homework-06

July 28, 2025

1 Homework 6

1.1 References

- Lectures 21-23 (inclusive).

1.2 Instructions

- Type your name and email in the "Student details" section below.
- Develop the code and generate the figures you need to solve the problems using this notebook.
- For the answers that require a mathematical proof or derivation you should type them using latex. If you have never written latex before and you find it exceedingly difficult, we will likely accept handwritten solutions.
- The total homework points are 100. Please note that the problems are not weighed equally.

If on Google Colab, install the following packages:

[ ]:
```
!pip install gpytorch
```

```
Collecting gpytorch
  Downloading gpytorch-1.14-py3-none-any.whl.metadata (8.0 kB)
Collecting jaxtyping (from gpytorch)
  Downloading jaxtyping-0.3.2-py3-none-any.whl.metadata (7.0 kB)
Requirement already satisfied: mpmath<=1.3,>=0.19 in
/usr/local/lib/python3.11/dist-packages (from gpytorch) (1.3.0)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.11/dist-
packages (from gpytorch) (1.6.1)
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.11/dist-
packages (from gpytorch) (1.16.0)
Collecting linear-operator>=0.6 (from gpytorch)
  Downloading linear_operator-0.6-py3-none-any.whl.metadata (15 kB)
Requirement already satisfied: torch>=2.0 in /usr/local/lib/python3.11/dist-
packages (from linear-operator>=0.6->gpytorch) (2.6.0+cu124)
Requirement already satisfied: numpy<2.6,>=1.25.2 in
/usr/local/lib/python3.11/dist-packages (from scipy>=1.6.0->gpytorch) (2.0.2)
Collecting wadler-lindig>=0.1.3 (from jaxtyping->gpytorch)
  Downloading wadler_lindig-0.1.7-py3-none-any.whl.metadata (17 kB)
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.11/dist-
packages (from scikit-learn->gpytorch) (1.5.1)
Requirement already satisfied: threadpoolctl>=3.1.0 in
```

```
/usr/local/lib/python3.11/dist-packages (from scikit-learn->gpytorch) (3.6.0)
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-
packages (from torch>=2.0->linear-operator>=0.6->gpytorch) (3.18.0)
Requirement already satisfied: typing-extensions>=4.10.0 in
/usr/local/lib/python3.11/dist-packages (from torch>=2.0->linear-
operator>=0.6->gpytorch) (4.14.1)
Requirement already satisfied: networkx in /usr/local/lib/python3.11/dist-
packages (from torch>=2.0->linear-operator>=0.6->gpytorch) (3.5)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-packages
(from torch>=2.0->linear-operator>=0.6->gpytorch) (3.1.6)
Requirement already satisfied: fsspec in /usr/local/lib/python3.11/dist-packages
(from torch>=2.0->linear-operator>=0.6->gpytorch) (2025.7.0)
Collecting nvidia-cuda-nvrtc-cu12==12.4.127 (from torch>=2.0->linear-
operator>=0.6->gpytorch)
  Downloading nvidia_cuda_nvrtc_cu12-12.4.127-py3-none-
manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cuda-runtime-cu12==12.4.127 (from torch>=2.0->linear-
operator>=0.6->gpytorch)
  Downloading nvidia_cuda_runtime_cu12-12.4.127-py3-none-
manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cuda-cupti-cu12==12.4.127 (from torch>=2.0->linear-
operator>=0.6->gpytorch)
  Downloading nvidia_cuda_cupti_cu12-12.4.127-py3-none-
manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cudnn-cu12==9.1.0.70 (from torch>=2.0->linear-
operator>=0.6->gpytorch)
  Downloading nvidia_cudnn_cu12-9.1.0.70-py3-none-
manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cublas-cu12==12.4.5.8 (from torch>=2.0->linear-
operator>=0.6->gpytorch)
  Downloading nvidia_cublas_cu12-12.4.5.8-py3-none-
manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cufft-cu12==11.2.1.3 (from torch>=2.0->linear-
operator>=0.6->gpytorch)
  Downloading nvidia_cufft_cu12-11.2.1.3-py3-none-
manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-curand-cu12==10.3.5.147 (from torch>=2.0->linear-
operator>=0.6->gpytorch)
  Downloading nvidia_curand_cu12-10.3.5.147-py3-none-
manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cusolver-cu12==11.6.1.9 (from torch>=2.0->linear-
operator>=0.6->gpytorch)
  Downloading nvidia_cusolver_cu12-11.6.1.9-py3-none-
manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cusparse-cu12==12.3.1.170 (from torch>=2.0->linear-
operator>=0.6->gpytorch)
  Downloading nvidia_cusparse_cu12-12.3.1.170-py3-none-
manylinux2014_x86_64.whl.metadata (1.6 kB)
```

```
Requirement already satisfied: nvidia-cusparselt-cu12==0.6.2 in
/usr/local/lib/python3.11/dist-packages (from torch>=2.0->linear-
operator>=0.6->gpytorch) (0.6.2)
Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in
/usr/local/lib/python3.11/dist-packages (from torch>=2.0->linear-
operator>=0.6->gpytorch) (2.21.5)
Requirement already satisfied: nvidia-nvtx-cu12==12.4.127 in
/usr/local/lib/python3.11/dist-packages (from torch>=2.0->linear-
operator>=0.6->gpytorch) (12.4.127)
Collecting nvidia-nvjitlink-cu12==12.4.127 (from torch>=2.0->linear-
operator>=0.6->gpytorch)
  Downloading nvidia_nvjitlink_cu12-12.4.127-py3-none-
manylinux2014_x86_64.whl.metadata (1.5 kB)
Requirement already satisfied: triton==3.2.0 in /usr/local/lib/python3.11/dist-
packages (from torch>=2.0->linear-operator>=0.6->gpytorch) (3.2.0)
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.11/dist-
packages (from torch>=2.0->linear-operator>=0.6->gpytorch) (1.13.1)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.11/dist-packages (from jinja2->torch>=2.0->linear-
operator>=0.6->gpytorch) (3.0.2)
Downloading gpytorch-1.14-py3-none-any.whl (277 kB)
                         277.7/277.7 kB
4.5 MB/s eta 0:00:00
Downloading linear_operator-0.6-py3-none-any.whl (176 kB)
                         176.3/176.3 kB
9.0 MB/s eta 0:00:00
Downloading jaxtyping-0.3.2-py3-none-any.whl (55 kB)
                         55.4/55.4 kB
3.5 MB/s eta 0:00:00
Downloading nvidia_cublas_cu12-12.4.5.8-py3-none-manylinux2014_x86_64.whl
(363.4 MB)
                         363.4/363.4 MB
3.4 MB/s eta 0:00:00
Downloading nvidia_cuda_cupti_cu12-12.4.127-py3-none-
manylinux2014_x86_64.whl (13.8 MB)
                         13.8/13.8 MB
40.0 MB/s eta 0:00:00
Downloading nvidia_cuda_nvrtc_cu12-12.4.127-py3-none-
manylinux2014_x86_64.whl (24.6 MB)
                         24.6/24.6 MB
24.1 MB/s eta 0:00:00
Downloading nvidia_cuda_runtime_cu12-12.4.127-py3-none-
manylinux2014_x86_64.whl (883 kB)
                         883.7/883.7 kB
25.7 MB/s eta 0:00:00
Downloading nvidia_cudnn_cu12-9.1.0.70-py3-none-manylinux2014_x86_64.whl
(664.8 MB)
                         664.8/664.8 MB
```

```
2.8 MB/s eta 0:00:00
Downloading nvidia_cufft_cu12-11.2.1.3-py3-none-manylinux2014_x86_64.whl
(211.5 MB)
                           211.5/211.5 MB
5.7 MB/s eta 0:00:00
Downloading nvidia_curand_cu12-10.3.5.147-py3-none-
manylinux2014_x86_64.whl (56.3 MB)
                           56.3/56.3 MB
12.1 MB/s eta 0:00:00
Downloading nvidia_cusolver_cu12-11.6.1.9-py3-none-
manylinux2014_x86_64.whl (127.9 MB)
                           127.9/127.9 MB
7.5 MB/s eta 0:00:00
Downloading nvidia_cusparse_cu12-12.3.1.170-py3-none-
manylinux2014_x86_64.whl (207.5 MB)
                           207.5/207.5 MB
6.0 MB/s eta 0:00:00
Downloading nvidia_nvjitlink_cu12-12.4.127-py3-none-
manylinux2014_x86_64.whl (21.1 MB)
                           21.1/21.1 MB
67.2 MB/s eta 0:00:00
Downloading wadler_lindig-0.1.7-py3-none-any.whl (20 kB)
Installing collected packages: wadler-lindig, nvidia-nvjitlink-cu12, nvidia-
curand-cu12, nvidia-cufft-cu12, nvidia-cuda-runtime-cu12, nvidia-cuda-nvrtc-
cu12, nvidia-cuda-cupti-cu12, nvidia-cublas-cu12, nvidia-cusparse-cu12, nvidia-
cudnn-cu12, jaxtyping, nvidia-cusolver-cu12, linear-operator, gpytorch
  Attempting uninstall: nvidia-nvjitlink-cu12
    Found existing installation: nvidia-nvjitlink-cu12 12.5.82
    Uninstalling nvidia-nvjitlink-cu12-12.5.82:
      Successfully uninstalled nvidia-nvjitlink-cu12-12.5.82
  Attempting uninstall: nvidia-curand-cu12
    Found existing installation: nvidia-curand-cu12 10.3.6.82
    Uninstalling nvidia-curand-cu12-10.3.6.82:
      Successfully uninstalled nvidia-curand-cu12-10.3.6.82
  Attempting uninstall: nvidia-cufft-cu12
    Found existing installation: nvidia-cufft-cu12 11.2.3.61
    Uninstalling nvidia-cufft-cu12-11.2.3.61:
      Successfully uninstalled nvidia-cufft-cu12-11.2.3.61
  Attempting uninstall: nvidia-cuda-runtime-cu12
    Found existing installation: nvidia-cuda-runtime-cu12 12.5.82
    Uninstalling nvidia-cuda-runtime-cu12-12.5.82:
      Successfully uninstalled nvidia-cuda-runtime-cu12-12.5.82
  Attempting uninstall: nvidia-cuda-nvrtc-cu12
    Found existing installation: nvidia-cuda-nvrtc-cu12 12.5.82
    Uninstalling nvidia-cuda-nvrtc-cu12-12.5.82:
      Successfully uninstalled nvidia-cuda-nvrtc-cu12-12.5.82
  Attempting uninstall: nvidia-cuda-cupti-cu12
    Found existing installation: nvidia-cuda-cupti-cu12 12.5.82
```

```
    Uninstalling nvidia-cuda-cupti-cu12-12.5.82:
      Successfully uninstalled nvidia-cuda-cupti-cu12-12.5.82
  Attempting uninstall: nvidia-cublas-cu12
    Found existing installation: nvidia-cublas-cu12 12.5.3.2
    Uninstalling nvidia-cublas-cu12-12.5.3.2:
      Successfully uninstalled nvidia-cublas-cu12-12.5.3.2
  Attempting uninstall: nvidia-cusparse-cu12
    Found existing installation: nvidia-cusparse-cu12 12.5.1.3
    Uninstalling nvidia-cusparse-cu12-12.5.1.3:
      Successfully uninstalled nvidia-cusparse-cu12-12.5.1.3
  Attempting uninstall: nvidia-cudnn-cu12
    Found existing installation: nvidia-cudnn-cu12 9.3.0.75
    Uninstalling nvidia-cudnn-cu12-9.3.0.75:
      Successfully uninstalled nvidia-cudnn-cu12-9.3.0.75
  Attempting uninstall: nvidia-cusolver-cu12
    Found existing installation: nvidia-cusolver-cu12 11.6.3.83
    Uninstalling nvidia-cusolver-cu12-11.6.3.83:
      Successfully uninstalled nvidia-cusolver-cu12-11.6.3.83
Successfully installed gpytorch-1.14 jaxtyping-0.3.2 linear-operator-0.6 nvidia-
cublas-cu12-12.4.5.8 nvidia-cuda-cupti-cu12-12.4.127 nvidia-cuda-nvrtc-
cu12-12.4.127 nvidia-cuda-runtime-cu12-12.4.127 nvidia-cudnn-cu12-9.1.0.70
nvidia-cufft-cu12-11.2.1.3 nvidia-curand-cu12-10.3.5.147 nvidia-cusolver-
cu12-11.6.1.9 nvidia-cusparse-cu12-12.3.1.170 nvidia-nvjitlink-cu12-12.4.127
wadler-lindig-0.1.7
```

[ ]:
```python
import torch
import gpytorch
from gpytorch.kernels import RBFKernel, ScaleKernel
```

[ ]:
```python
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import matplotlib_inline
matplotlib_inline.backend_inline.set_matplotlib_formats('svg')
import seaborn as sns
sns.set_context("paper")
sns.set_style("ticks")

import scipy
import scipy.stats as st
import urllib.request
import os

def download(
    url : str,
    local_filename : str = None
):
```

```
    """Download a file from a url.

    Arguments
    url             -- The url we want to download.
    local_filename -- The filemame to write on. If not
                      specified
    """
    if local_filename is None:
        local_filename = os.path.basename(url)
    urllib.request.urlretrieve(url, local_filename)

def sample_functions(mean_func, kernel_func, num_samples=10, num_test=100,␣
 ↪nugget=1e-3):
    """Sample functions from a Gaussian process.

    Arguments:
        mean_func -- the mean function. It must be a callable that takes a␣
 ↪tensor
            of shape (num_test, dim) and returns a tensor of shape (num_test,␣
 ↪1).
        kernel_func -- the covariance function. It must be a callable that takes
            a tensor of shape (num_test, dim) and returns a tensor of shape
            (num_test, num_test).
        num_samples -- the number of samples to take. Defaults to 10.
        num_test -- the number of test points. Defaults to 100.
        nugget -- a small number required for stability. Defaults to 1e-5.
    """
    X = torch.linspace(0, 1, num_test)[:, None]
    m = mean_func(X)
    C = kernel_func.forward(X, X) + nugget * torch.eye(X.shape[0])
    L = torch.linalg.cholesky(C)
    fig, ax = plt.subplots()
    ax.plot(X, m.detach(), label='mean')
    for i in range(num_samples):
        z = torch.randn(X.shape[0], 1)
        f = m[:, None] + L @ z
        ax.plot(X.flatten(), f.detach().flatten(), color=sns.
 ↪color_palette()[1], linewidth=0.5,
                label='sample' if i == 0 else None
            )
    plt.legend(loc='best', frameon=False)
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    ax.set_ylim(-5, 5)
    sns.despine(trim=True);
```

```
import gpytorch

class ExactGP(gpytorch.models.ExactGP):
    def __init__(self,
                 train_x,
                 train_y,
                 likelihood=gpytorch.likelihoods.GaussianLikelihood(),
                mean_module=gpytorch.means.ConstantMean(),
                covar_module=ScaleKernel(RBFKernel())
        ):
        super().__init__(train_x, train_y, likelihood)
        self.mean_module = mean_module
        self.covar_module = covar_module

    def forward(self, x):
        mean_x = self.mean_module(x)
        covar_x = self.covar_module(x)
        return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)


def plot_1d_regression(
    x_star,
    model,
    ax=None,
    f_true=None,
    num_samples=10,
    xlabel='$x$',
    ylabel='$y$'
):
    """Plot the posterior predictive.

    Arguments
    x_start  --  The test points on which to evaluate.
    model    --  The trained model.

    Keyword Arguments
    ax          --  An axes object to write on.
    f_true      --  The true function.
    num_samples --  The number of samples.
    xlabel      --  The x-axis label.
    ylabel      --  The y-axis label.
    """
    f_star = model(x_star)
    m_star = f_star.mean
    v_star = f_star.variance
    y_star = model.likelihood(f_star)
    yv_star = y_star.variance
```

```
    f_lower = (
        m_star - 2.0 * torch.sqrt(v_star)
    )
    f_upper = (
        m_star + 2.0 * torch.sqrt(v_star)
    )

    y_lower = m_star - 2.0 * torch.sqrt(yv_star)
    y_upper = m_star + 2.0 * torch.sqrt(yv_star)

    if ax is None:
        fig, ax = plt.subplots()

    ax.plot(model.train_inputs[0].flatten().detach(),
            model.train_targets.detach(),
            'k.',
            markersize=1,
            markeredgewidth=2,
            label='Observations'
    )

    ax.plot(
        x_star,
        m_star.detach(),
        lw=2,
        label='Posterior mean',
        color=sns.color_palette()[0]
    )

    ax.fill_between(
        x_star.flatten().detach(),
        f_lower.flatten().detach(),
        f_upper.flatten().detach(),
        alpha=0.5,
        label='Epistemic uncertainty',
        color=sns.color_palette()[0]
    )

    ax.fill_between(
        x_star.detach().flatten(),
        y_lower.detach().flatten(),
        f_lower.detach().flatten(),
        color=sns.color_palette()[1],
        alpha=0.5,
        label='Aleatory uncertainty'
    )
```

```
    ax.fill_between(
        x_star.detach().flatten(),
        f_upper.detach().flatten(),
        y_upper.detach().flatten(),
        color=sns.color_palette()[1],
        alpha=0.5,
        label=None
    )


    if f_true is not None:
        ax.plot(
            x_star,
            f_true(x_star),
            'm-.',
            label='True function'
        )

    if num_samples > 0:
        f_post_samples = f_star.sample(
            sample_shape=torch.Size([10])
        )
        ax.plot(
            x_star.numpy(),
            f_post_samples.T.detach().numpy(),
            color="red",
            lw=0.5
        )
        # This is just to add the legend entry
        ax.plot(
            [],
            [],
            color="red",
            lw=0.5,
            label="Posterior samples"
        )

    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)

    plt.legend(loc='best', frameon=False)
    sns.despine(trim=True)

    return dict(m_star=m_star, v_star=v_star, ax=ax)


def train(model, train_x, train_y, n_iter=10, lr=0.1):
```

```
    """Train the model.

    Arguments
    model   --  The model to train.
    train_x --  The training inputs.
    train_y --  The training labels.
    n_iter  --  The number of iterations.
    """
    model.train()
    optimizer = torch.optim.LBFGS(model.parameters(),␣
↪line_search_fn='strong_wolfe')
    likelihood = model.likelihood
    mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)
    def closure():
        optimizer.zero_grad()
        output = model(train_x)
        loss = -mll(output, train_y)
        loss.backward()
        print(loss)
        return loss
    for i in range(n_iter):
        loss = optimizer.step(closure)
        if (i + 1) % 1 == 0:
            print(f'Iter {i + 1:3d}/{n_iter} - Loss: {loss.item():.3f}')
    model.eval()
```

1.3 Student details

- **First Name:** Hannah
- **Last Name:** Moskios
- **Email:** hmoskios@purdue.edu

1.4 Problem 1 - Defining priors on function spaces

In this problem, we will explore further how Gaussian processes can be used to define probability measures over function spaces. To this end, assume that there is a 1D function, call if $f(x)$, which we do not know. For simplicity, assume that x takes values in $[0, 1]$. We will employ Gaussian process regression to encode our state of knowledge about $f(x)$ and sample some possibilities. For each of the cases below: + Assume that $f \sim \text{GP}(m, k)$ and pick a mean $(m(x))$ and a covariance function $f(x)$ that match the provided information. + Write code that samples a few times (up to five) the values of $f(x)$ at 100 equidistant points between 0 and 1.

1.4.1 Part A - Super smooth function with known length scale

Assume that you hold the following beliefs + You know that $f(x)$ has as many derivatives as you want and they are all continuous + You don't know if $f(x)$ has a specific trend. + You think that $f(x)$ has "wiggles" that are approximatly of size $\Delta x = 0.1$. + You think that $f(x)$ is between -4 and 4.

Answer:

I am doing this for you so that you have a concrete example of what is requested.

