# ME539.Homework.6

July 27, 2022

## 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.

[1]:
```
import numpy as np
np.set_printoptions(precision=3)
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
sns.set(rc={"figure.dpi":100, "savefig.dpi":300})
sns.set_context("notebook")
sns.set_style("ticks")

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

# !sudo apt install texlive texlive-latex-extra texlive-fonts-recommended dvipng
# !sudo apt install cm-super

from matplotlib import rc
rc('text', usetex=True)

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)

try:
  import GPy
except:
  _=!pip install GPy
  import GPy
import scipy.stats as st
```

[2]:
```
def sample_and_plot(
    xs,
    ms,
    k
):
    """Take 5 samples of f ~ GP(m, k) and plot them

    Arguments
    xs      -- The dense set of x values to plot against
    ms      -- User specified 1D array for the mean function
    k       -- User specified covariance function
    """
    # Find the covariance matrix. You need to add a small number
    # to the diagonal to ensure numerical stability
    nugget = 1e-6
    K = k.K(xs[:, None]) + nugget * np.eye(xs.shape[0])
    # A multivariate normal that can be used to sample the function values
    F = st.multivariate_normal(mean=ms.flatten(), cov=K)
    # Take the function samples
    f_samples = F.rvs(size=5)
    # Plot the samples
    fig, ax = plt.subplots(figsize = (9, 6))
    ax.plot(xs, f_samples.T, lw=0.5)
    ax.set_ylabel("$f(x)$")
    ax.set_xlabel("$x$")
    ax.set_title("5 Samples of $f(x)$")
```

```
[3]: def compute_BIC(
         model,
         n
     ):
         """Compute Bayesian Information Criterion (BIC) for a fitted model

         Arguments
         model      -- fitted model using maximum likelihood
         n          -- number of data points/observations
         """
         # return value for BIC using formal mathematical definition
         return model._size_transformed()*np.log(n) - 2*model.log_likelihood()
```

```
[4]: def calc_y(
         ys,
         y
     ):
         """Compute original y from scaled y's

         Arguments
         ys      -- the scaled y values
         y       -- the original data set that contains necessary information
                    about the maximum and minimum value in the data set
         """
         # use known conversion equation specified earlier in the problem
         orig = ys * (y.max() - y.min()) + y.min()
         return orig
```

## 1.3 Student details

- **First Name:** Jack
- **Last Name:** Girard
- **Email:** girard2@purdue.edu

## 1.4 Problem 1 - Defining priors on function spaces

In this problem we are going to 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 it $f(x)$, 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 it. 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 a 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:

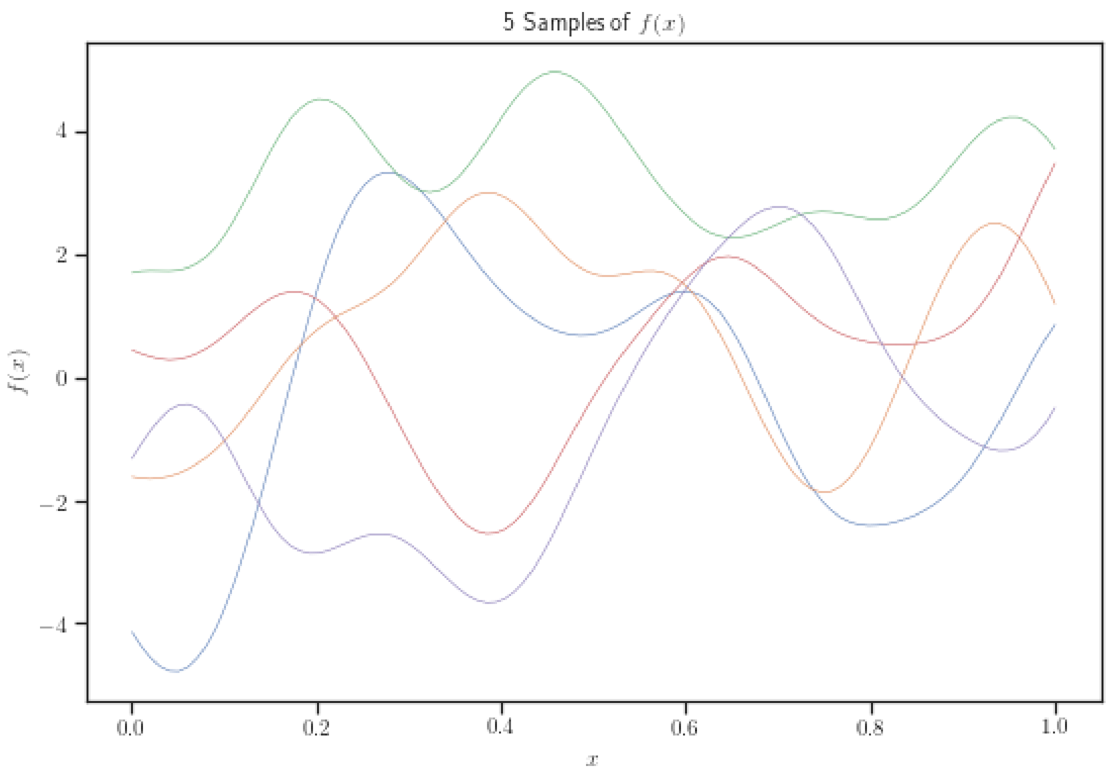
$$k(x, x') = s^2 \exp\left\{-\frac{(x - x')^2}{2\ell^2}\right\},$$

