"train_hold_out"` setting. Only a different data loader is used for testing.

D.15.1.2 Detailed Description of the `"train_cv"` Setting

It uses the loss function specfied in `fun_control` and the metric specified in `fun_control`.

1. First, the method `HyperTorch().fun_torch` is called.
2. `fun_torc()` calls `spotPython.torch.traintest.evaluate_cv()` as follows (Note: Only the data set `fun_control["train"]` is used for CV.):

```
df_eval, _ = evaluate_cv(
    model,
    dataset=fun_control["train"],
    shuffle=self.fun_control["shuffle"],
    device=self.fun_control["device"],
    show_batch_interval=self.fun_control["show_batch_interval"],
    task=self.fun_control["task"],
    writer=self.fun_control["writer"],
    writerId=config_id,
)
```

3. In ‘evaluate_cv(), the following steps are performed: The optimizer is set up as follows:

```
optimizer_instance = net.optimizer
lr_instance = net.lr
sgd_momentum_instance = net.sgd_momentum
optimizer = optimizer_handler(optimizer_name=optimizer_instance,
     params=net.parameters(), lr_mult=lr_mult_instance)
```

`evaluate_cv()` sets the `net` attributes such as `epochs`, `batch_size`, `optimizer`, and `patience`. CV is implemented as follows:

```
def evaluate_cv(
    net,
    dataset,
    shuffle=False,
    loss_function=None,
    num_workers=0,
    device=None,
    show_batch_interval=10_000,
    metric=None,
    path=None,
    task=None,
    writer=None,
    writerId=None,
):
    lr_mult_instance = net.lr_mult
    epochs_instance = net.epochs
    batch_size_instance = net.batch_size
    k_folds_instance = net.k_folds
    optimizer_instance = net.optimizer
    patience_instance = net.patience
    sgd_momentum_instance = net.sgd_momentum
    removed_attributes, net = get_removed_attributes_and_base_net(net)
    metric_values = {}
    loss_values = {}
    try:
        device = getDevice(device=device)
        if torch.cuda.is_available():
            device = "cuda:0"
            if torch.cuda.device_count() > 1:
                print("We will use", torch.cuda.device_count(), "GPUs!")
                net = nn.DataParallel(net)
        net.to(device)
        optimizer = optimizer_handler(
            optimizer_name=optimizer_instance,
            params=net.parameters(),
            lr_mult=lr_mult_instance,
            sgd_momentum=sgd_momentum_instance,
        )
        kfold = KFold(n_splits=k_folds_instance, shuffle=shuffle)
```

```
        for fold, (train_ids, val_ids) in enumerate(kfold.split(dataset)):
            print(f"Fold: {fold + 1}")
            train_subsampler = torch.utils.data.SubsetRandomSampler(train_ids)
            val_subsampler = torch.utils.data.SubsetRandomSampler(val_ids)
            trainloader = torch.utils.data.DataLoader(
                dataset, batch_size=batch_size_instance,
                sampler=train_subsampler, num_workers=num_workers
            )
            valloader = torch.utils.data.DataLoader(
                dataset, batch_size=batch_size_instance,
                sampler=val_subsampler, num_workers=num_workers
            )
            # each fold starts with new weights:
            reset_weights(net)
            # Early stopping parameters
            best_val_loss = float("inf")
            counter = 0
            for epoch in range(epochs_instance):
                print(f"Epoch: {epoch + 1}")
                # training loss from one epoch:
                training_loss = train_one_epoch(
                    net=net,
                    trainloader=trainloader,
                    batch_size=batch_size_instance,
                    loss_function=loss_function,
                    optimizer=optimizer,
                    device=device,
                    show_batch_interval=show_batch_interval,
                    task=task,
                )
                # Early stopping check. Calculate validation loss from one epoch:
                metric_values[fold], loss_values[fold] = validate_one_epoch(
                    net, valloader=valloader, loss_function=loss_function,
                    metric=metric, device=device, task=task
                )
                # Log the running loss averaged per batch
                metric_name = "Metric"
                if metric is None:
                    metric_name = type(metric).__name__
                    print(f"{metric_name} value on hold-out data:
                        {metric_values[fold]}")
```

```
                if writer is not None:
                    writer.add_scalars(
                        "evaluate_cv fold:" + str(fold + 1) +
                        ". Train & Val Loss and Val Metric" + writerId,
                        {"Train loss": training_loss, "Val loss":
                        loss_values[fold], metric_name: metric_values[fold]},
                        epoch + 1,
                    )
                    writer.flush()
                if loss_values[fold] < best_val_loss:
                    best_val_loss = loss_values[fold]
                    counter = 0
                    # save model:
                    if path is not None:
                        torch.save(net.state_dict(), path)
                else:
                    counter += 1
                    if counter >= patience_instance:
                        print(f"Early stopping at epoch {epoch}")
                        break
        df_eval = sum(loss_values.values()) / len(loss_values.values())
        df_metrics = sum(metric_values.values()) / len(metric_values.values())
        df_preds = np.nan
    except Exception as err:
        print(f"Error in Net_Core. Call to evaluate_cv() failed. {err=},
            {type(err)=}")
        df_eval = np.nan
        df_preds = np.nan
    add_attributes(net, removed_attributes)
    if writer is not None:
        metric_name = "Metric"
        if metric is None:
            metric_name = type(metric).__name__
        writer.add_scalars(
            "CV: Val Loss and Val Metric" + writerId,
            {"CV-loss": df_eval, metric_name: df_metrics},
            epoch + 1,
        )
        writer.flush()
    return df_eval, df_preds, df_metrics
```