The mean function should be:

$$m(x) = 0.$$

The covariance function should be a squared exponential:

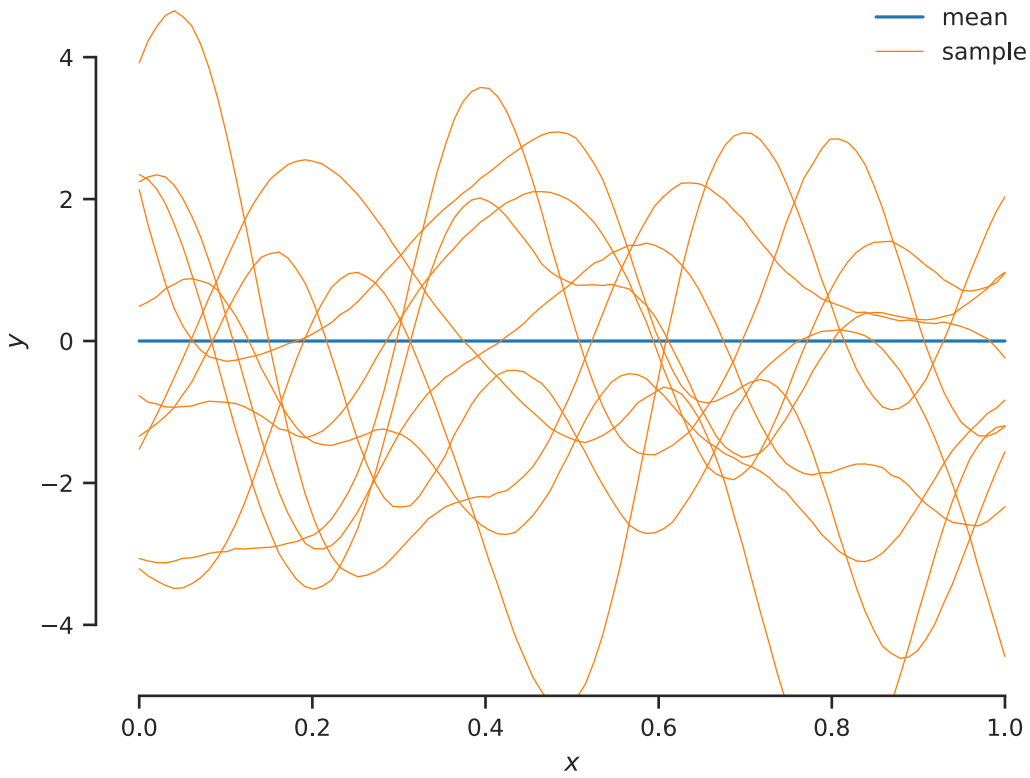
$$k(x, x') = s^2 \exp\left\{-\frac{(x - x')^2}{2\ell^2}\right\},$$

with variance:

$$s^2 = k(x, x) = \mathbb{V}[f(x)] = 4,$$

and lengthscale $\ell = 0.1$. We chose the variance to be 4.0 so that with (about) 95% probability, the values of $f(x)$ are between -4 and 4.

[ ]:
```
import torch
import gpytorch
from gpytorch.kernels import RBFKernel, ScaleKernel

# Define the covariance function
k = ScaleKernel(RBFKernel())
k.outputscale = 4.0
k.base_kernel.lengthscale = 0.1

# Define the mean function
mean = gpytorch.means.ConstantMean()
mean.constant = 0.0

# Sample functions
sample_functions(mean, k, nugget=1e-4)
```



1.4.2 Part B - Super smooth function with known ultra-small length scale

Assume that you hold the following beliefs + You know that $f(x)$ has as many derivatives as you want and they are all continuous + You don't know if $f(x)$ has a specific trend. + You think that $f(x)$ has "wiggles" that are approximatly of size $\Delta x = 0.05$. + You think that $f(x)$ is between -3 and 3.

Answer:

Since we do not know if $f(x)$ has a specific trend, the mean function should be:

$$m(x) = 0$$

Since $f(x)$ is differentiable and continuous, the covariance function should be a squared exponential:

$$k(x, x') = s^2 \exp\left\{-\frac{(x - x')^2}{2\ell^2}\right\}$$

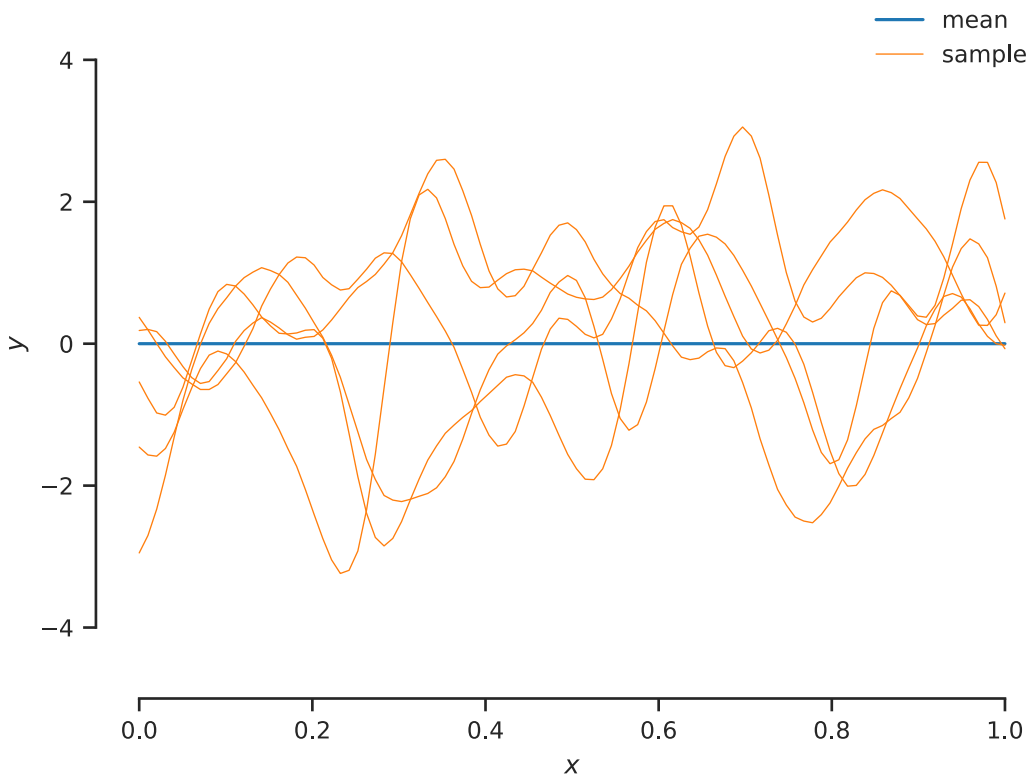
Since the size of the "wiggles" is 0.05, the length scale should be:

$$\ell = 0.05$$

Since we think that $f(x)$ is between -3 and 3, this means that $2\sigma = 3$ with 95% probability. Therefore, the variance is:

$$2\sigma = 3 \longrightarrow \sigma = \frac{3}{2} \longrightarrow \sigma^2 = (\frac{3}{2})^2$$

[ ]:
```
# Your code here
# Define the covariance function
k = ScaleKernel(RBFKernel())
k.outputscale = (3.0 / 2) ** 2
k.base_kernel.lengthscale = 0.05

# Define the mean function
mean = gpytorch.means.ConstantMean()
mean.constant = 0.0

# Sample functions
sample_functions(mean, k, nugget=1e-4, num_samples=5)
```



1.4.3 Part C - Continuous function with known length scale

Assume that you hold the following beliefs + You know that $f(x)$ is continuous, nowhere differentiable. + You don't know if $f(x)$ has a specific trend. + You think that $f(x)$ has "wiggles" that are approximately of size $\Delta x = 0.1$. + You think that $f(x)$ is between -5 and 5.

Hint: Use `gpytorch.kernels.MaternKernel` with $\nu = 1/2$.

Answer:

Since we do not know if $f(x)$ has a specific trend, the mean function should be:

$$m(x) = 0$$

Since $f(x)$ is continuous and nowhere differentiable, we should use the Matérn covariance function with $\nu = 0.5$:

$$k(x, x') = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\sqrt{2\nu} \frac{|x - x'|}{\ell} \right)^{\nu} K_{\nu} \left(\sqrt{2\nu} \frac{|x - x'|}{\ell} \right)$$

Since the size of the "wiggles" is 0.1, the length scale should be:

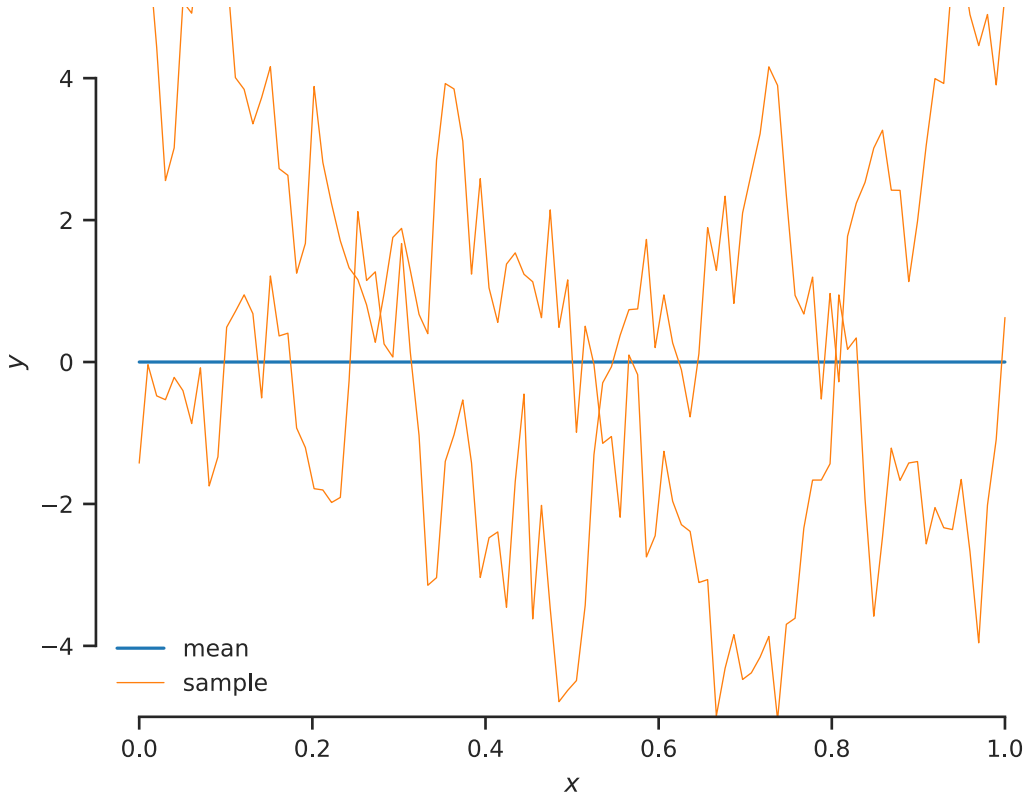
$$\ell = 0.1$$

Since we think that $f(x)$ is between -5 and 5, this means that $2\sigma = 5$ with 95% probability. Therefore, the variance is:

$$2\sigma = 5 \longrightarrow \sigma = \frac{5}{2} \longrightarrow \sigma^2 = (\frac{5}{2})^2$$

Note: We will only use 2 samples here to make the functions easier to visualize.

[ ]:
```
# Your code here
from gpytorch.kernels import MaternKernel

# Define the covariance function
k = ScaleKernel(MaternKernel(nu=0.5))
k.outputscale = (5.0 / 2) ** 2
k.base_kernel.lengthscale = 0.1

# Define the mean function
mean = gpytorch.means.ConstantMean()
mean.constant = 0.0

# Sample functions
sample_functions(mean, k, nugget=1e-4, num_samples=2)
```



1.4.4 Part D - Smooth periodic function with known length scale

Assume that you hold the following beliefs + You know that $f(x)$ is smooth. + You know that $f(x)$ is periodic with period 0.1. + You don't know if $f(x)$ has a specific trend. + You think that $f(x)$ has "wiggles" that are approximately of size $\Delta x = 0.5$ of the period. + You think that $f(x)$ is between -5 and 5.

Hint: Use `gpytorch.kernels.PeriodicKernel`.

Answer:

Since we do not know if $f(x)$ has a specific trend, the mean function should be:

$$m(x) = 0$$

Since $f(x)$ is smooth and periodic, we should use the Periodic kernel:

$$k(x, x') = \exp\left\{-\frac{2}{\ell}\sin^2\left(\frac{\pi}{p}|x - x'|\right)\right\}$$

Since the size of the "wiggles" is 0.5 of the period, the length scale should be:

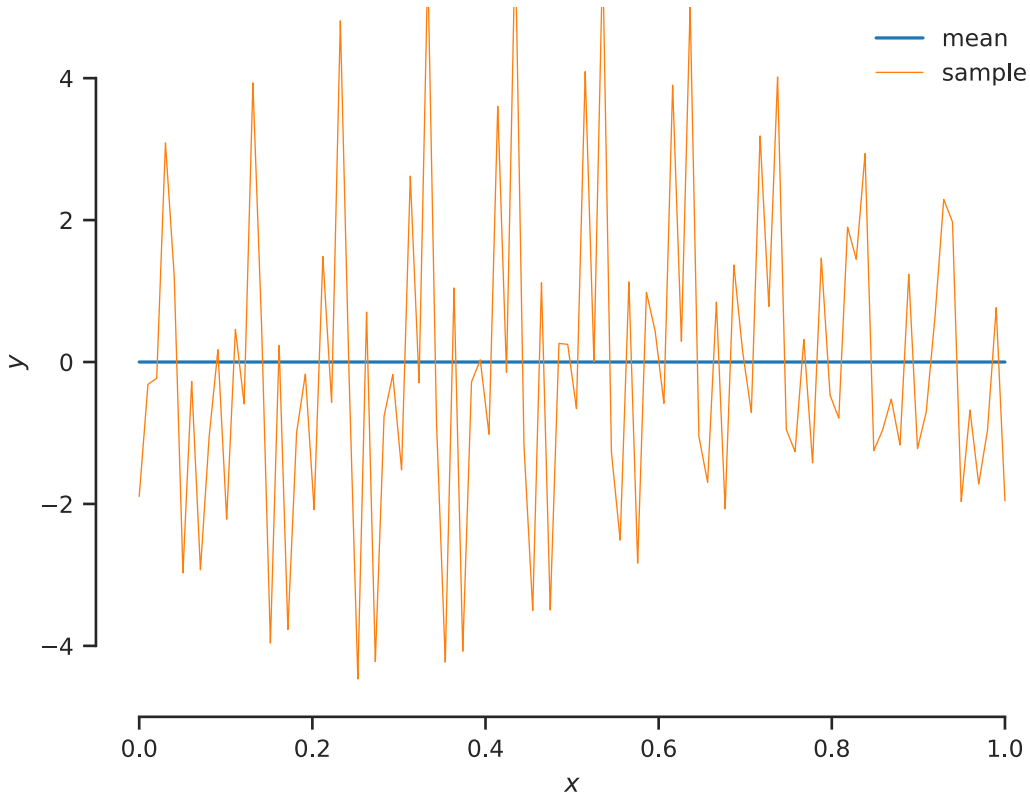
$$\ell = 0.5 \cdot \texttt{period} = 0.5 \cdot 0.1 = 0.05$$

Since we think that $f(x)$ is between -5 and 5, this means that $2\sigma = 5$ with 95% probability. Therefore, the variance is:

$$2\sigma = 5 \longrightarrow \sigma = \frac{5}{2} \longrightarrow \sigma^2 = (\frac{5}{2})^2$$

Note: We will only use 1 sample here to make the function easier to visualize.

[ ]:
```
# Your code here
from gpytorch.kernels import PeriodicKernel

# Define the covariance function
k = ScaleKernel(PeriodicKernel())
k.outputscale = (5 / 2) ** 2
period_length = 0.1
k.base_kernel.lengthscale = 0.5 * period_length
k.base_kernel.period_length = period_length

# Define the mean function
mean = gpytorch.means.ConstantMean()
mean.constant = 0.0

# Sample functions
sample_functions(mean, k, nugget=1e-3, num_samples=1)
```



1.4.5 Part E - Smooth periodic function with known length scale

Assume that you hold the following beliefs + You know that $f(x)$ is smooth. + You know that $f(x)$ is periodic with period 0.1. + You don't know if $f(x)$ has a specific trend. + You think that $f(x)$ has "wiggles" that are approximately of size $\Delta x = 0.1$ of the period (**the only thing that is different compared to D**). + You think that $f(x)$ is between -5 and 5.

Hint: Use `gpytorch.kernels.PeriodicKernel`.

Answer:

Since we do not know if $f(x)$ has a specific trend, the mean function should be:

$$m(x) = 0$$

Since $f(x)$ is smooth and periodic, we should use the Periodic kernel:

$$k(x, x') = \exp\left\{-\frac{2}{\ell}\sin^2\left(\frac{\pi}{p}|x - x'|\right)\right\}$$

Since the size of the "wiggles" is 0.1 of the period, the length scale should be:

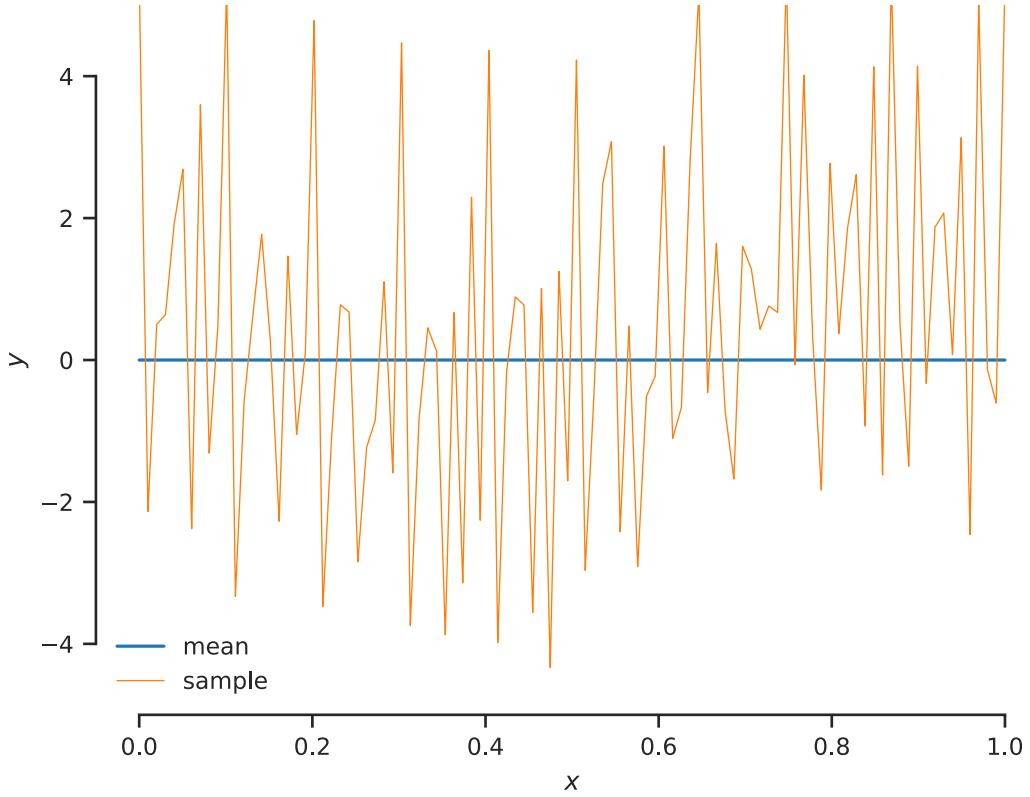
$$\ell = 0.1 \cdot \texttt{period} = 0.1 \cdot 0.1 = 0.01$$

Since we think that $f(x)$ is between -5 and 5, this means that $2\sigma = 5$ with 95% probability. Therefore, the variance is:

$$2\sigma = 5 \longrightarrow \sigma = \frac{5}{2} \longrightarrow \sigma^2 = (\frac{5}{2})^2$$

Note: We will only use 1 sample here to make the function easier to visualize.