with variance:

$$1.96s = 4 \to s^2 = k(x, x) = \mathbb{V}[f(x)] = \left(\frac{4}{1.96}\right)^2,$$

and lengthscale $\ell = 0.1$. We chose the variance to be $\left(\frac{4}{1.96}\right)^2$ so that with (about) 95% probability the values of $f(x)$ are between -4 and 4.

[5]:
```
# Define the covariance function
k = GPy.kern.RBF(1)
k.lengthscale = 0.1
k.variance = (4.0/1.96)**2
# Define dense set of x values to plot against
xs = np.linspace(0, 1, 100)
# The mean function at xs
ms = np.zeros(xs.shape)
# Sample and plot using function
sample_and_plot(xs, ms, k)
```



### 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:**

The mean function should be:

$$m(x) = 0.$$

The covariance function should be a squared exponential:

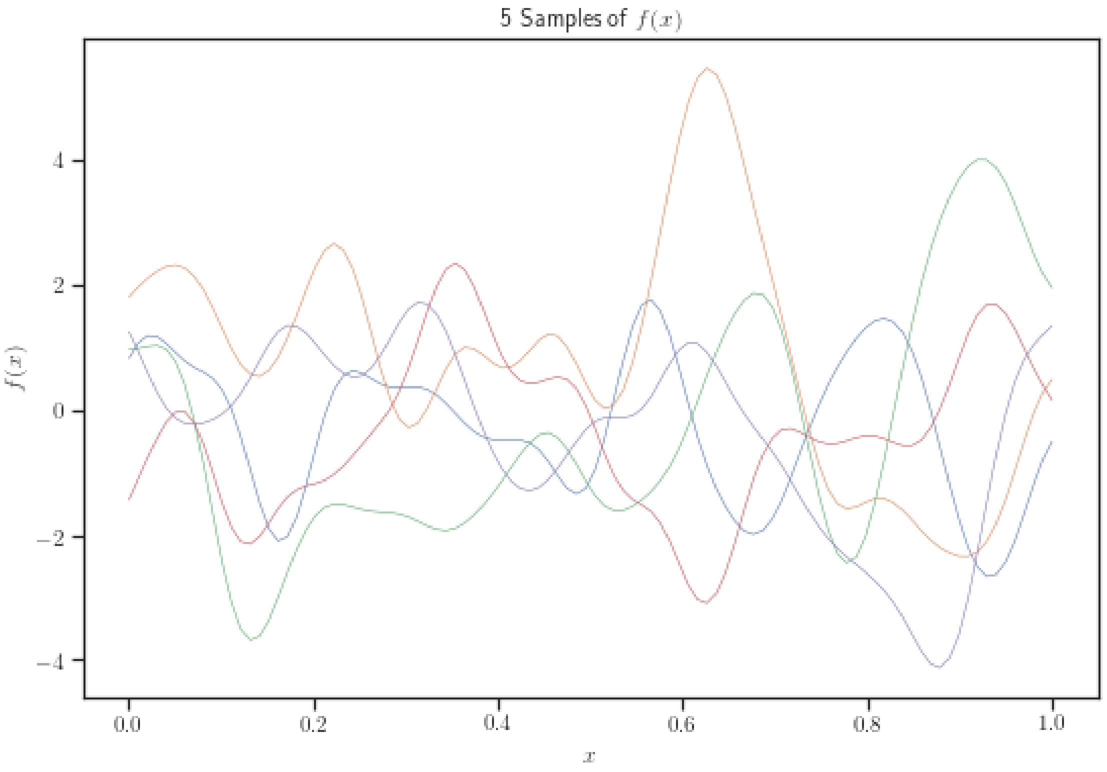
$$k(x, x') = s^2 \exp\left\{-\frac{(x - x')^2}{2\ell^2}\right\},$$

with variance:

$$1.96s = 3 \rightarrow s^2 = k(x, x) = \mathbb{V}[f(x)] = \left(\frac{3}{1.96}\right)^2,$$

and lengthscale $\ell = 0.05$. We chose the variance to be $\left(\frac{3}{1.96}\right)^2$ so that with (about) 95% probability the values of $f(x)$ are between -3 and 3.