4. The method `train_fold()` is implemented as shown above.

5. The method `validate_one_epoch()` is implemented as shown above. In contrast to the hold-out setting, it is called for each of the k folds. The results are stored in a dictionaries `metric_values` and `loss_values`. The results are averaged over the k folds and returned as `df_eval`.

D.15.1.3 Detailed Description of the `"test_cv"` Setting

It uses the loss function specfied in `fun_control` and the metric specified in `fun_control`.

1. First, the method `HyperTorch().fun_torch` is called.
2. `fun_torc()` calls `spotPython.torch.traintest.evaluate_cv()` as follows:

```
df_eval, _ = evaluate_cv(
    model,
    dataset=fun_control["test"],
    shuffle=self.fun_control["shuffle"],
    device=self.fun_control["device"],
    show_batch_interval=self.fun_control["show_batch_interval"],
    task=self.fun_control["task"],
    writer=self.fun_control["writer"],
    writerId=config_id,
)
```

Note: The data set `fun_control["test"]` is used for CV. The rest is the same as for the `"train_cv"` setting.

D.15.1.4 Detailed Description of the Final Model Training and Evaluation

There are two methods that can be used for the final evaluation of a Pytorch model:

1. `"train_tuned` and
2. `"test_tuned"`.

`train_tuned()` is just a wrapper to `evaluate_hold_out` using the `train` data set. It is implemented as follows:

```
def train_tuned(
    net,
    train_dataset,
    shuffle,
    loss_function,
    metric,
```

```
    device=None,
    show_batch_interval=10_000,
    path=None,
    task=None,
    writer=None,
):
    evaluate_hold_out(
        net=net,
        train_dataset=train_dataset,
        shuffle=shuffle,
        test_dataset=None,
        loss_function=loss_function,
        metric=metric,
        device=device,
        show_batch_interval=show_batch_interval,
        path=path,
        task=task,
        writer=writer,
    )
```

The `test_tuned()` procedure is implemented as follows:

```
def test_tuned(net, shuffle, test_dataset=None, loss_function=None,
    metric=None, device=None, path=None, task=None):
    batch_size_instance = net.batch_size
    removed_attributes, net = get_removed_attributes_and_base_net(net)
    if path is not None:
        net.load_state_dict(torch.load(path))
        net.eval()
    try:
        device = getDevice(device=device)
        if torch.cuda.is_available():
            device = "cuda:0"
            if torch.cuda.device_count() > 1:
                print("We will use", torch.cuda.device_count(), "GPUs!")
                net = nn.DataParallel(net)
        net.to(device)
        valloader = torch.utils.data.DataLoader(
            test_dataset, batch_size=int(batch_size_instance),
            shuffle=shuffle,
            num_workers=0
        )
```

```
        metric_value, loss = validate_one_epoch(
            net, valloader=valloader, loss_function=loss_function,
            metric=metric, device=device, task=task
        )
        df_eval = loss
        df_metric = metric_value
        df_preds = np.nan
    except Exception as err:
        print(f"Error in Net_Core. Call to test_tuned() failed. {err=},
            {type(err)=}")
        df_eval = np.nan
        df_metric = np.nan
        df_preds = np.nan
    add_attributes(net, removed_attributes)
    print(f"Final evaluation: Validation loss: {df_eval}")
    print(f"Final evaluation: Validation metric: {df_metric}")
    print("------------------------------------------------")
    return df_eval, df_preds, df_metric
```

References

Bartz, Eva, Thomas Bartz-Beielstein, Martin Zaefferer, and Olaf Mersmann, eds. 2022. *Hyperparameter Tuning for Machine and Deep Learning with R - A Practical Guide*. Springer.

Bartz-Beielstein, Thomas. 2023. "PyTorch Hyperparameter Tuning with SPOT: Comparison with Ray Tuner and Default Hyperparameters on CIFAR10." https://github.com/sequential-parameter-optimization/spotPython/blob/main/notebooks/14_spot_ray_hpt_torch_cifar10.ipynb.

Forrester, Alexander, András Sóbester, and Andy Keane. 2008. *Engineering Design via Surrogate Modelling*. Wiley.

Santner, T J, B J Williams, and W I Notz. 2003. *The Design and Analysis of Computer Experiments*. Berlin, Heidelberg, New York: Springer.