[ ]:
```
# Your code here
# Define the covariance function
k = ScaleKernel(PeriodicKernel())
k.outputscale = (5 / 2) ** 2
period_length = 0.1
k.base_kernel.lengthscale = 0.1 * period_length
k.base_kernel.period_length = period_length

# Define the mean function
mean = gpytorch.means.ConstantMean()
mean.constant = 0.0

# Sample functions
sample_functions(mean, k, nugget=1e-4, num_samples=1)
```



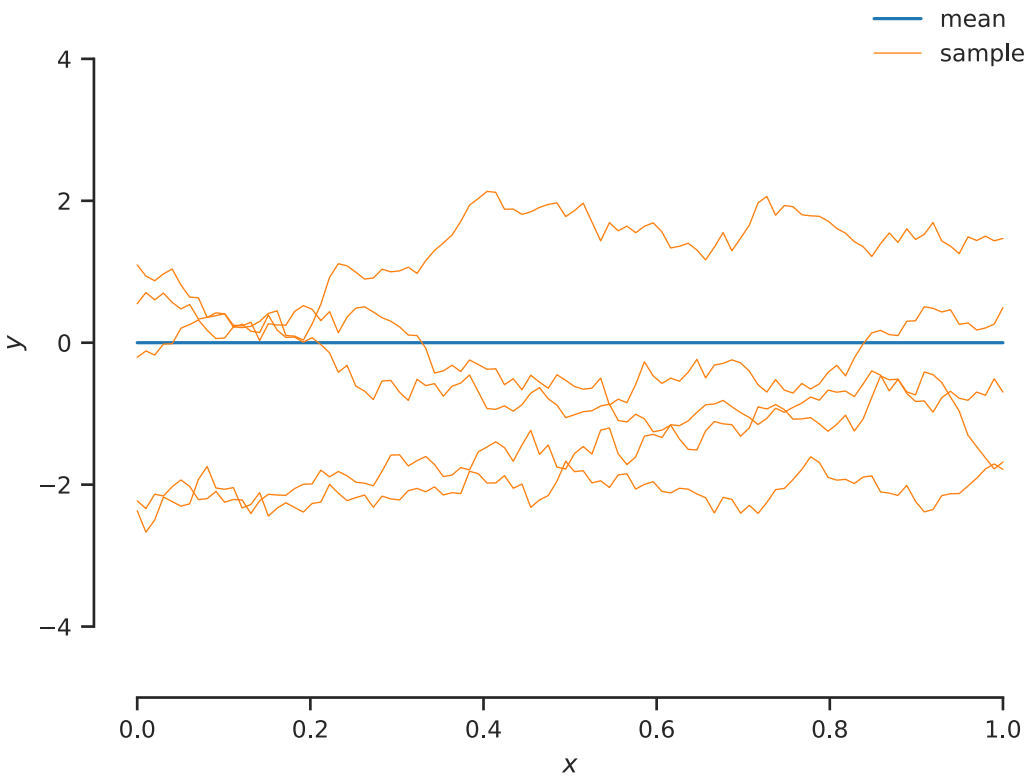
1.4.6 Part F - The sum of two functions

Assume that you hold the following beliefs + You know that $f(x) = f_1(x) + f_2(x)$, where: - $f_1(x)$ is smooth with variance 2 and length scale 0.5 - $f_2(x)$ is continuous, nowhere differentiable with variance 0.1 and length scale 0.1

Hint: Use must create a new covariance function that is the sum of two other covariances.

[ ]:
```
# Your code here
# Define the covariance function for f1(x)
k1 = ScaleKernel(RBFKernel())
k1.outputscale = 2.0
k1.base_kernel.lengthscale = 0.5

# Define the covariance function for f2(x)
k2 = ScaleKernel(MaternKernel(nu=0.5))
k2.outputscale = 0.1
k2.base_kernel.lengthscale = 0.1

# Define the overall covariance function
k = k1 + k2
```

```
# Define the mean function
mean = gpytorch.means.ConstantMean()
mean.constant = 0.0

# Sample functions
sample_functions(mean, k, nugget=1e-4, num_samples=5)
```



1.4.7 Part G - The product of two functions

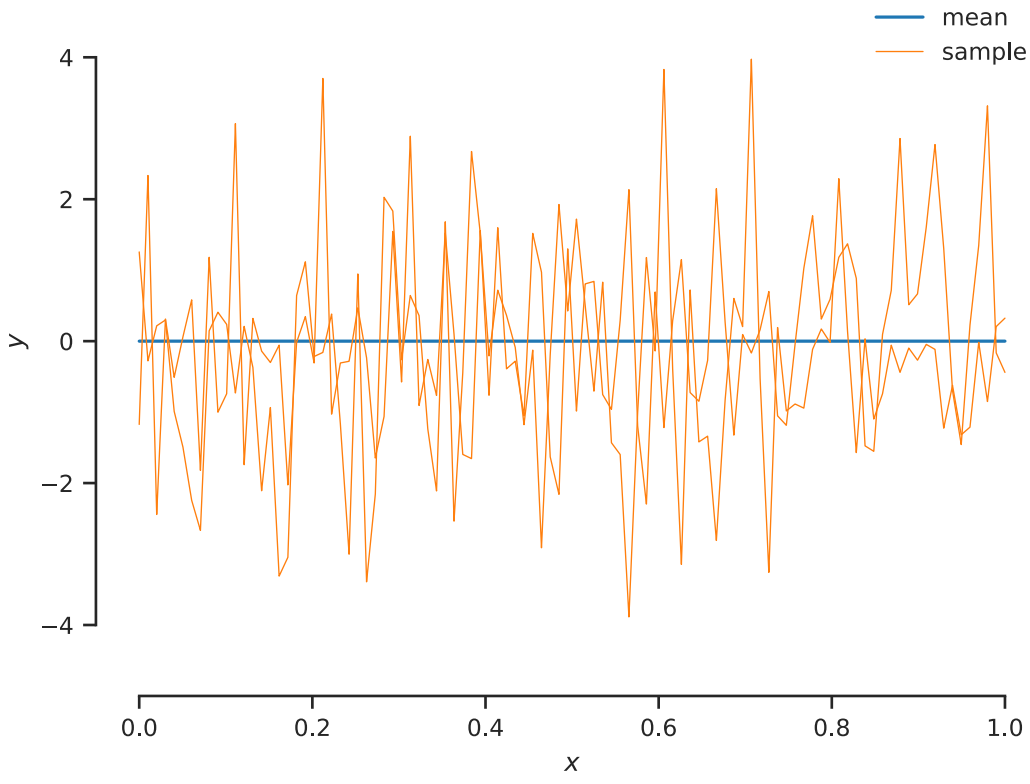
Assume that you hold the following beliefs + You know that $f(x) = f_1(x)f_2(x)$, where: - $f_1(x)$ is smooth, periodic (period = 0.1), length scale 0.1 (relative to the period), and variance 2. - $f_2(x)$ is smooth with length scale 0.5 and variance 1.

Hint: Use must create a new covariance function that is the product of two other covariances.

Note: We will only use 2 samples here to make the functions easier to visualize.

[ ]:
```
# Your code here
# Define the covariance function for f1(x)
k1 = ScaleKernel(PeriodicKernel())
k1.outputscale = 2.0
period_length = 0.1
```

```
k1.base_kernel.lengthscale = 0.1 * period_length
k1.base_kernel.period_length = period_length

# Define the covariance function for f2(x)
k2 = ScaleKernel(RBFKernel())
k2.outputscale = 1.0
k2.base_kernel.lengthscale = 0.5

# Define the overall covariance function
k = k1 * k2

# Define the mean function
mean = gpytorch.means.ConstantMean()
mean.constant = 0.0

# Sample functions
sample_functions(mean, k, nugget=1e-4, num_samples=2)
```



1.5 Problem 2

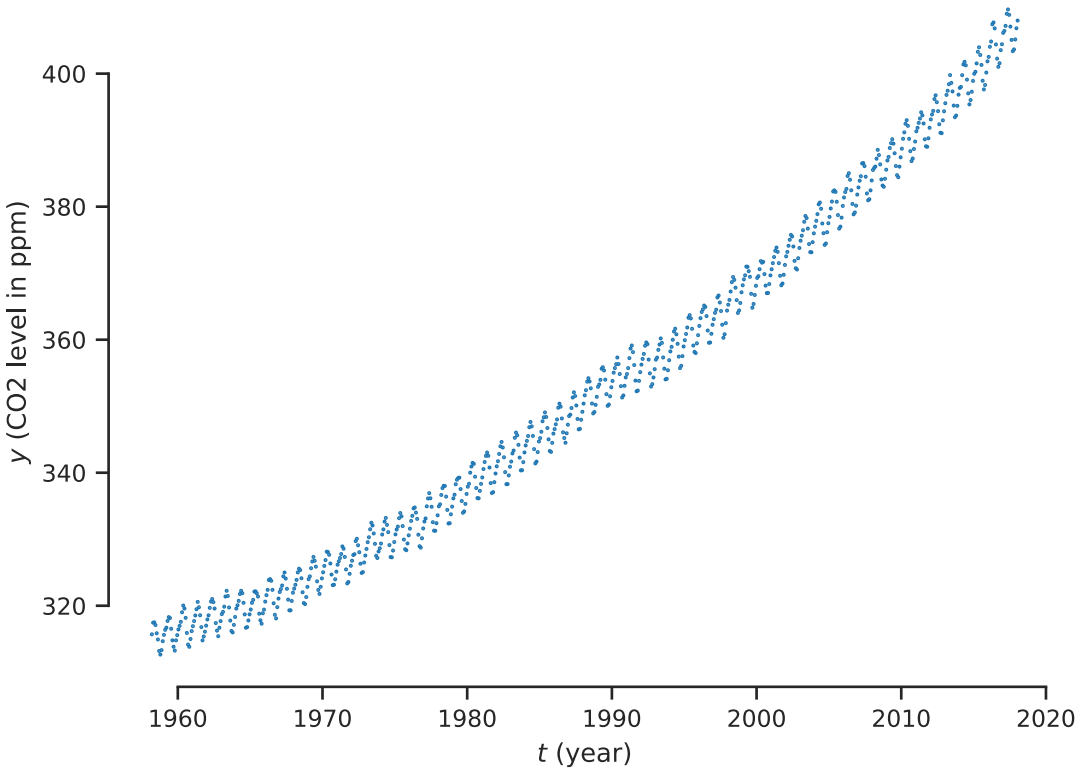
The National Oceanic and Atmospheric Administration (NOAA) has been measuring the levels of atmospheric CO2 at the Mauna Loa, Hawaii. The measurements start in March 1958 and go back to January 2016. The data can be found here. The Python cell below downloads and plots the data set.

[ ]:
```
url = "https://raw.githubusercontent.com/PredictiveScienceLab/data-analytics-se/
 ↪master/lecturebook/data/mauna_loa_co2.txt"
!curl -O $url
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 46015  100 46015    0     0   216k      0 --:--:-- --:--:-- --:--:--  217k
```

[ ]:
```
data = np.loadtxt('mauna_loa_co2.txt')
```

[ ]:
```
#load data
t = data[:, 2]  #time (in decimal dates)
y = data[:, 4]  #CO2 level (mole fraction in dry air, micromol/mol, abbreviated␣
 ↪as ppm)
fig, ax = plt.subplots(1, 1)
ax.plot(t, y, '.', markersize=1)
ax.set_xlabel('$t$ (year)')
ax.set_ylabel('$y$ (CO2 level in ppm)')
sns.despine(trim=True);
```



Overall, we observe a steady growth of CO2 levels. The wiggles correspond to seasonal changes. Since most of the population inhabits the northern hemisphere, fuel consumption increases during the northern winters, and CO2 emissions follow. Our goal is to study this dataset with Gaussian process regression. Specifically, we would like to predict the evolution of the CO2 levels from Feb 2018 to Feb 2028 and quantify our uncertainty about this prediction.

Working with a scaled version of the inputs and outputs is always a good idea. We are going to scale the times as follows:

$$t_s = t - t_{\text{min}}.$$

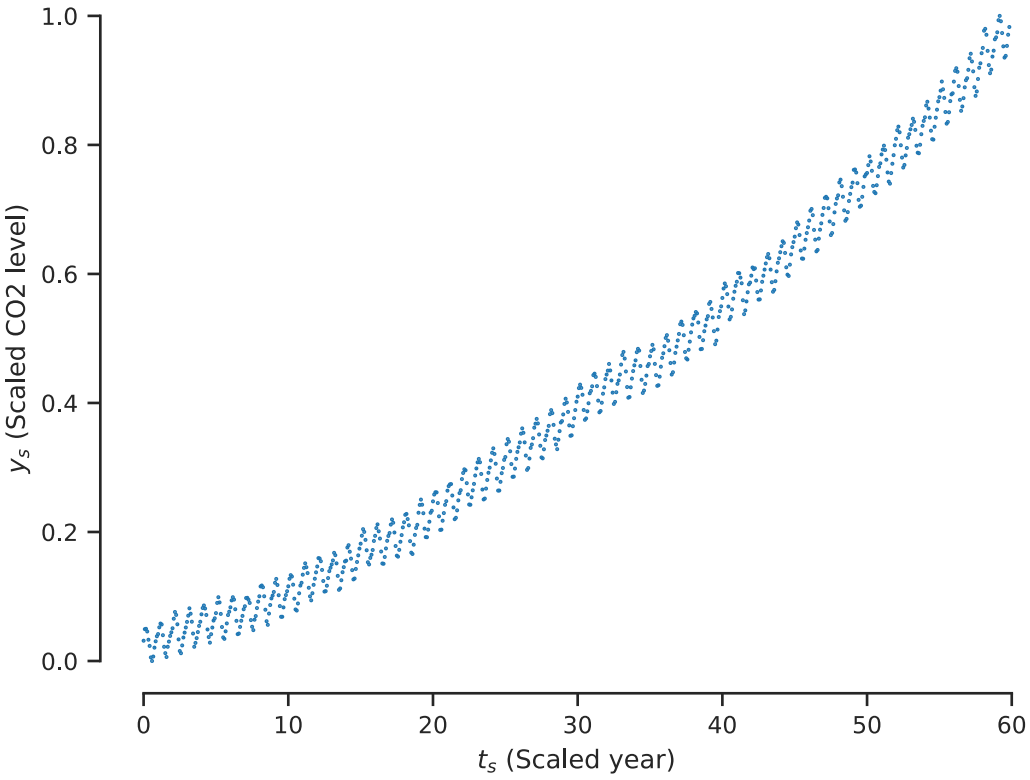
So, time is still in fractional years, but we start counting at zero instead of 1950. We scale the y's as:

$$y_s = \frac{y - y_{\text{min}}}{y_{\text{max}} - y_{\text{min}}}.$$

This takes all the y between 0 and 1. Here is what the scaled data look like:

[ ]:
```
t_s = t - t.min()
y_s = (y - y.min()) / (y.max() - y.min())
fig, ax = plt.subplots(1, 1)
```

```
ax.plot(t_s, y_s, '.', markersize=1)
ax.set_xlabel('$t_s$ (Scaled year)')
ax.set_ylabel('$y_s$ (Scaled CO2 level)')
sns.despine(trim=True);
```



Work with the scaled data in what follows as you develop your model. Scale back to the original units for your final predictions.

1.6 Part A - Naive approach

Use a zero mean Gaussian process with a squared exponential covariance function to fit the data and make the required prediction (ten years after the last observation).

Answer:

Again, this is done for you so that you have a concrete example of what is requested.

[ ]:
```
cov_module = ScaleKernel(RBFKernel())
mean_module = gpytorch.means.ConstantMean()
train_x = torch.from_numpy(t_s).float()
train_y = torch.from_numpy(y_s).float()
naive_model = ExactGP(
    train_x,
```

```
    train_y,
    mean_module=mean_module,
    covar_module=cov_module
)
train(naive_model, train_x, train_y)
```

```
tensor(0.8545, grad_fn=<NegBackward0>)
tensor(0.7392, grad_fn=<NegBackward0>)
tensor(-0.5164, grad_fn=<NegBackward0>)
tensor(-1.7353, grad_fn=<NegBackward0>)
tensor(-2.1123, grad_fn=<NegBackward0>)
tensor(-2.2573, grad_fn=<NegBackward0>)
tensor(-2.0044, grad_fn=<NegBackward0>)
tensor(-2.2871, grad_fn=<NegBackward0>)
tensor(-2.3023, grad_fn=<NegBackward0>)
tensor(-2.3135, grad_fn=<NegBackward0>)
tensor(-2.3297, grad_fn=<NegBackward0>)
tensor(-2.3330, grad_fn=<NegBackward0>)
tensor(-2.2609, grad_fn=<NegBackward0>)
tensor(-2.3376, grad_fn=<NegBackward0>)
tensor(-2.3396, grad_fn=<NegBackward0>)
tensor(-2.3418, grad_fn=<NegBackward0>)
tensor(-2.3452, grad_fn=<NegBackward0>)
tensor(-2.3467, grad_fn=<NegBackward0>)
tensor(-2.3478, grad_fn=<NegBackward0>)
tensor(-2.3483, grad_fn=<NegBackward0>)
tensor(-2.3508, grad_fn=<NegBackward0>)
tensor(-2.3504, grad_fn=<NegBackward0>)
tensor(-2.3522, grad_fn=<NegBackward0>)
tensor(-2.3530, grad_fn=<NegBackward0>)
tensor(-2.3531, grad_fn=<NegBackward0>)
Iter   1/10 - Loss: 0.854
tensor(-2.3531, grad_fn=<NegBackward0>)
tensor(-2.3538, grad_fn=<NegBackward0>)
tensor(-2.3537, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3540, grad_fn=<NegBackward0>)
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3540, grad_fn=<NegBackward0>)
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
Iter   2/10 - Loss: -2.353
tensor(-2.3542, grad_fn=<NegBackward0>)
```

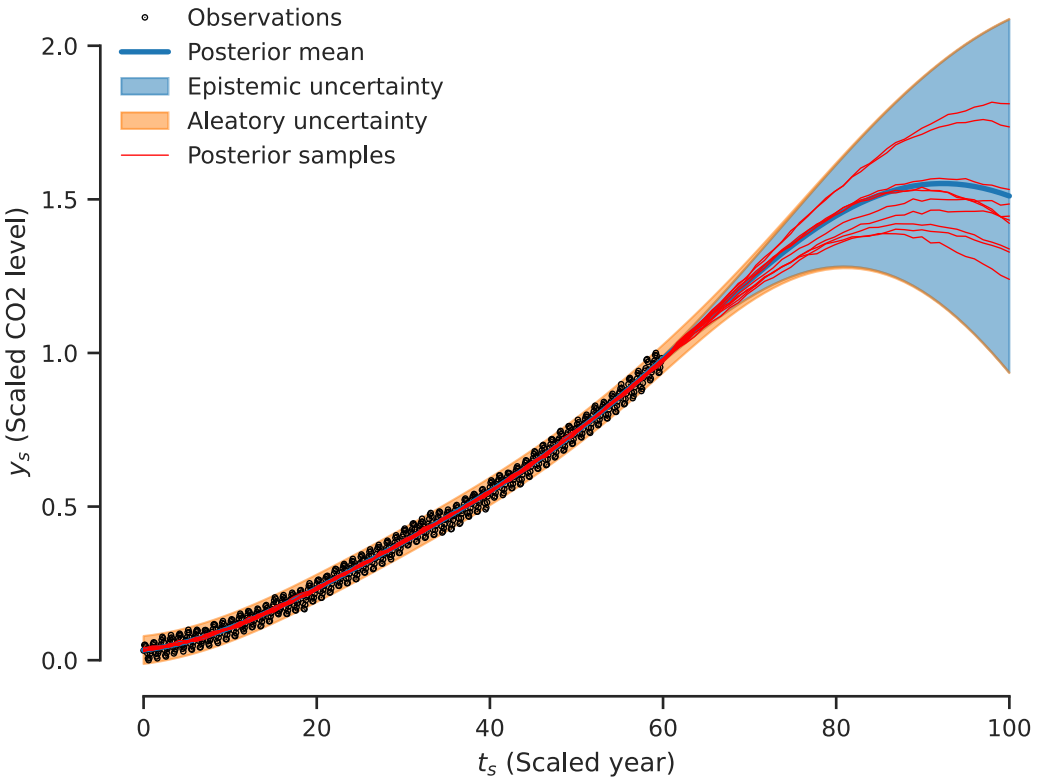
```
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3540, grad_fn=<NegBackward0>)
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3540, grad_fn=<NegBackward0>)
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
Iter   3/10 - Loss: -2.354
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3540, grad_fn=<NegBackward0>)
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3540, grad_fn=<NegBackward0>)
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
Iter   4/10 - Loss: -2.354
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3540, grad_fn=<NegBackward0>)
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3540, grad_fn=<NegBackward0>)
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
Iter   5/10 - Loss: -2.354
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3540, grad_fn=<NegBackward0>)
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3540, grad_fn=<NegBackward0>)
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
Iter   6/10 - Loss: -2.354
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3540, grad_fn=<NegBackward0>)
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3540, grad_fn=<NegBackward0>)
```

```
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
Iter   7/10 - Loss: -2.354
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3540, grad_fn=<NegBackward0>)
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3540, grad_fn=<NegBackward0>)
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
Iter   8/10 - Loss: -2.354
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3540, grad_fn=<NegBackward0>)
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3540, grad_fn=<NegBackward0>)
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
Iter   9/10 - Loss: -2.354
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3540, grad_fn=<NegBackward0>)
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3540, grad_fn=<NegBackward0>)
tensor(-2.3541, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
tensor(-2.3542, grad_fn=<NegBackward0>)
Iter  10/10 - Loss: -2.354
```

Predict everything:

[ ]:
```
x_star = torch.linspace(0, 100, 100)
plot_1d_regression(model=naive_model,
                   x_star=x_star,
                   xlabel='$t_s$ (Scaled year)',
                   ylabel='$y_s$ (Scaled CO2 level)');
```

```
/usr/local/lib/python3.11/dist-packages/linear_operator/utils/cholesky.py:40:
NumericalWarning: A not p.d., added jitter of 1.0e-06 to the diagonal
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/linear_operator/utils/cholesky.py:40:
NumericalWarning: A not p.d., added jitter of 1.0e-05 to the diagonal
  warnings.warn(
```



Notice that the squared exponential covariance captures the long terms but fails to capture the seasonal fluctuations. The seasonal fluctuations are treated as noise. This is wrong. You will have to fix this in the next part.