[6]:
```
# Define the covariance function
k = GPy.kern.RBF(1)
k.lengthscale = 0.05
k.variance = (3.0/1.96)**2
# Define dense set of x values to plot against
xs = np.linspace(0, 1, 100)
# The mean function at xs
ms = np.zeros(xs.shape)
# Sample and plot using function
sample_and_plot(xs, ms, k)
```



### 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 approximatly of size $\Delta x = 0.1$. + You think that $f(x)$ is between -5 and 5.

Hint: Use `GPy.kern.Exponential`.

**Answer:**

The mean function should be:

$$m(x) = 0.$$

The covariance function should be an exponential:

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

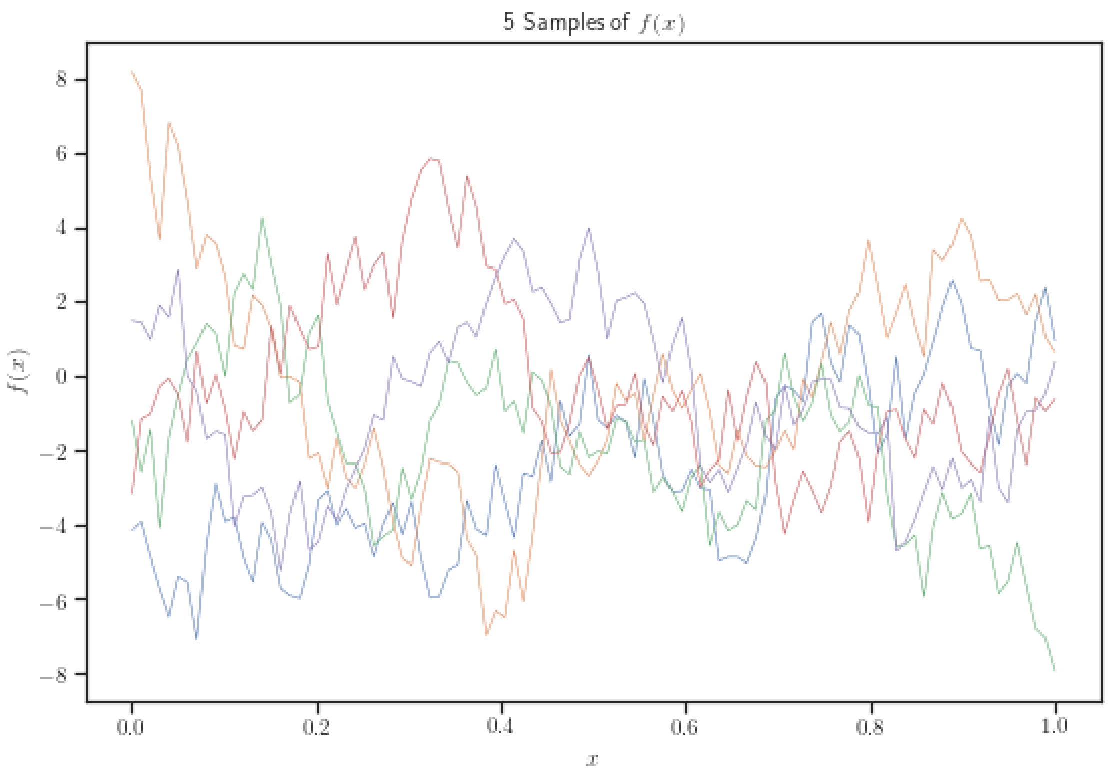
with variance:

$$1.96s = 5 \rightarrow s^2 = k(x, x) = \mathbb{V}[f(x)] = \left(\frac{5}{1.96}\right)^2,$$

and lengthscale $\ell = 0.1$. We chose the variance to be $\left(\frac{5}{1.96}\right)^2$ so that with (about) 95% probability the values of $f(x)$ are between -5 and 5.