1.7 Part B - Improving the prior covariance

Now, use the ideas of Problem 1 to develop a covariance function that exhibits the following characteristics visible in the data (call $f(x)$ the scaled CO2 level. $+ f(x)$ is smooth. $+ f(x)$ has a clear trend with a multi-year length scale. $+ f(x)$ has seasonal fluctuations with a period of one year. $+ f(x)$ exhibits small fluctuations within its period.

There is more than one correct answer.

Answer:

Covariance:

To model the long-term multi-year trend, we will use the RBF kernel with a length scale of $\ell = 30$.

To model the seasonal fluctuations, we will use the product of a periodic kernel (with a period of 1 year) and an RBF kernel (which allows the shape of the seasonality to evolve). This forms a locally periodic kernel. Multiplying these kernels allows the seasonal pattern to be slightly different from year to year. This is a more realistic representation of actual real-world fluctuations.

To model the small fluctuations within the period, we will use the Matérn kernel. Since the problem statement says that $f(x)$ is smooth, we will use $\nu = 1.5$.

When choosing the overall covariance, we add the long-term kernel, seasonal fluctuation kernel, and small fluctuation kernel.

Mean:

Since we know that $f(x)$ has a monotonic increasing trend, we will use a linear mean function.

[ ]:
```
# your code here
from gpytorch.kernels import ScaleKernel,RBFKernel,PeriodicKernel,MaternKernel

# Model the long-term trend of the data
k_long_term = ScaleKernel(RBFKernel())
k_long_term.base_kernel.lengthscale = 30.0
k_long_term.outputscale = 1.0

# Model the seasonal fluctuations with a period of 1 year
# PeriodicKernel() allows the seasonal fluctuations to occur in periods
# RBFKernel() allows seasonal fluctuations to vary slightly from year to year
seasonal_fluc = PeriodicKernel()
seasonal_fluc.period_length = 1.0
seasonal_fluc.lengthscale = 0.5

seasonal_modulation = RBFKernel()
seasonal_modulation.lengthscale = 20.0

k_seasonal_fluc = ScaleKernel(seasonal_fluc * seasonal_modulation)
k_seasonal_fluc.outputscale = 0.2

# Model the small fluctuations within the period
k_small_fluc = ScaleKernel(MaternKernel(nu=1.5))
k_small_fluc.base_kernel.lengthscale = 0.1
k_small_fluc.outputscale = 0.1

# Your choice of covariance here
cov_module = k_long_term + k_seasonal_fluc + k_small_fluc

# Your choice of mean here
mean_module = gpytorch.means.LinearMean(input_size=1)

model = ExactGP(
```

```
    train_x,
    train_y,
    mean_module=mean_module,
    covar_module=cov_module
)

# Add a nugget to ensure the covariance is positive definite
with gpytorch.settings.cholesky_jitter(1e-4):
  train(model, train_x, train_y)
```

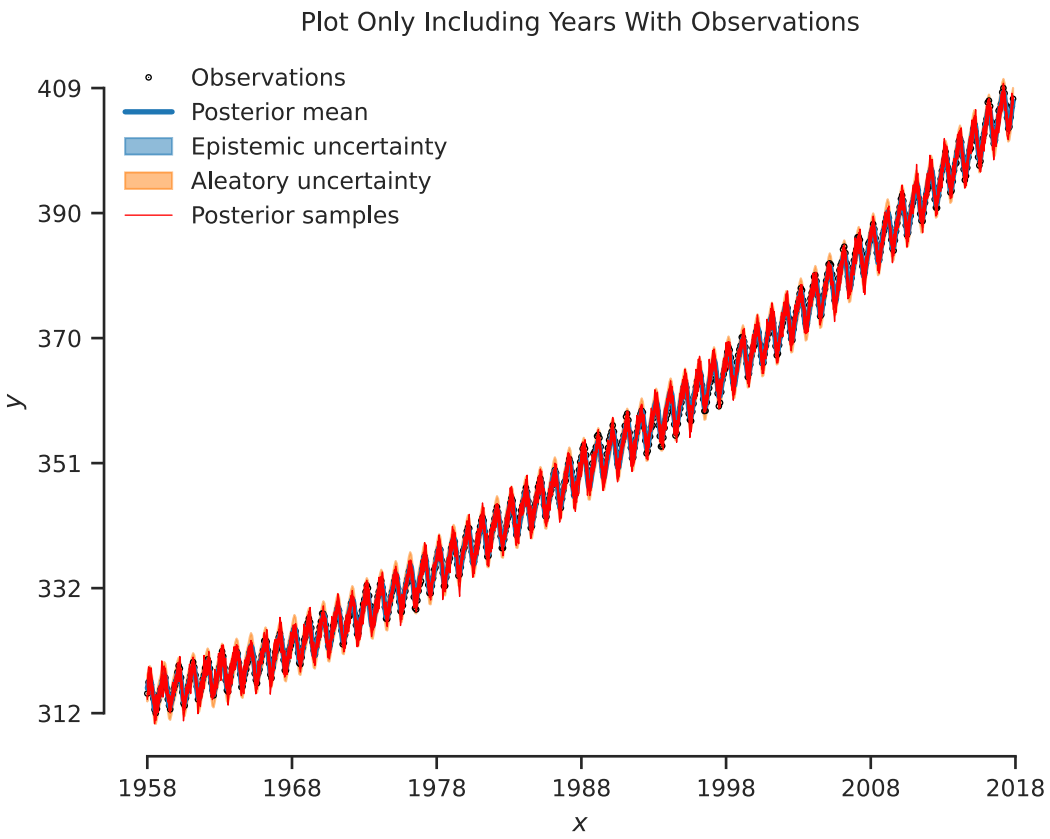
```
tensor(-0.3724, grad_fn=<NegBackward0>)
tensor(-0.8127, grad_fn=<NegBackward0>)
tensor(-1.9511, grad_fn=<NegBackward0>)
tensor(24.5765, grad_fn=<NegBackward0>)
tensor(-2.0222, grad_fn=<NegBackward0>)
tensor(-2.0287, grad_fn=<NegBackward0>)
tensor(-2.0705, grad_fn=<NegBackward0>)
tensor(-2.0962, grad_fn=<NegBackward0>)
tensor(-2.2740, grad_fn=<NegBackward0>)
tensor(-2.2894, grad_fn=<NegBackward0>)
tensor(-2.2937, grad_fn=<NegBackward0>)
tensor(-2.2955, grad_fn=<NegBackward0>)
tensor(-2.3055, grad_fn=<NegBackward0>)
tensor(-2.3134, grad_fn=<NegBackward0>)
tensor(-3.3724, grad_fn=<NegBackward0>)
tensor(-2.5620, grad_fn=<NegBackward0>)
tensor(-3.3683, grad_fn=<NegBackward0>)
tensor(-3.3887, grad_fn=<NegBackward0>)
tensor(-3.4085, grad_fn=<NegBackward0>)
tensor(-3.4288, grad_fn=<NegBackward0>)
tensor(-3.4138, grad_fn=<NegBackward0>)
tensor(-3.4310, grad_fn=<NegBackward0>)
tensor(-3.4317, grad_fn=<NegBackward0>)
tensor(-3.4321, grad_fn=<NegBackward0>)
tensor(-3.4326, grad_fn=<NegBackward0>)
Iter   1/10 - Loss: -0.372
tensor(-3.4326, grad_fn=<NegBackward0>)
tensor(-3.4312, grad_fn=<NegBackward0>)
tensor(-3.4315, grad_fn=<NegBackward0>)
tensor(-3.4323, grad_fn=<NegBackward0>)
tensor(-3.4326, grad_fn=<NegBackward0>)
Iter   2/10 - Loss: -3.433
tensor(-3.4326, grad_fn=<NegBackward0>)
tensor(-3.4312, grad_fn=<NegBackward0>)
tensor(-3.4315, grad_fn=<NegBackward0>)
tensor(-3.4323, grad_fn=<NegBackward0>)
tensor(-3.4326, grad_fn=<NegBackward0>)
```

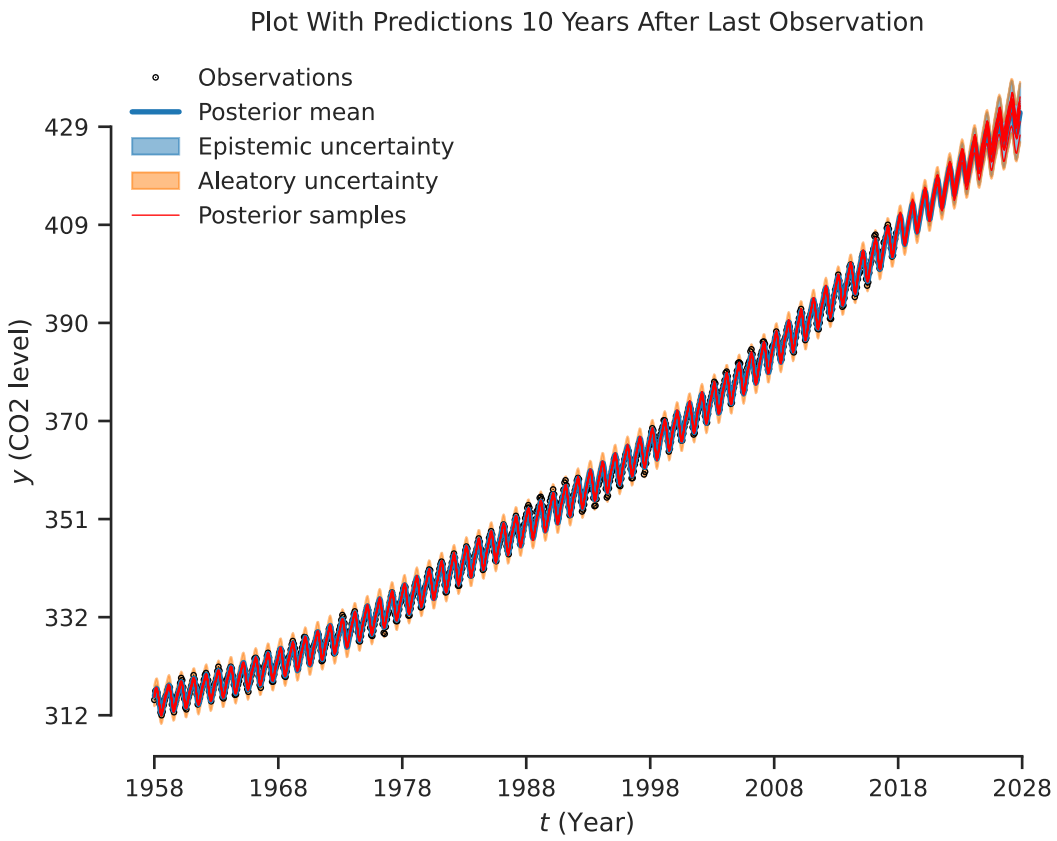
```
Iter   3/10 - Loss: -3.433
tensor(-3.4326, grad_fn=<NegBackward0>)
tensor(-3.4312, grad_fn=<NegBackward0>)
tensor(-3.4315, grad_fn=<NegBackward0>)
tensor(-3.4323, grad_fn=<NegBackward0>)
tensor(-3.4326, grad_fn=<NegBackward0>)
Iter   4/10 - Loss: -3.433
tensor(-3.4326, grad_fn=<NegBackward0>)
tensor(-3.4312, grad_fn=<NegBackward0>)
tensor(-3.4315, grad_fn=<NegBackward0>)
tensor(-3.4323, grad_fn=<NegBackward0>)
tensor(-3.4326, grad_fn=<NegBackward0>)
Iter   5/10 - Loss: -3.433
tensor(-3.4326, grad_fn=<NegBackward0>)
tensor(-3.4312, grad_fn=<NegBackward0>)
tensor(-3.4315, grad_fn=<NegBackward0>)
tensor(-3.4323, grad_fn=<NegBackward0>)
tensor(-3.4326, grad_fn=<NegBackward0>)
Iter   6/10 - Loss: -3.433
tensor(-3.4326, grad_fn=<NegBackward0>)
tensor(-3.4312, grad_fn=<NegBackward0>)
tensor(-3.4315, grad_fn=<NegBackward0>)
tensor(-3.4323, grad_fn=<NegBackward0>)
tensor(-3.4326, grad_fn=<NegBackward0>)
Iter   7/10 - Loss: -3.433
tensor(-3.4326, grad_fn=<NegBackward0>)
tensor(-3.4312, grad_fn=<NegBackward0>)
tensor(-3.4315, grad_fn=<NegBackward0>)
tensor(-3.4323, grad_fn=<NegBackward0>)
tensor(-3.4326, grad_fn=<NegBackward0>)
Iter   8/10 - Loss: -3.433
tensor(-3.4326, grad_fn=<NegBackward0>)
tensor(-3.4312, grad_fn=<NegBackward0>)
tensor(-3.4315, grad_fn=<NegBackward0>)
tensor(-3.4323, grad_fn=<NegBackward0>)
tensor(-3.4326, grad_fn=<NegBackward0>)
Iter   9/10 - Loss: -3.433
tensor(-3.4326, grad_fn=<NegBackward0>)
tensor(-3.4312, grad_fn=<NegBackward0>)
tensor(-3.4315, grad_fn=<NegBackward0>)
tensor(-3.4323, grad_fn=<NegBackward0>)
tensor(-3.4326, grad_fn=<NegBackward0>)
Iter  10/10 - Loss: -3.433
```

Plot using the following block:

[ ]:
```
# Plot the training data
plot_1d_regression(model=model, x_star=train_x);
```

```
plt.title('Plot Only Including Years With Observations')

# Unscale the data
xticks = plt.xticks()[0] # Get current tick locations
plt.xticks(xticks, [int(t + 1958) for t in xticks]);
y_max = y.max()
y_min = y.min()
yticks = plt.yticks()[0] # Get current tick locations
plt.yticks(yticks, [int(y * (y_max - y_min) + y_min) for y in yticks]);

# Plot the predictions 10 years after the last observation
x_star = torch.linspace(0, max(train_x) + 10, 1000)
plot_1d_regression(model=model, x_star=x_star,
                   xlabel='$t$ (Year)', ylabel='$y$ (CO2 level)');
plt.title('Plot With Predictions 10 Years After Last Observation')

# Unscale the data
xticks = plt.xticks()[0] # Get current tick locations
plt.xticks(xticks, [int(t + 1958) for t in xticks]);
y_max = y.max()
y_min = y.min()
yticks = plt.yticks()[0] # Get current tick locations
plt.yticks(yticks, [int(y * (y_max - y_min) + y_min) for y in yticks]);
```


Plot Only Including Years With Observations
Observations
Posterior mean
Epistemic uncertainty
Aleatory uncertainty
Posterior samples
409
390
370
351
332
312
y
1958
1968
1978
1988
1998
2008
2018
x



1.8 Part C - Predicting the future

How does your model predict the future? Why is it better than the naive model?

Answer: My model outperforms the naive model in predicting the future. It accurately captures the seasonal fluctuations, which can be seen by the zig-zag pattern on the plots. In contrast, the naive model wrongly treats the fluctuations as noise.

In the 10-year forecast, my model continues the same trend as the observed data. Specifically, the future predictions display an increasing trend with the same slope as the training data. Additionally, the future predictions have a zig-zag pattern, similar to the previously observed data.

On the contrary, the naive model does not display the correct trend for the future. Although the CO2 level is monotonically increasing, the naive model predicts an increase, following by a decrease. Moreover, the future predictions in the naive model do not include any seasonal fluctuations, which does not match what we expect.

The differences between the models can be explained by the choice of the mean and covariance. My model uses periodic and Matérn kernels to model the fluctuations, which the naive model lacks. My model uses a linear mean to capture the increasing trend, whereas the naive model uses a constant mean.

1.9 Part D - Bayesian information criterion

As we have seen in earlier lectures, the Bayesian informationc criterion (BIC), see this, can bse used to compare two models. The criterion says that one should: + fit the models with maximum likelihood, + and compute the quantity:

$$\text{BIC} = d\ln(n) - 2\ln(\hat{L}),$$

where d is the number of model parameters, and $\hat{L}$ the maximum likelihood. + pick the model with the smallest BIC.

Use BIC to show that the model you constructed in Part C is indeed better than the naïve model of Part A.

Answer:

[ ]:
```
# Hint: You can find the parameters of a model like this
list(naive_model.hyperparameters())
```

[ ]:
```
[Parameter containing:
 tensor([-7.8269], requires_grad=True),
 Parameter containing:
 tensor(0.5107, requires_grad=True),
 Parameter containing:
 tensor(-0.7554, requires_grad=True),
 Parameter containing:
 tensor([[34.6343]], requires_grad=True)]
```

[ ]:
```
m = sum(p.numel() for p in naive_model.hyperparameters())
print(m)
```

```
4
```

[ ]:
```
# Hint: You can find the (marginal) log likelihood of a model like this
mll = gpytorch.mlls.ExactMarginalLogLikelihood(naive_model.likelihood,␣
 ↪naive_model)
log_like = mll(naive_model(train_x), train_y)
print(log_like)
```

```
tensor(2.3860, grad_fn=<DivBackward0>)
```

[ ]:
```
# Hint: The BIC is
bic = -2 * log_like + m * np.log(train_x.shape[0])
print(bic)
```

```
tensor(21.5394, grad_fn=<AddBackward0>)
```

[ ]:
```
# Your code here
def bic_score(trained_model, x, y):
```

```
    # Number of free hyper-parameters
    m = sum(p.numel() for p in trained_model.hyperparameters())

    # Marginal log-likelihood
    mll = gpytorch.mlls.ExactMarginalLogLikelihood(trained_model.likelihood,␣
 ↪trained_model)
    log_like = mll(trained_model(x), y).item()

    # Compute the BIC
    bic = -2 * log_like + m * np.log(x.shape[0])
    return m, log_like, bic

bic_naive = bic_score(naive_model, train_x, train_y)
bic_mymodel = bic_score(model, train_x, train_y)

print(f"Number of parameters for the naive model: {bic_naive[0]}")
print(f"Number of parameters for my model: {bic_mymodel[0]}\n")

print(f"Log-likelihood for the naive model: {bic_naive[1]:.2f}")
print(f"Log-likelihood for my model: {bic_mymodel[1]:.2f}\n")

print(f"BIC for the naive model: {bic_naive[2]:.2f}")
print(f"BIC for my model: {bic_mymodel[2]:.2f}")
```

```
Number of parameters for the naive model: 4
Number of parameters for my model: 11

Log-likelihood for the naive model: 2.39
Log-likelihood for my model: 3.52

BIC for the naive model: 21.54
BIC for my model: 65.31
```

Although my model achieves a higher log-likelihood than the naive model (3.52 vs. 2.39), it also has more parameters (11 vs. 4). As a result, my model's BIC score is worse (65.31 vs. 21.54). Lower BIC values indicate a better trade-off between fit and complexity, meaning the naive model is preferred by this metric. However, it is important to note that the BIC difference between the two models is not very large.

Real-world time series often have complex patterns that simple models cannot represent. However, the BIC penalizes model complexity heavily and may favor overly simplistic models. Despite the higher BIC, my model is qualitatively better at capturing the features of the data, such as the seasonal structure and realistic future trend. In contrast, the simpler naive model cannot accurately capture the seasonal fluctuations and fails to accurately predict the future. This shows that the increased complexity is necessary in this problem.

1.10 Problem 3 - Bayesian Global Optimization

As a toy example, we will apply Bayesian Optimization to some synthetic data. We will study the classic Forrester function

$$f(x) = (6x - 2)^2 \sin(12x - 4)$$

on the domain $[0, 1]$. We will also *standardize* the output of the function, such that it has a mean of 0 and a standard deviation of 1. This is a good habit to get into when working with Gaussian processes. We will stick to a zero mean prior, so ensuring that the data has a mean of zero aligns with this.

The mean and standard deviation of this function on $[0, 1]$ are known:

$$\mu = 0.45321$$
$$\text{std} = 4.4248$$

The goal is to find the minimum of this objective function.

1.10.1 Part A - Visualize the function and generate some data

Let's visualize the ground truth objective function and our synthetic data. First, code the **standardized** Forrester function in a way that allows for **minimization** using our Bayesian global **maximization** algorithms from the lecture book.

(Hint: to minimize a function, you can maximize the negative of that function)

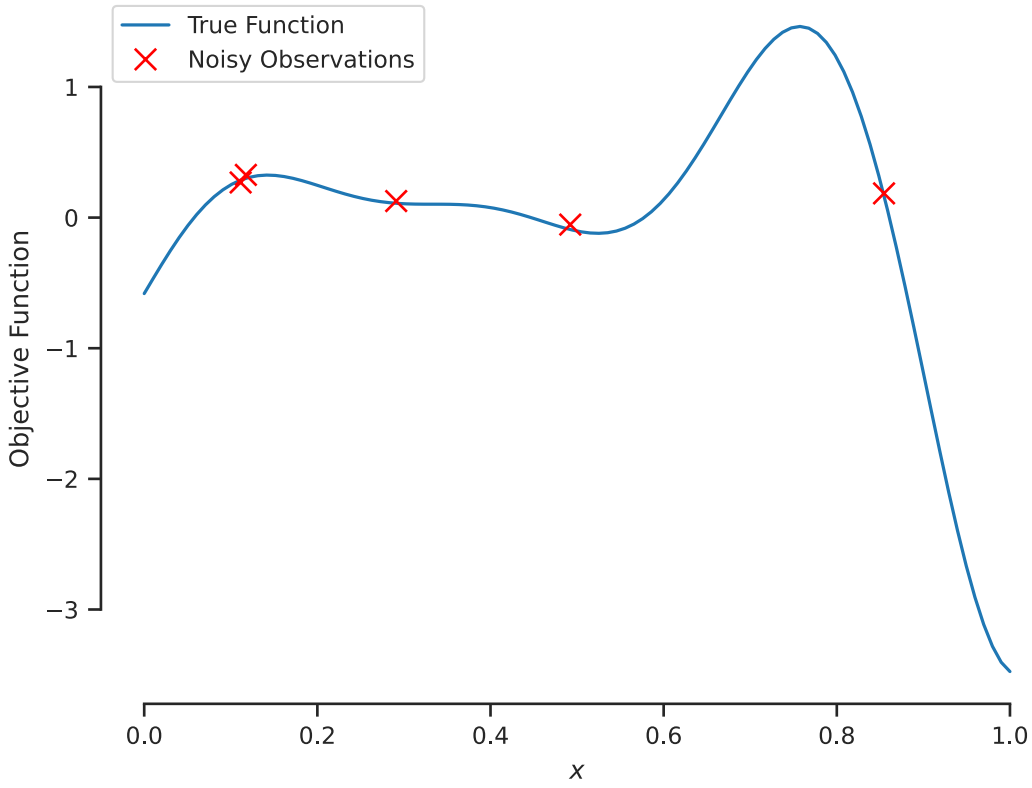
[ ]:
```
# your code here
def Forrester(x):
    """ground truth function to optimize"""

    mu = 0.45321
    std = 4.4248

    # Define the raw function
    def raw_forrester(x):
        return (6 * x - 2) ** 2 * np.sin(12 * x - 4)

    # Standardize the Forrester function
    standardized_forrester = (raw_forrester(x) - mu) / std

    # The objective function to be maximized
    objective_function = -standardized_forrester
    return objective_function
```

[ ]:
```
# making synthetic data from your function

np.random.seed(539)
```

```
sigma_noise = 0.025

# noisy version of the above function
F_noisy = lambda x: (
    Forrester(x)
    + sigma_noise * np.random.randn(x.shape[0])
)

# generate synthetic data
n_init = 5
X = np.random.rand(n_init)
Y = F_noisy(X)

train_x = torch.from_numpy(X).float()
train_y = torch.from_numpy(Y).float()
```

Plot it on $[0, 1]$ and make sure to include the data points

[ ]:
```
# your code here
xs = np.linspace(0, 1, 100)
ys = Forrester(xs)

fig, ax = plt.subplots()
ax.plot(xs, ys, label='True Function')
ax.plot(X, Y, 'rx', ms=8, label='Noisy Observations')
ax.set_xlabel('$x$')
ax.set_ylabel('Objective Function')
ax.tick_params(bottom=True, left=True)
ax.legend(loc='best')
sns.despine(trim=True)
```



1.10.2 Part B - Set up the Gaussian process model

Set up the Gaussian process model.

Specifically, use this:

1. A Matern covariance kernel
2. Zero mean function
3. A Gaussian likelihood model
4. Set the likelihood noise to the ground truth noise (since we assume it is known)

[ ]:
```
# your code here
from gpytorch.kernels import MaternKernel, ScaleKernel
from gpytorch.means import ConstantMean

# Matern covariance kernel
cov_module = ScaleKernel(MaternKernel(nu=0.5))

# It is not a good idea to train the model when we do not have enough data
# So we fix the hyperparameters to something reasonable
cov_module.base_kernel.lengthscale = 0.2
cov_module.outputscale = 2.0
```

```
# Zero mean function
mean_module = ConstantMean()
mean_module.constant = 0

# Gaussian likelihood with noise equal to the ground truth noise
likelihood = gpytorch.likelihoods.GaussianLikelihood()
likelihood.noise = torch.tensor(sigma_noise ** 2)

# Create the model
model = ExactGP(
    train_x,
    train_y,
    mean_module=mean_module,
    covar_module=cov_module,
    likelihood=likelihood
)
```

1.10.3 Now train the model on the data points to optimize the rest of the hyperparameters

Here is the training function you should be using:

[ ]:
```
def train(model, train_x, train_y, n_iter=10, lr=0.1):
    """Train the model.

    Arguments
    model   --  The model to train.
    train_x --  The training inputs.
    train_y --  The training labels.
    n_iter  --  The number of iterations.
    """
    model.train()
    optimizer = torch.optim.LBFGS(model.parameters(),␣
 ↪line_search_fn='strong_wolfe')
    likelihood = model.likelihood
    mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)
    def closure():
        optimizer.zero_grad()
        output = model(train_x)
        loss = -mll(output, train_y)
        loss.backward()
        return loss
    for i in range(n_iter):
        loss = optimizer.step(closure)
        if (i + 1) % 1 == 0:
            print(f'Iter {i + 1:3d}/{n_iter} - Loss: {loss.item():.3f}')
```

```
    model.eval()
```

[ ]:
```
# your code here
train(model, train_x, train_y)
```

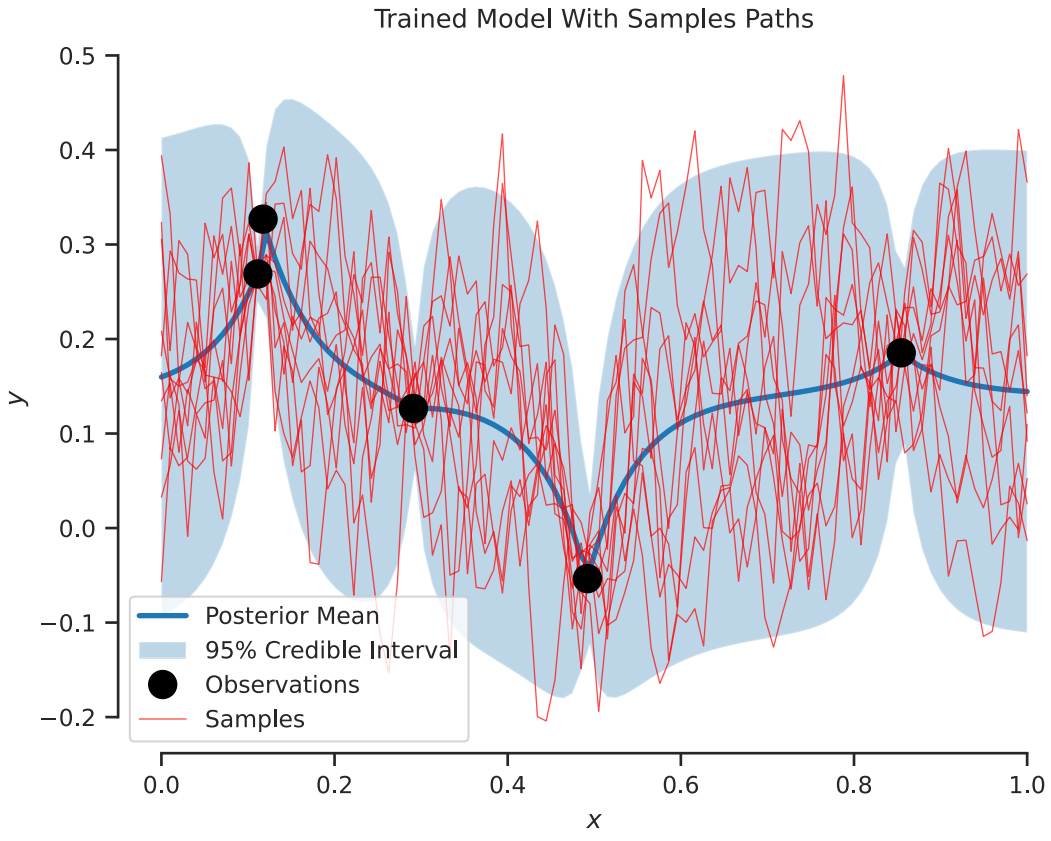
```
Iter   1/10 - Loss: 0.957
Iter   2/10 - Loss: -0.776
Iter   3/10 - Loss: -0.797
Iter   4/10 - Loss: -0.797
Iter   5/10 - Loss: -0.797
Iter   6/10 - Loss: -0.797
Iter   7/10 - Loss: -0.797
Iter   8/10 - Loss: -0.797
Iter   9/10 - Loss: -0.797
Iter  10/10 - Loss: -0.797
```

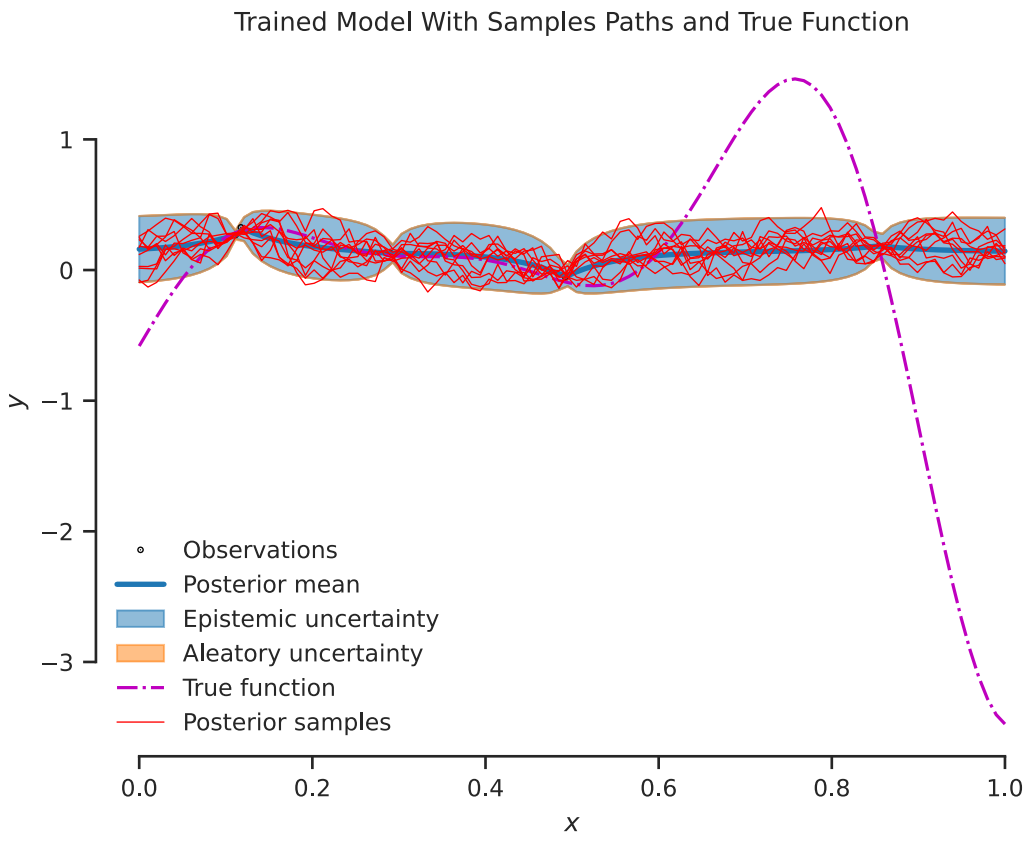
1.10.4 Plot the trained model along with some sample paths

[ ]:
```
# your code here
# Plot the trained model with sample paths
# Evaluate the posterior
xs = torch.linspace(0, 1, 100)
model.eval()
model.likelihood.eval()
with torch.no_grad(), gpytorch.settings.fast_pred_var():
    posterior = model(xs)
    mean_xs = posterior.mean
    std_xs = posterior.variance.sqrt()

# Plot the trained model
fig, ax = plt.subplots()
ax.plot(xs.numpy(), mean_xs.numpy(), lw=2, label='Posterior Mean')

# Plot the 95% credible interval
ax.fill_between(xs.numpy(), (mean_xs - 2 * std_xs).numpy(),
                (mean_xs + 2 * std_xs).numpy(), alpha=0.3,
                 label='95% Credible Interval')

# Plot the observed data points
ax.plot(train_x.numpy(), train_y.numpy(), 'k.',
        ms=20, zorder=3, label='Observations')

# Plot some posterior sample paths
samples = posterior.sample(sample_shape=torch.Size([10]))
for i, y in enumerate(samples.numpy()):
    label = 'Samples' if i == 0 else None  # Label only the first sample
    ax.plot(xs.numpy(), y, 'r', lw=0.5, alpha=0.7, label=label)
```

```
# Add title, labels, and legend
ax.set_title('Trained Model With Samples Paths')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
ax.tick_params(bottom=True, left=True)
ax.legend(loc='best')
sns.despine(trim=True)

# Also make a plot with the true function
plot_1d_regression(xs, model, f_true=Forrester, num_samples=10);
plt.title('Trained Model With Samples Paths and True Function');
```

```
/tmp/ipython-input-4-753138915.py:10: DeprecationWarning: __array_wrap__ must
accept context and return_scalar arguments (positionally) in the future.
(Deprecated NumPy 2.0)
  return (6 * x - 2) ** 2 * np.sin(12 * x - 4)
```

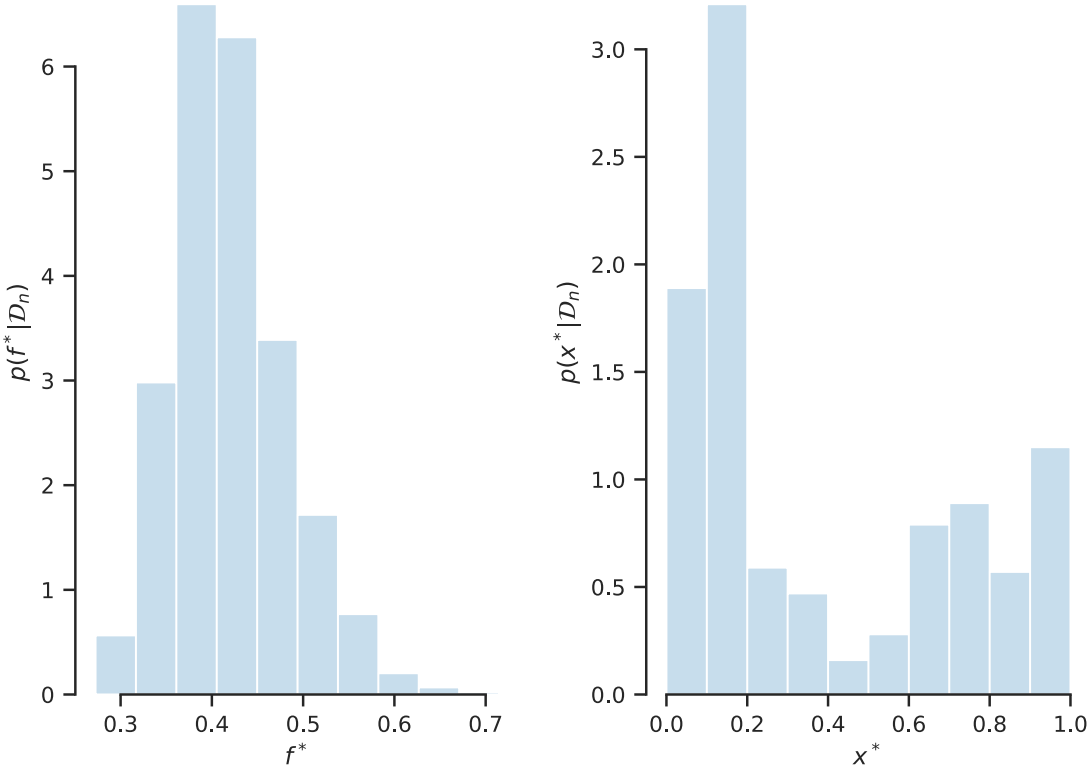




1.10.5 Plot the uncertainty about the optimization problem for the initial Gaussian process surrogate

[ ]:
```
# your code here
def plot_max_and_argmax(gpr, X_design, n_samples=1000):
    """Plot histograms of max/argmax of the function represented by model gpr.

    Arguments
    gpr      -- A trained Gaussian process object.
    X_design -- A set of points to evaluate the response on.

    Keyword Arguments
    n_samples -- The number of samples to take to make the histograms.
    """
    f_star = gpr(X_design)
    f_samples = f_star.sample(sample_shape=torch.Size([n_samples])).numpy()
    max_f_samples = np.max(f_samples, axis=1)
    x_star_samples = X_design.numpy()[np.argmax(f_samples, axis=1)]

    fig, ax = plt.subplots(1,2)
```

```
    ax[0].hist(max_f_samples, density=True, alpha=0.25)
    ax[0].set_xlabel('$f^*$')
    ax[0].set_ylabel('$p(f^*|\mathcal{D}_n)$')

    ax[1].hist(x_star_samples, density=True, alpha=0.25)
    ax[1].set_xlabel('$x^*$')
    ax[1].set_ylabel('$p(x^*|\mathcal{D}_n)$')

    plt.tight_layout()
    sns.despine(trim=True)

    return fig, ax

plot_max_and_argmax(model, xs);
```



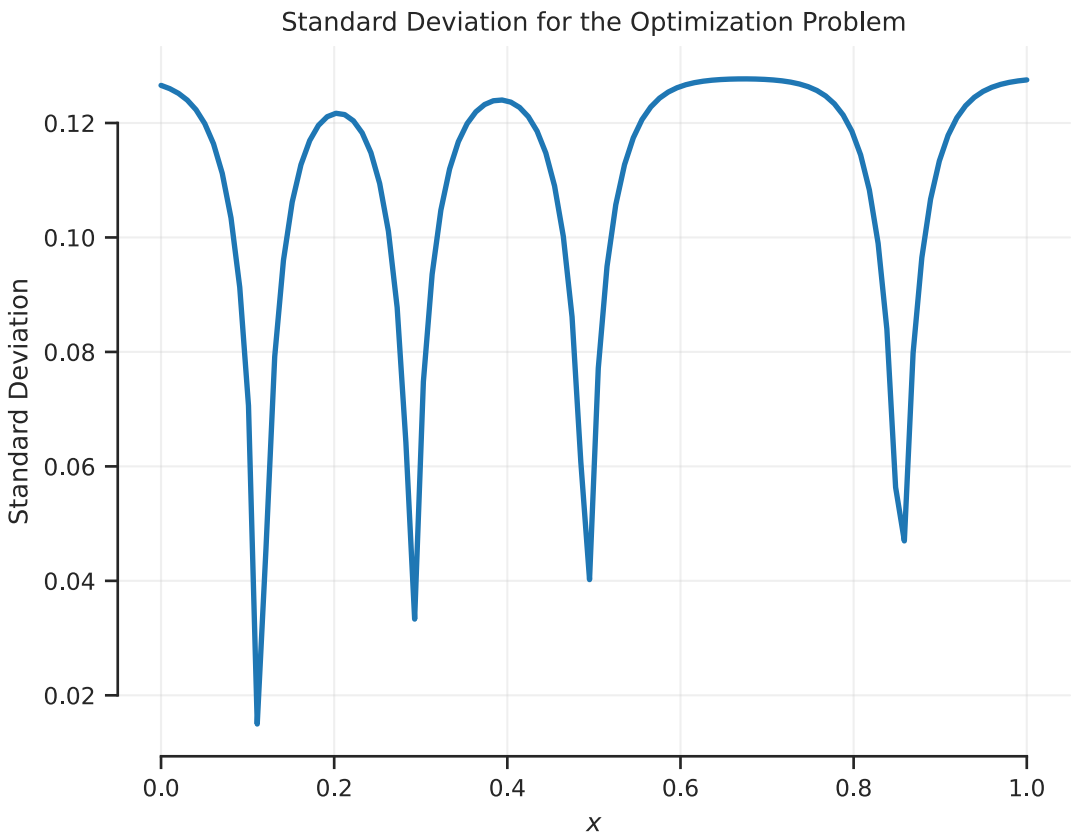
[ ]:
```
# Sample many functions from the surrogate GP
f_star = model(xs)
f_samples = f_star.sample(sample_shape=torch.Size([10000])).numpy()

# Find the maximum of each sample and its location
max_f_samples = np.max(f_samples, axis=1)
```

```
x_star_samples = xs.numpy()[np.argmax(f_samples, axis=1)]

# Find the mean and 95% credible interval for the location of the maximum
x_star_mean = np.mean(x_star_samples)
x_star_ci = np.percentile(x_star_samples, [2.5, 97.5])
print(f"Posterior mean of x*: {x_star_mean:.4f}")
print(f"95% credible interval: [{x_star_ci[0]:.4f}, {x_star_ci[1]:.4f}]\n")

# Find the mean and 95% credible interval for the value of the maximum
f_star_mean = np.mean(max_f_samples)
f_star_ci = np.percentile(max_f_samples, [2.5, 97.5])
print(f"Posterior mean of f*: {f_star_mean:.4f}")
print(f"95% credible interval: [{f_star_ci[0]:.4f}, {f_star_ci[1]:.4f}]")
```

```
Posterior mean of x*: 0.3907
95% credible interval: [0.0101, 0.9899]

Posterior mean of f*: 0.4197
95% credible interval: [0.3164, 0.5556]
```

From the above histograms and the 95% credible intervals, we can see that there is a lot of uncertainty about the location and value of the optimum point.