Reference on the mathematical form of the exponential covariance function: https://www.mathworks.com/help/stats/kernel-covariance-function-options.html

[7]:
```
# Define the covariance function
k = GPy.kern.Exponential(1)
k.lengthscale = 0.1
k.variance = (5.0/1.96)**2
# Define dense set of x values to plot against
xs = np.linspace(0, 1, 100)
# The mean function at xs
ms = np.zeros(xs.shape)
# Sample and plot using function
sample_and_plot(xs, ms, k)
```



### 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 approximatly of size $\Delta x = 0.5$ of the period. + You think that $f(x)$ is between -5 and 5.

Hint: Use `GPy.kern.StdPeriodic`.

**Answer:**

The mean function should be:

$$m(x) = 0.$$

The covariance function should be a standard periodic:

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

with period:

$$p = 0.1,$$

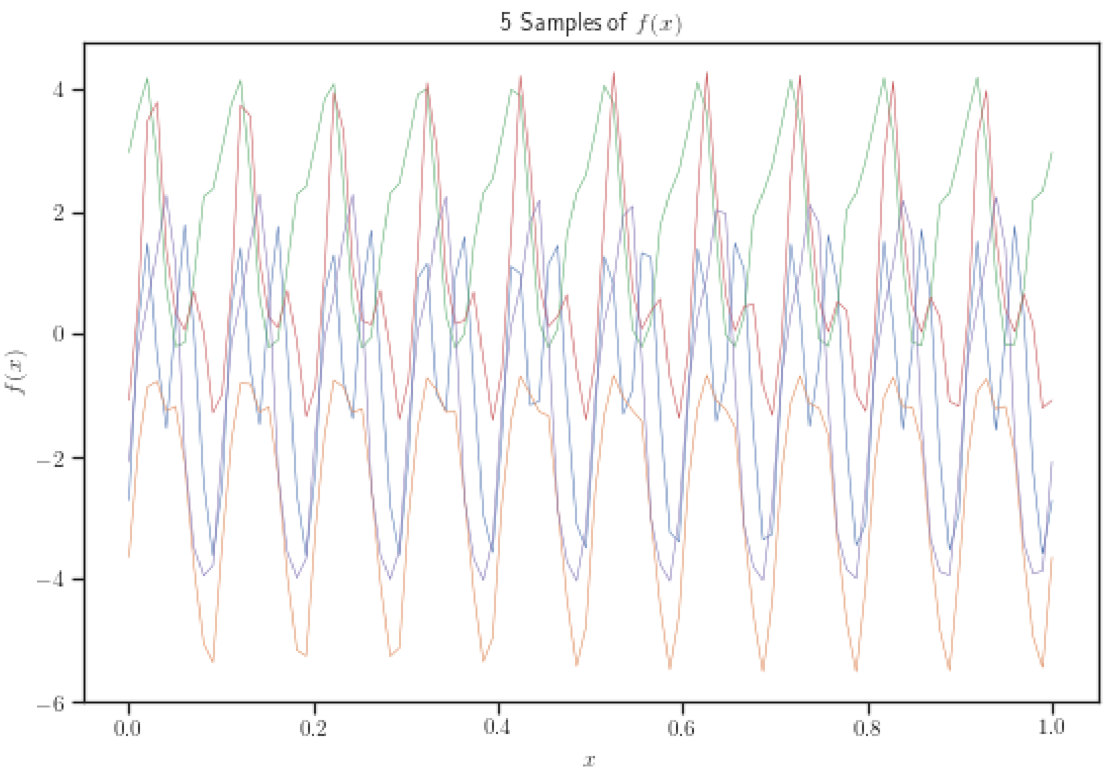
and with variance:

$$1.96s = 5 \to s^2 = k(x,x) = \mathbb{V}[f(x)] = \left(\frac{5}{1.96}\right)^2,$$

and lengthscale $\ell = 0.5$. We chose the variance to be $\left(\frac{5}{1.96}\right)^2$ so that with (about) 95% probability the values of $f(x)$ are between -5 and 5.

Reference on the mathematical form of the standard periodic covariance function: https://www.cs.toronto.edu/~duvenaud/cookbook/

[8]:
```
# Define the covariance function
k = GPy.kern.StdPeriodic(1)
k.period = 0.1
k.lengthscale = 0.5
k.variance = (5.0/1.96)**2
# Define dense set of x values to plot against
xs = np.linspace(0, 1, 100)
# The mean function at xs
ms = np.zeros(xs.shape)
# Sample and plot using function
sample_and_plot(xs, ms, k)
```



### 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 approximatly 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 `GPy.kern.StdPeriodic`.