[ ]:
```
# Plot the standard deviation as a function of x
fig, ax = plt.subplots()
ax.plot(xs.numpy(), std_xs.numpy(), lw=2)
ax.set_title('Standard Deviation for the Optimization Problem')
ax.set_xlabel('$x$')
ax.set_ylabel('Standard Deviation')
plt.grid(alpha=0.3)
sns.despine(trim=True)
```



1.11 Part C - Expected improvement with noise

Solve the optimization problem by applying the expected improvement with noise algorithm

[ ]:
```
def plot_1d_regression(
    x_star,
    model,
    ax=None,
    f_true=None,
    num_samples=10
):
    """Plot the posterior predictive.

    Arguments
    x_start  --  The test points on which to evaluate.
    model    --  The trained model.

    Keyword Arguments
    ax          --  An axes object to write on.
    f_true      --  The true function.
```

```
    num_samples --  The number of samples.
    """
    f_star = model(x_star)
    m_star = f_star.mean
    v_star = f_star.variance
    y_star = model.likelihood(f_star)
    yv_star = y_star.variance

    f_lower = (
        m_star - 2.0 * torch.sqrt(v_star)
    )
    f_upper = (
        m_star + 2.0 * torch.sqrt(v_star)
    )

    y_lower = m_star - 2.0 * torch.sqrt(yv_star)
    y_upper = m_star + 2.0 * torch.sqrt(yv_star)

    if ax is None:
        fig, ax = plt.subplots()

    ax.plot(model.train_inputs[0].flatten().detach(),
            model.train_targets.detach(),
            'kx',
            markersize=10,
            markeredgewidth=2,
            label='Observations'
    )

    ax.plot(
        x_star,
        m_star.detach(),
        lw=2,
        label='$m_n(x)$',
        color=sns.color_palette()[0]
    )

    ax.fill_between(
        x_star.flatten().detach(),
        f_lower.flatten().detach(),
        f_upper.flatten().detach(),
        alpha=0.5,
        label='$f(\mathbf{x}^*)$ 95% pred.',
        color=sns.color_palette()[0]
    )

    ax.fill_between(
```

```
        x_star.detach().flatten(),
        y_lower.detach().flatten(),
        f_lower.detach().flatten(),
        color=sns.color_palette()[1],
        alpha=0.5,
        label='$y^*$ 95% pred.'
    )
    ax.fill_between(
        x_star.detach().flatten(),
        f_upper.detach().flatten(),
        y_upper.detach().flatten(),
        color=sns.color_palette()[1],
        alpha=0.5,
        label=None
    )


    if f_true is not None:
        ax.plot(
            x_star,
            f_true(x_star),
            'm-.',
            label='True function'
        )

    if num_samples > 0:
        f_post_samples = f_star.sample(
            sample_shape=torch.Size([10])
        )
        ax.plot(
            x_star.numpy(),
            f_post_samples.T.detach().numpy(),
            color="red",
            lw=0.5
        )
        # This is just to add the legend entry
        ax.plot(
            [],
            [],
            color="red",
            lw=0.5,
            label="Posterior samples"
        )

    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
```

```
    plt.legend(loc='lower left', frameon=False, bbox_to_anchor=(0.025, 0.08))
    sns.despine(trim=True)

    return m_star, v_star

def plot_iaf(
    x_star,
    gpr,
    alpha,
    alpha_params={},
    ax=None,
    f_true=None,
    iaf_label="Information Acquisition Function"
):
    """Plot the information acquisition function.

    Arguments
    x_star       -- A set of points to plot on.
    gpr          -- A rained Gaussian process regression
                    object.
    alpha        -- The information acquisition function.
                    This assumed to be a function of the
                    posterior mean and standard deviation.

    Keyword Arguments
    ax           -- An axes object to plot on.
    f_true       -- The true function - if available.
    alpha_params -- Extra parameters to the information
                    acquisition function.
    ax           -- An axes object to plot on.
    f_true       -- The true function - if available.
    iaf_label    -- The label for the information acquisition
                    function. Default is "Information Acquisition".

    The evaluation of the information acquisition function
    is as follows:

        af_values = alpha(mu, sigma, y_max, **alpha_params)

    """
    if ax is None:
        fig, ax = plt.subplots()

    ax.set_title(
        ", ".join(
            f"{n}={k:.2f}"
            for n, k in alpha_params.items()
```

```
            )
    )

    m, v = plot_1d_regression(
        x_star,
        gpr,
        ax=ax,
        f_true=f_true,
        num_samples=0
    )

    sigma = torch.sqrt(v)
    af_values = alpha(m, sigma, gpr.train_targets.numpy().max(), **alpha_params)
    next_id = torch.argmax(af_values)
    next_x = x_star[next_id]
    af_max = af_values[next_id]

    ax2 = ax.twinx()
    ax2.plot(x_star, af_values.detach(), color=sns.color_palette()[1])
    ax2.set_ylabel(
        iaf_label,
        color=sns.color_palette()[1]
    )
    plt.setp(
        ax2.get_yticklabels(),
        color=sns.color_palette()[1]
    )
    ax2.plot(
        next_x * np.ones(100),
        torch.linspace(0, af_max.item(), 100),
        color=sns.color_palette()[1],
        linewidth=1
    )

def ei(m, sigma, ymax):
    """Return the expected improvement.

    Arguments
    m     -- The predictive mean at the test points.
    sigma -- The predictive standard deviation at
             the test points.
    ymin  -- The minimum observed value (so far).
    """
    diff = m - ymax
    u = diff / sigma
    ei = ( diff * torch.distributions.Normal(0, 1).cdf(u) +
          sigma * torch.distributions.Normal(0, 1).log_prob(u).exp()
```

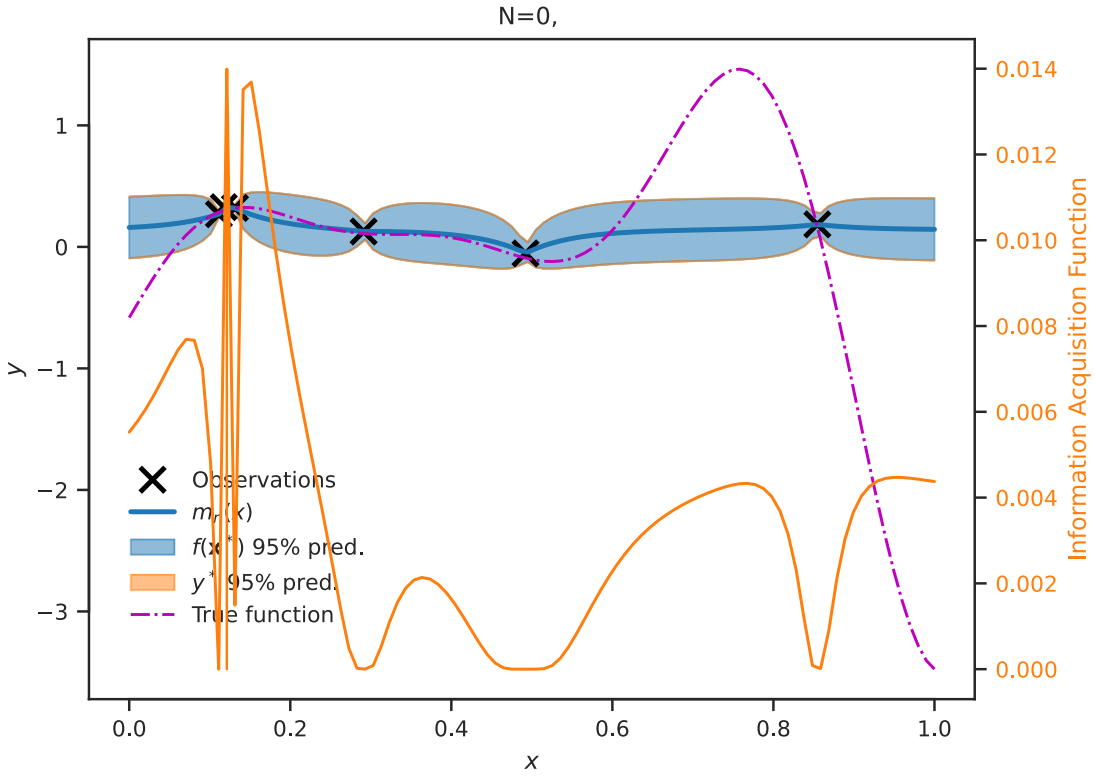
```
    )
    ei[sigma <= 0.] = 0.
    return ei

def maximize(
    f,
    model,
    X_design,
    alpha,
    alpha_params={},
    max_it=10,
    optimize=False,
    plot=False,
    **kwargs
):
    """Optimize a function using a limited number of evaluations.

    Arguments
    f             -- The function to optimize.
    gpr           -- A Gaussian process model to use for representing
                     our state of knowledge.
    X_design      -- The set of candidate points for identifying the
                     maximum.
    alpha         -- The information acquisition function.
                     This assumed to be a function of the
                     posterior mean and standard deviation.

    Keyword Arguments
    alpha_params -- Extra parameters to the information
                     acquisition function.
    max_it        -- The maximum number of iterations.
    optimize      -- Whether or not to optimize the hyper-parameters.
    plot          -- Determines how often to plot. Make it one
                     to plot at each iteration. Make it max_it
                     to plot at the last iteration.

    The rest of the keyword arguments are passed to plot_iaf().
    """
    af_all = []
    for count in range(max_it):
        # Predict
        f_design = model(X_design)
        m = f_design.mean
        sigma2 = f_design.variance
        sigma = torch.sqrt(sigma2)

        # Evaluate information acquisition function
```

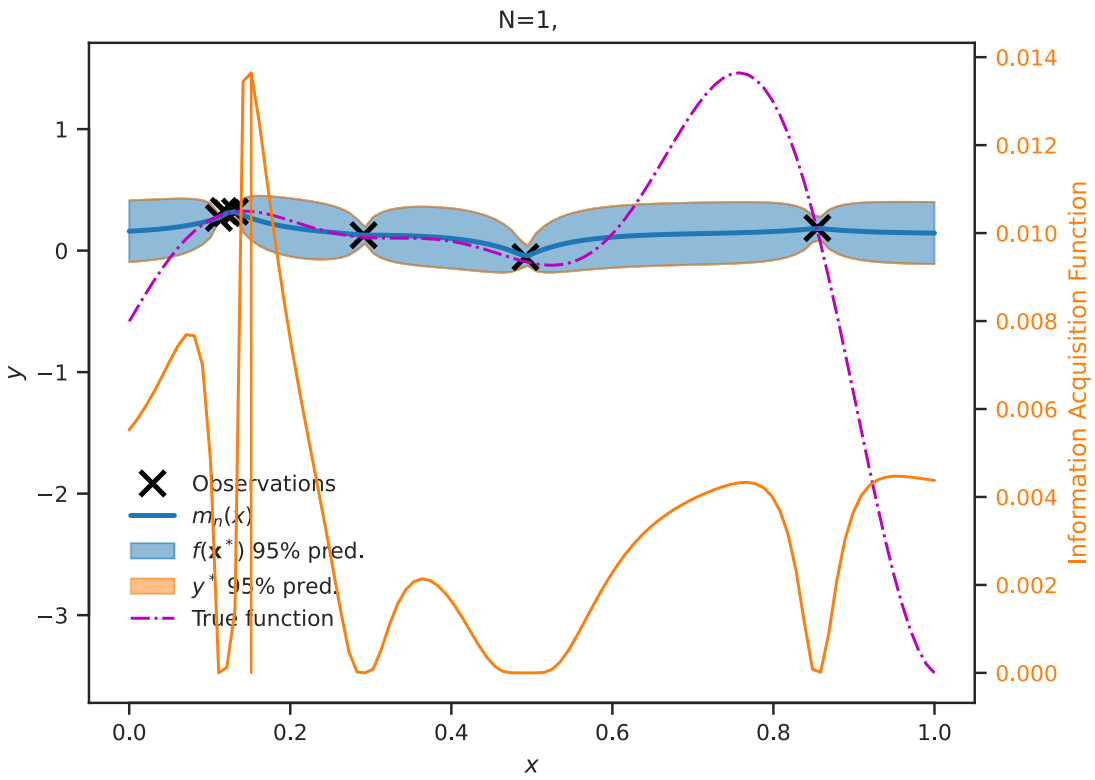
```
        y_train = model.train_targets.numpy()
        af_values = alpha(
            m,
            sigma,
            y_train.max(),
            **alpha_params
        )

        # Find best point to include
        i = torch.argmax(af_values)
        af_all.append(af_values[i])

        new_x = X_design[i:(i+1)].float()
        new_y = f(new_x)
        train_x = torch.cat([model.train_inputs[0], new_x[:, None]])
        train_y = torch.cat([model.train_targets, new_y])
        model.set_train_data(train_x, train_y, strict=False)

        if optimize:
            train(model, train_x, train_y, n_iter=100, lr=0.1)
        else:
            model.train()
            model.eval()

        # Plot if required
        if count % plot == 0:
            if "ax" in kwargs:
                ax = kwargs[ax]
            else:
                fig, ax = plt.subplots()
            plot_iaf(
                X_design,
                model,
                alpha,
                alpha_params=alpha_params,
                f_true=f,
                ax=ax,
                **kwargs
            )
            ax.set_title(
                f"N={count}, " + ax.get_title()
            )
    return af_all
```

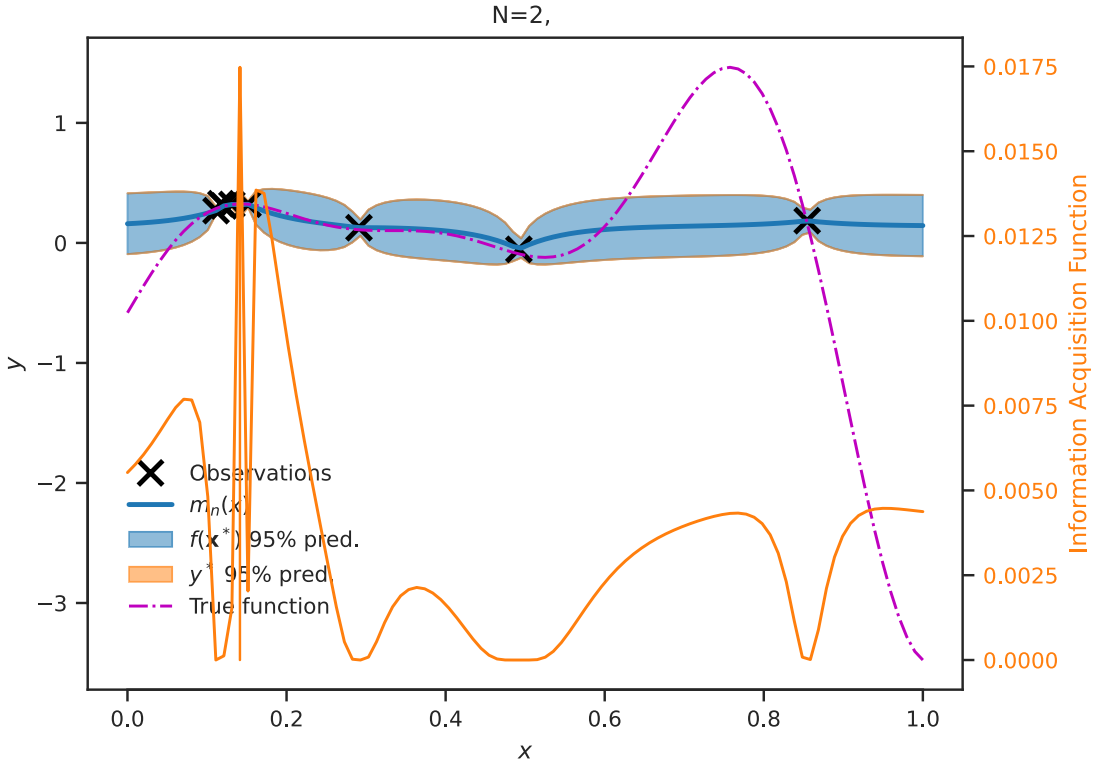
1.11.1 run the algorithm

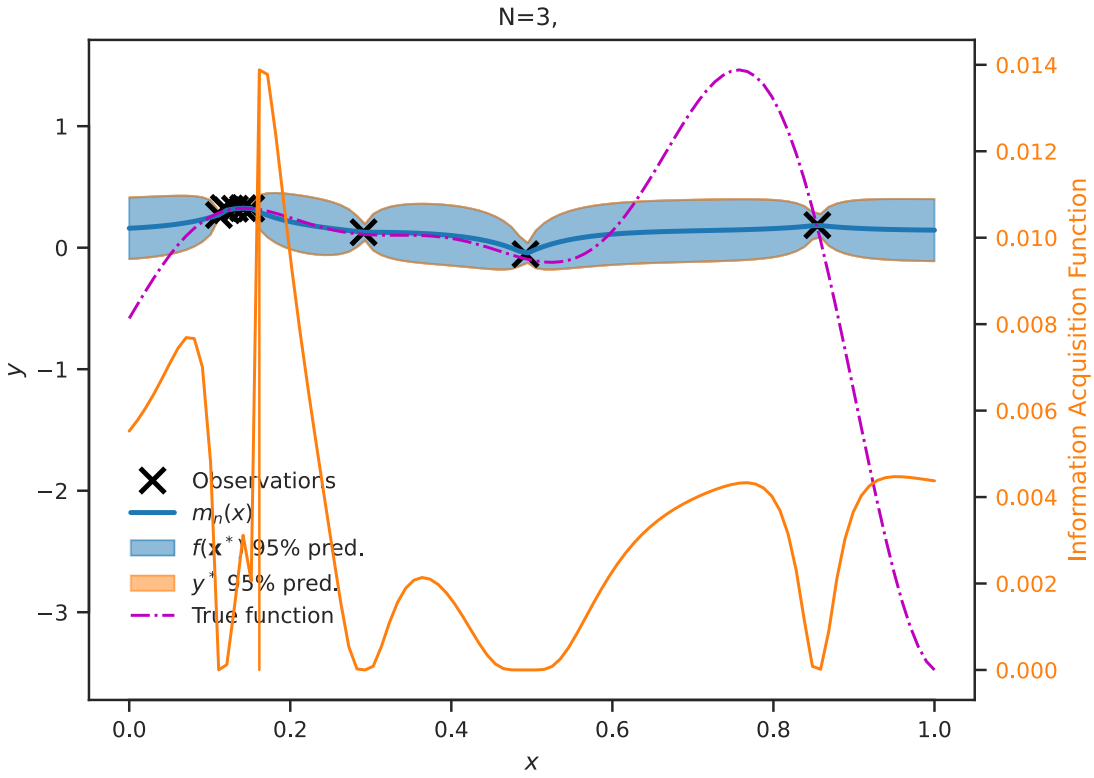
[ ]:
```
# your code here
maximize(Forrester, model, xs, ei, max_it=15, plot=1);
```

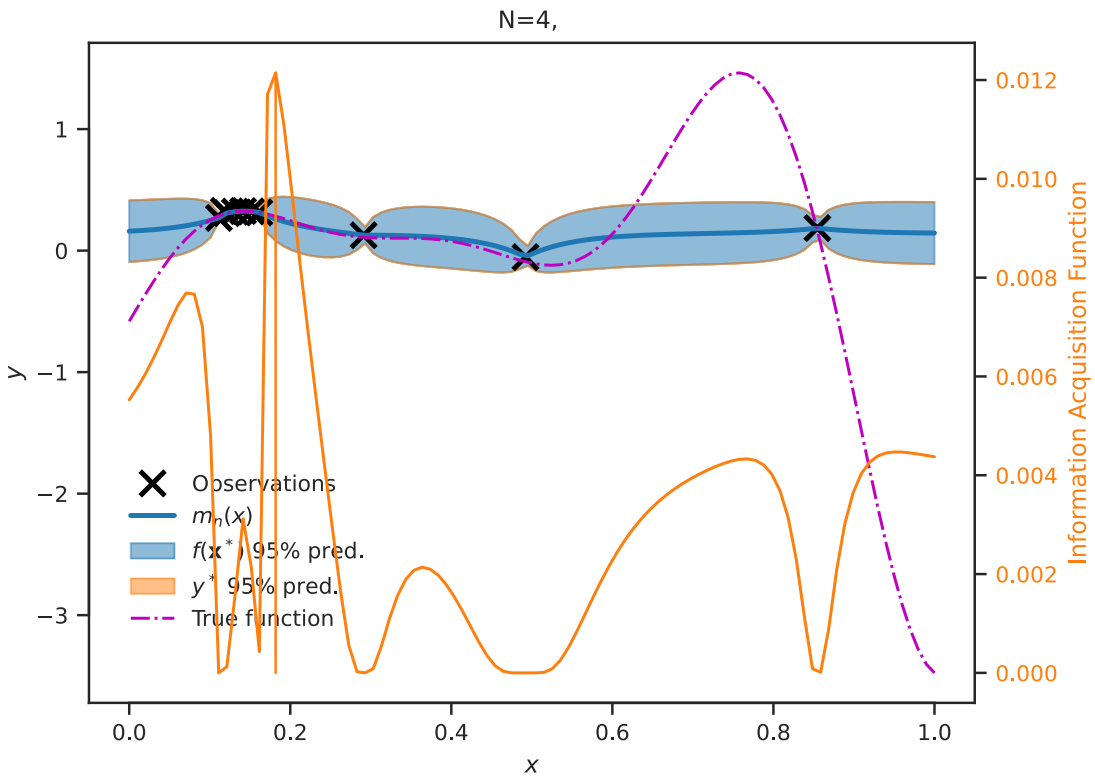
```
/tmp/ipython-input-4-753138915.py:10: DeprecationWarning: __array_wrap__ must
accept context and return_scalar arguments (positionally) in the future.
(Deprecated NumPy 2.0)
  return (6 * x - 2) ** 2 * np.sin(12 * x - 4)
/tmp/ipython-input-17-1455736800.py:186: DeprecationWarning: __array_wrap__ must
accept context and return_scalar arguments (positionally) in the future.
(Deprecated NumPy 2.0)
  next_x * np.ones(100),
```

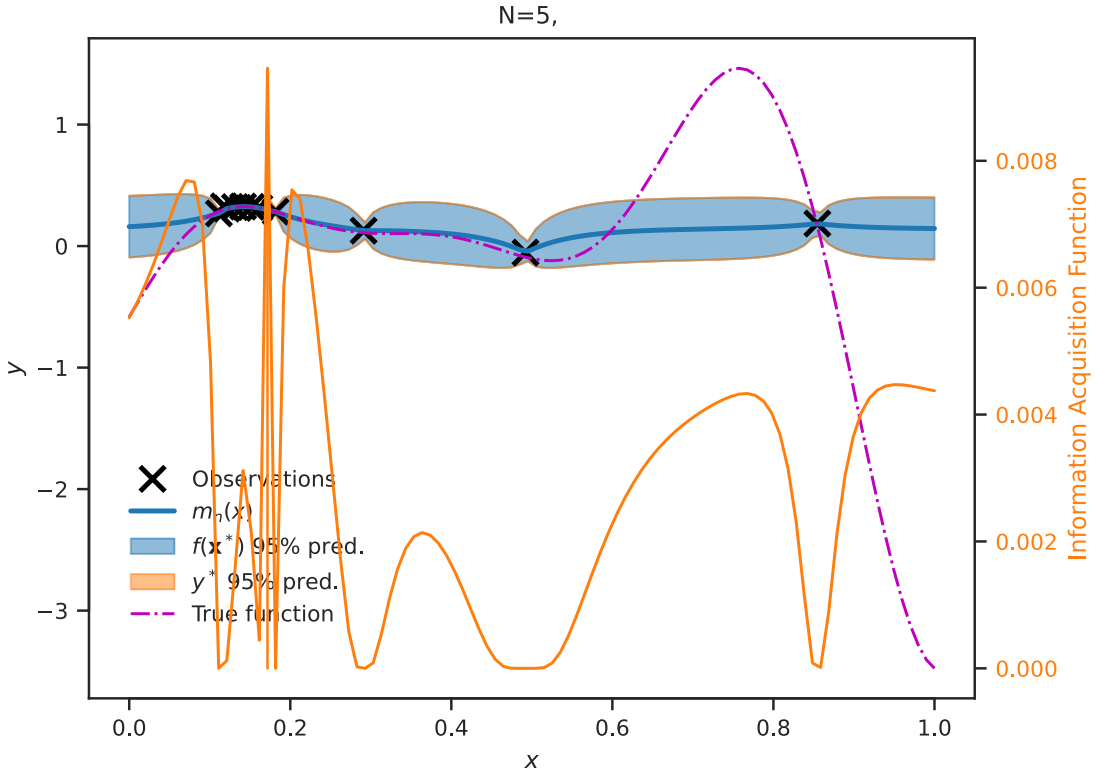


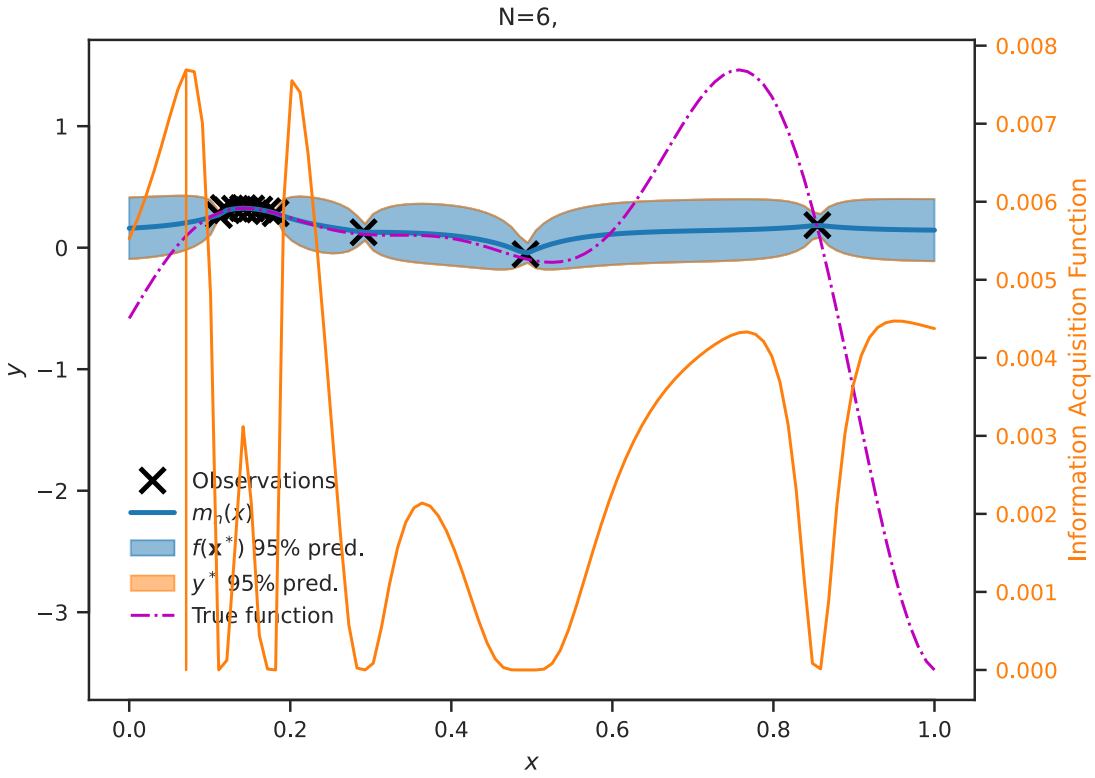

N=1,
Observations
$m_n(x)$
$f(\mathbf{x}^*)$ 95% pred.
y^* 95% pred.
True function
x
y
Information Acquisition Function

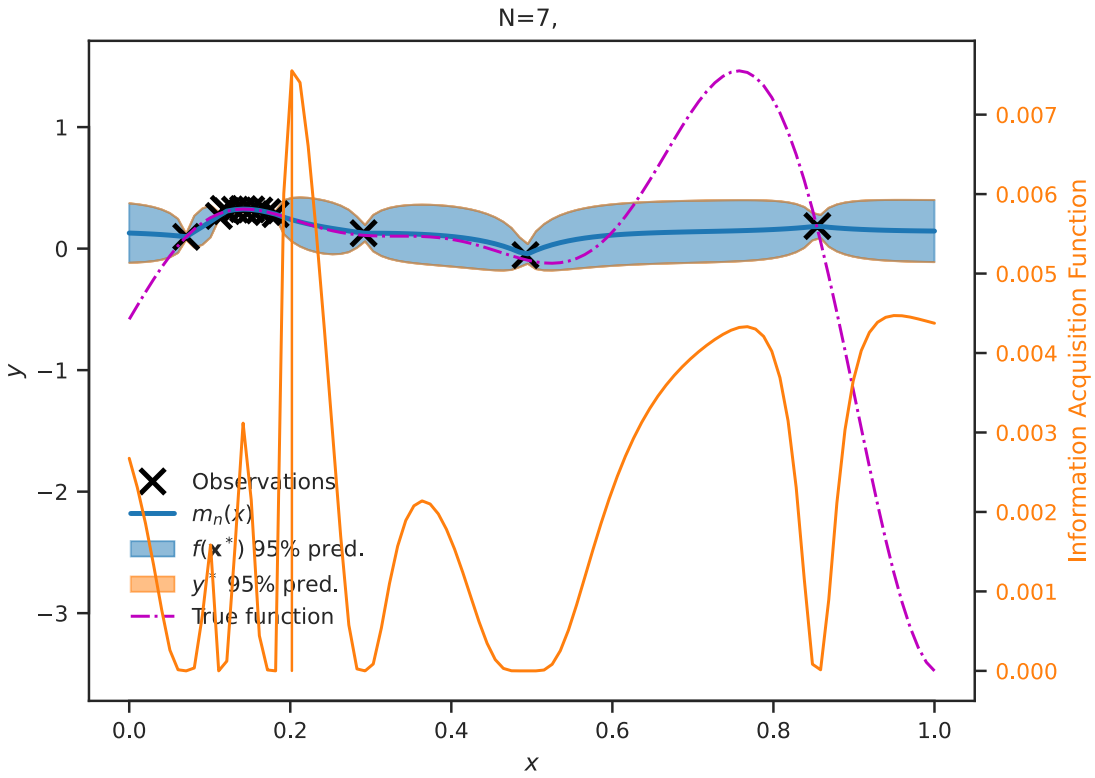

N=2,
Observations
$m_n(x)$
$f(\mathbf{x}^*)$ 95% pred.
y^* 95% pred.
True function
x
y
Information Acquisition Function

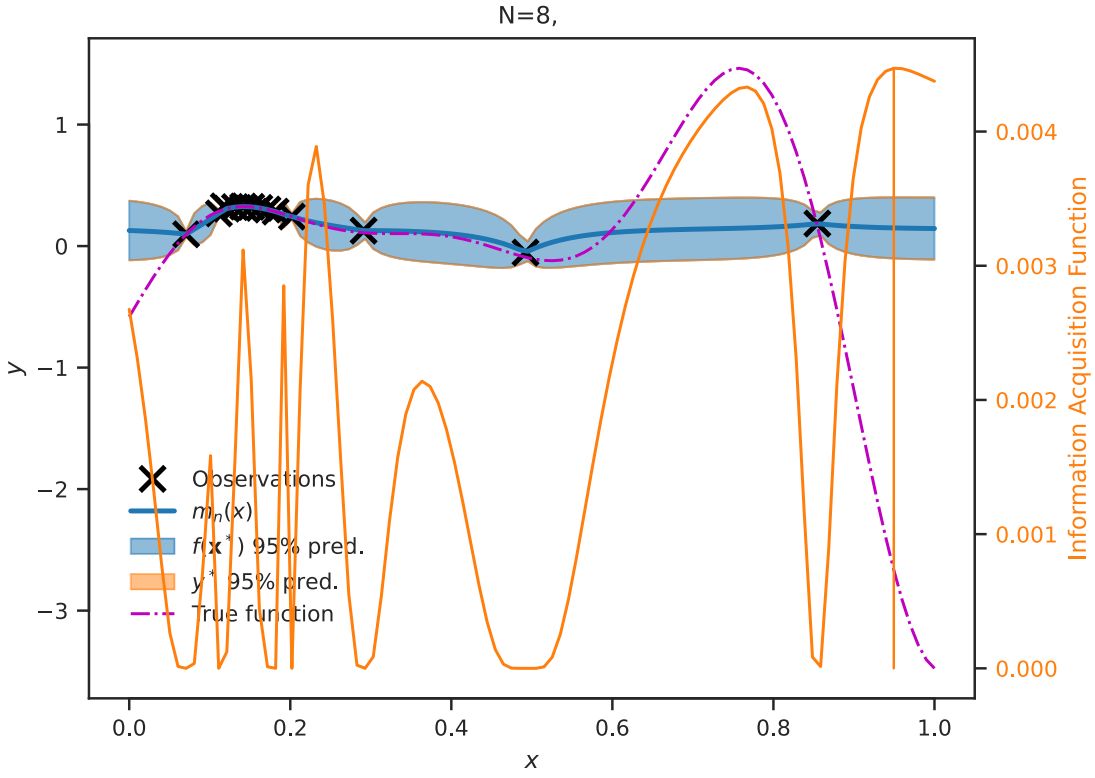

N=3,
Observations
$m_n(x)$
$f(x^*)$ 95% pred.
y^* 95% pred.
True function
y
x
Information Acquisition Function
1
0
−1
−2
−3
0.0
0.2
0.4
0.6
0.8
1.0
0.014
0.012
0.010
0.008
0.006
0.004
0.002
0.000

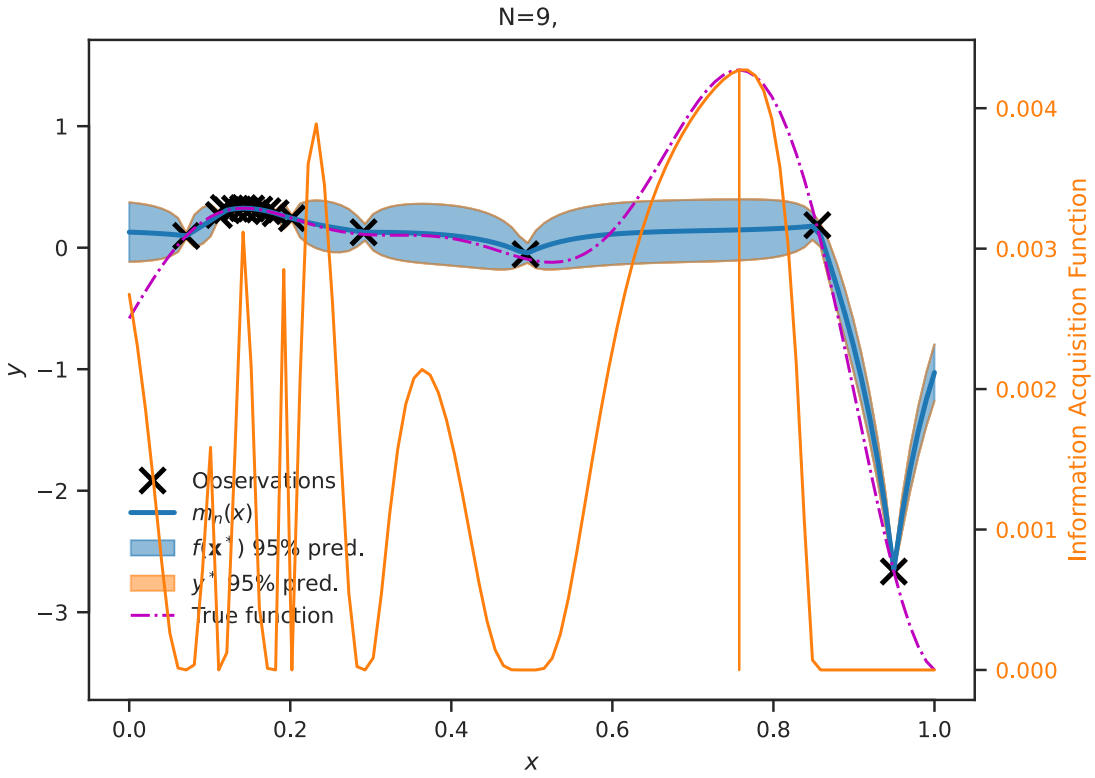

N=4,
Observations
$m_n(x)$
$f(\mathbf{x}^*)$ 95% pred.
y^* 95% pred.
True function
Information Acquisition Function
x
y

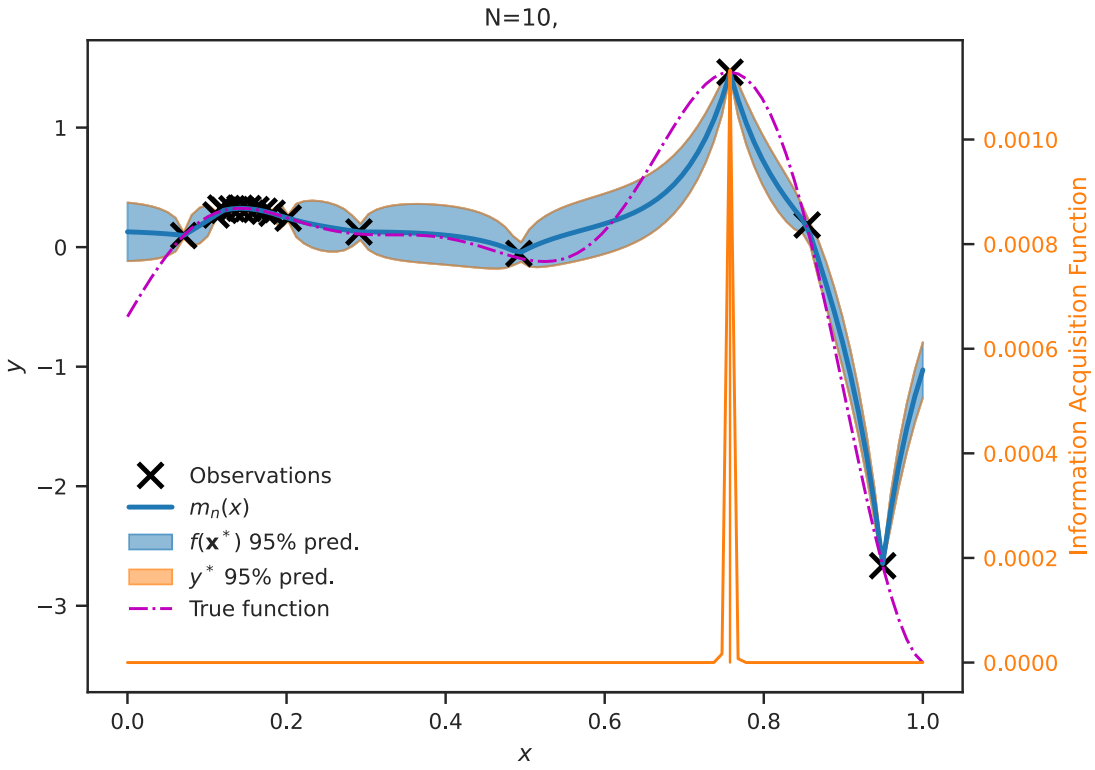

N=5,
Observations
$m_n(x)$
$f(\mathbf{x}^*)$ 95% pred.
y^* 95% pred.
True function
Information Acquisition Function
x
y

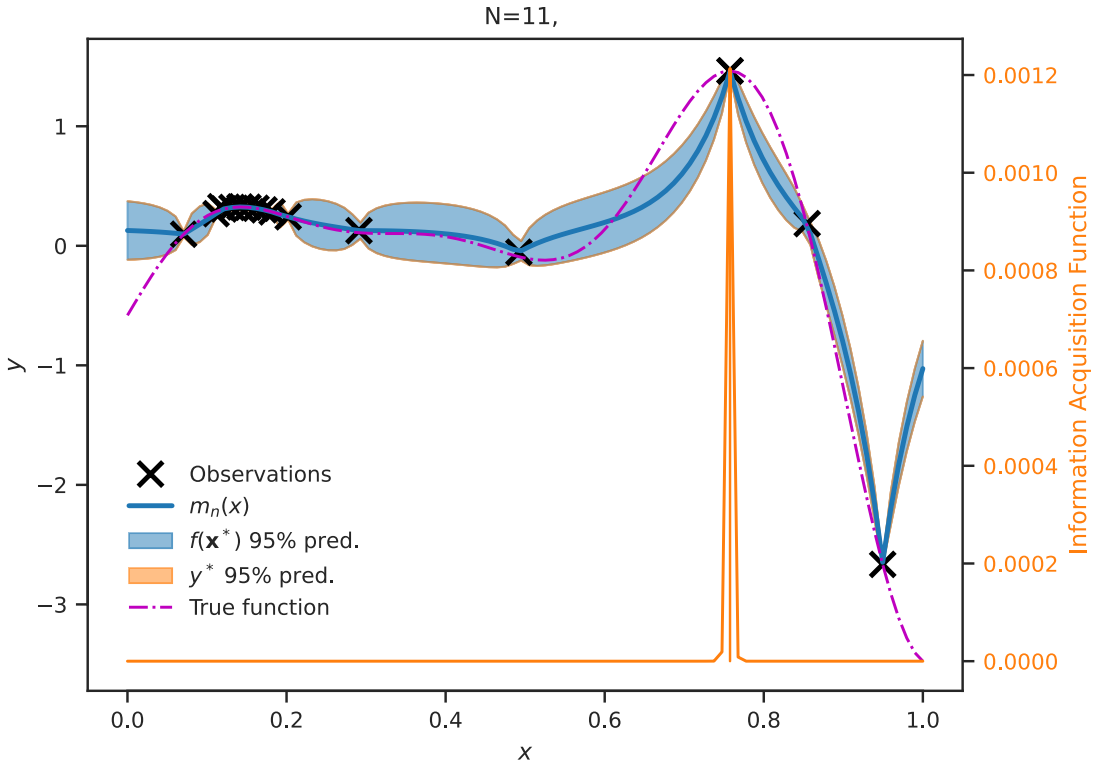

N=6,
Observations
$m_\eta(x)$
$f(\mathbf{x}^*)$ 95% pred.
y^* 95% pred.
True function
y
x
Information Acquisition Function
1
0
−1
−2
−3
0.0
0.2
0.4
0.6
0.8
1.0
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0.000

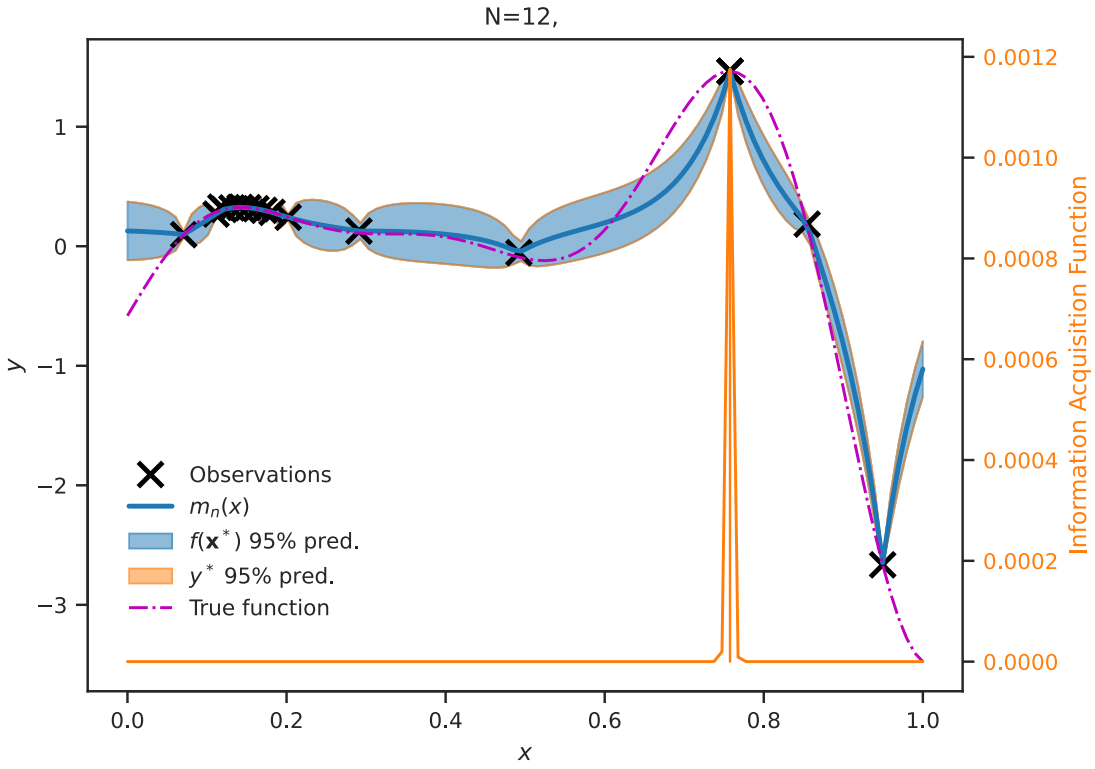

N=7,
Observations
$m_n(x)$
$f(x^*)$ 95% pred.
y^* 95% pred.
True function
Information Acquisition Function
x
y

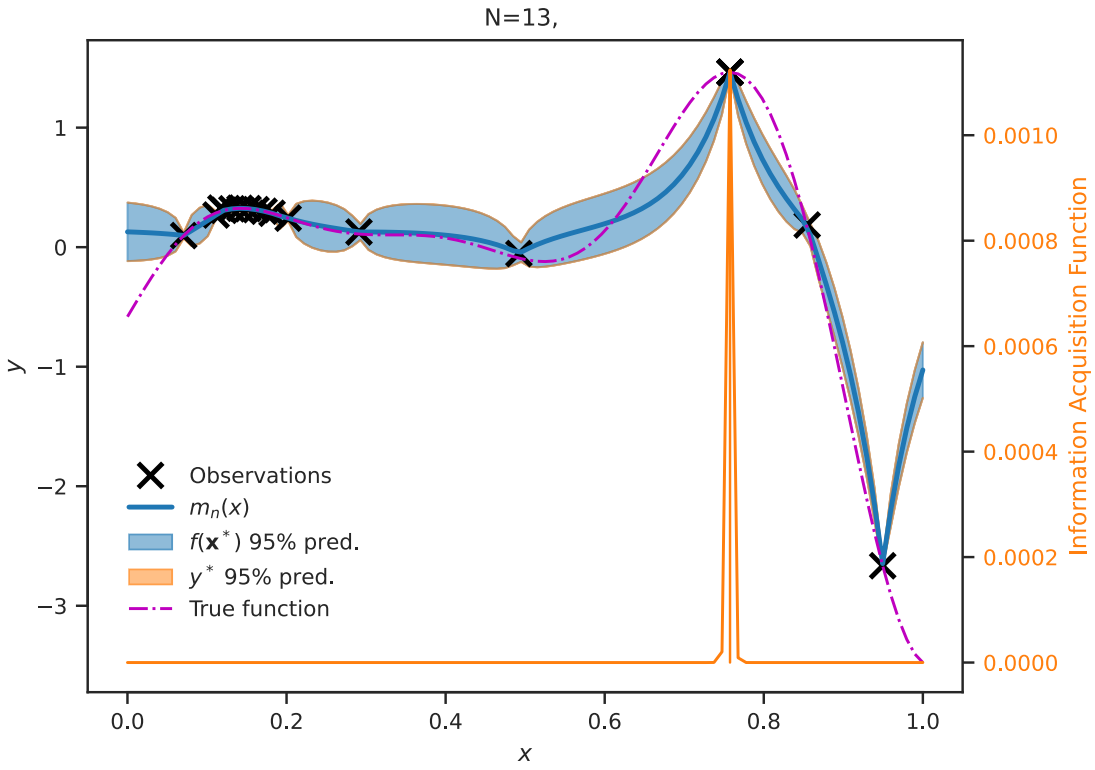

N=8,
Observations
$m_n(x)$
$f(x^*)$ 95% pred.
y^* 95% pred.
True function
Information Acquisition Function
x
y

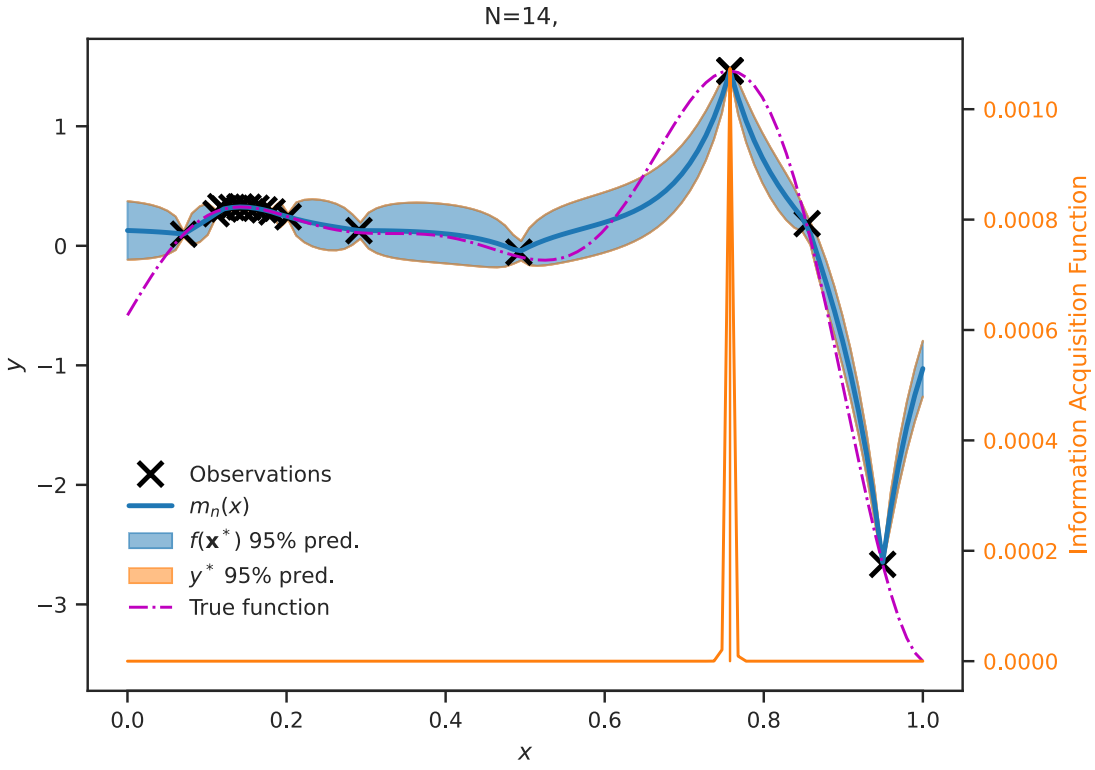

N=9,
Observations
$m_n(x)$
$f(x^*)$ 95% pred.
y^* 95% pred.
True function
Information Acquisition Function
x
y
0.0
0.2
0.4
0.6
0.8
1.0
1
0
−1
−2
−3
0.004
0.003
0.002
0.001
0.000


N=10,
Observations
$m_n(x)$
$f(\mathbf{x}^*)$ 95% pred.
y^* 95% pred.
True function
Information Acquisition Function
x
y


N=11,
Observations
$m_n(x)$
$f(\mathbf{x}^*)$ 95% pred.
y^* 95% pred.
True function
Information Acquisition Function
x
y


N=12,
Observations
$m_n(x)$
$f(\mathbf{x}^*)$ 95% pred.
y^* 95% pred.
True function
y
x
Information Acquisition Function
1
0
−1
−2
−3
0.0
0.2
0.4
0.6
0.8
1.0
0.0012
0.0010
0.0008
0.0006
0.0004
0.0002
0.0000


N=13,
Observations
$m_n(x)$
$f(\mathbf{x}^*)$ 95% pred.
y^* 95% pred.
True function
Information Acquisition Function
x
y


N=14,
Observations
$m_n(x)$
$f(\mathbf{x}^*)$ 95% pred.
y^* 95% pred.
True function
Information Acquisition Function
x
y

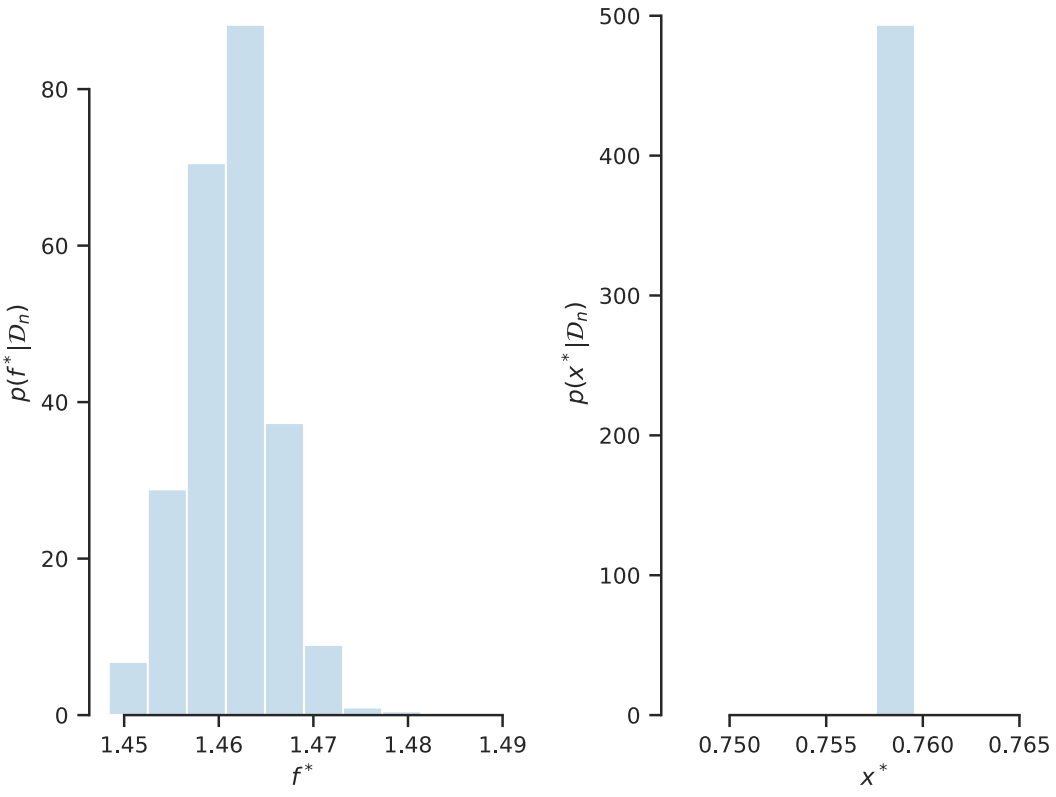
1.11.2 How many iterations does the algorithm take to converge? That is, how quickly does it identify the critical point?

The algorithm converges at the 11th iteration. On the above plots, we see that at $N = 10$ (which is the 11th iteration), the algorithm successfully identifies the maximum of the true function. Note that this critical point represents the minimum of the Forrester function.

1.11.3 Quantify the uncertainty about the solution to the optimization problem with the trained Gaussian process

To quantify the uncertainty, we will sample many functions, find the location and value of the maximum of each function, then plot the histogram. We will also find the mean location and and value of the maximum, as well as the 95% confidence intervals.

[ ]:
```
# Use same function as in Part B to make histograms of the location and value
# of the optimum point
plot_max_and_argmax(model, xs);
```



[ ]:
```python
# Sample many functions from the optimized GP
f_star = model(xs)
f_samples = f_star.sample(sample_shape=torch.Size([10000])).numpy()

# Find the maximum of each sample and its location
max_f_samples = np.max(f_samples, axis=1)
x_star_samples = xs.numpy()[np.argmax(f_samples, axis=1)]

# Find the mean and 95% credible interval for the location of the maximum
x_star_mean = np.mean(x_star_samples)
x_star_ci = np.percentile(x_star_samples, [2.5, 97.5])
print(f"Posterior mean of x*: {x_star_mean:.4f}")
print(f"95% credible interval: [{x_star_ci[0]:.4f}, {x_star_ci[1]:.4f}]\n")

# Find the mean and 95% credible interval for the value of the maximum
f_star_mean = np.mean(max_f_samples)
f_star_ci = np.percentile(max_f_samples, [2.5, 97.5])
print(f"Posterior mean of f*: {f_star_mean:.4f}")
print(f"95% credible interval: [{f_star_ci[0]:.4f}, {f_star_ci[1]:.4f}]")
```

```
Posterior mean of x*: 0.7576
95% credible interval: [0.7576, 0.7576]

Posterior mean of f*: 1.4615
95% credible interval: [1.4527, 1.4702]
```

After solving the optimization problem we are much more certain about the location and value of the optimum point. With 95% certainty, we believe that the location is 0.7576. For the value of the optimum point, we are 95% certain that it lies between 1.4547 and 1.4702.

1.12 Part D - Testing your intuition

In a real-world scenario, you may not be able to keep running experiments until the optimization problem has obviously converged due to time, budget considerations, etc. Imagine yourself in a situation where you are deciding whether or not to query the blackbox function an additional time.

Describe (in words) how you could make this decision using the principles you've learned in this course.

Answer:

When deciding whether to query the blackbox function again, I would use the following decision metrics:

1. Expected Improvement for the Next Experiment: Keep track of the expected improvement for each iteration. When the maximum expected improvement falls below a chosen threshold, further experiments are not worth it. The chosen threshold would represent the cost (time, money, risk) of an additional experiment.
2. Posterior Credible Interval Width: If the model already shows high confidence in the optimum, then no more experiments are necessary. We can choose a threshold that the interval width

must be below, such as a 95% confidence interval width < 0.01.

3. Probability of Improvement for the Next Experiment: Compute the probability of getting a value that is greater than the currently observed maximum. If that probability is exceeds a certain threshold, then it is worth it to perform another experiment. If the probability of improvement is small (close to 0), then more experiments are unlikely to be helpful.
4. Evolution of the Estimated Optimum: Monitor how much the estimated optimum has changed over time in the previous experiments. If the estimated optimum appears to be stabilizing/converging, then no more experiments are needed.

By using the tools I learned in this course, I can make an informed decision that balances potential gain with real-world constraints.