**Answer:**

The mean function should be:

$$m(x) = 0.$$

The covariance function should be a standard periodic:

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

with period:

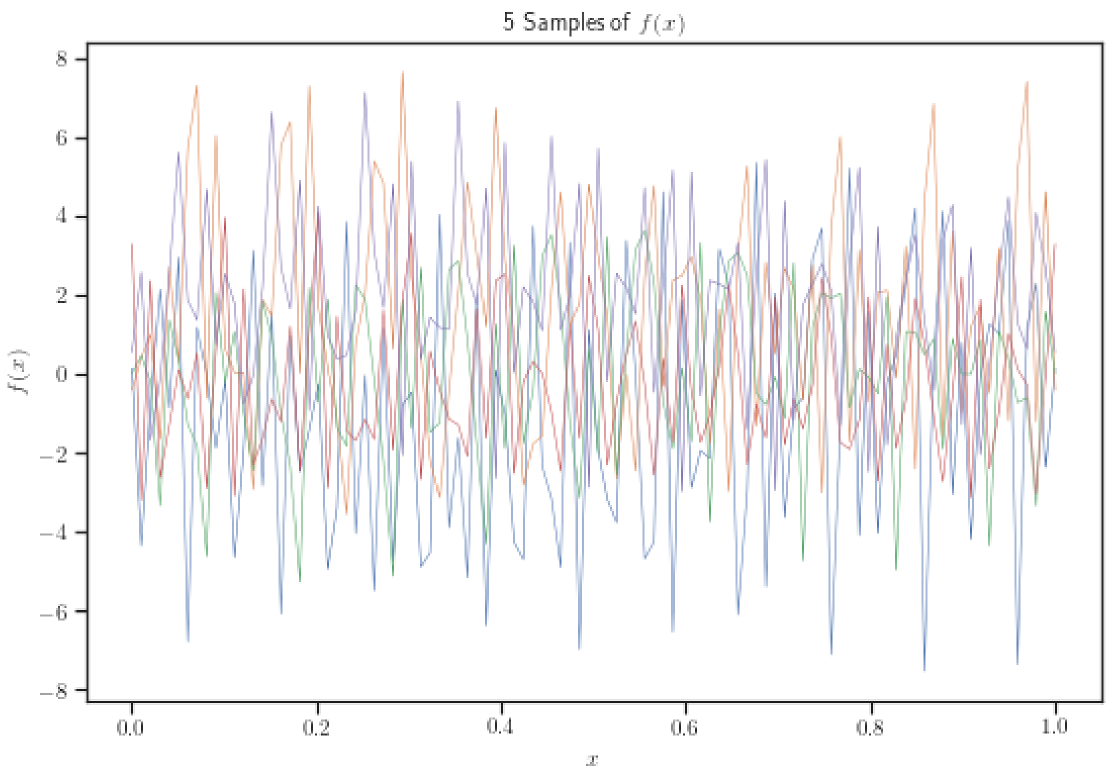
$$p = 0.1,$$

and with variance:

$$1.96s = 5 \rightarrow s^2 = k(x, x) = \mathbb{V}[f(x)] = \left(\frac{5}{1.96}\right)^2,$$

and lengthscale $\ell = 0.1$. We chose the variance to be $\left(\frac{5}{1.96}\right)^2$ so that with (about) 95% probability the values of $f(x)$ are between -5 and 5.

[9]:
```python
# Define the covariance function
k = GPy.kern.StdPeriodic(1)
k.period = 0.1
k.lengthscale = 0.1
k.variance = (5.0/1.96)**2
# Define dense set of x values to plot against
xs = np.linspace(0, 1, 100)
# The mean function at xs
ms = np.zeros(xs.shape)
# Sample and plot using function
sample_and_plot(xs, ms, k)
```



### 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 lengthscale 0.5 - $f_2(x)$ is continuous, nowhere differentiable with variance 0.1 and lengthscale 0.1

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

**Answer:**

For the *first* covariance function, $f_1(x)$:

The mean function should be:

$$m_1(x) = 0.$$

The covariance function should be a squared exponential:

$$k_1(x, x') = s_1^2 \exp\left\{-\frac{(x - x')^2}{2\ell_1^2}\right\},$$

with variance:

$$s_1^2 = k_1(x,x) = \mathbb{V}[f_1(x)] = 2,$$

and lengthscale $\ell_1 = 0.5$. We chose the variance to be 2 so that with (about) 95% probability the values of $f_1(x)$ are between $-2\sqrt{2}$ and $2\sqrt{2}$.

For the *second* covariance function, $f_2(x)$:

The mean function should be:

$$m_2(x) = 0.$$

The covariance function should be an exponential:

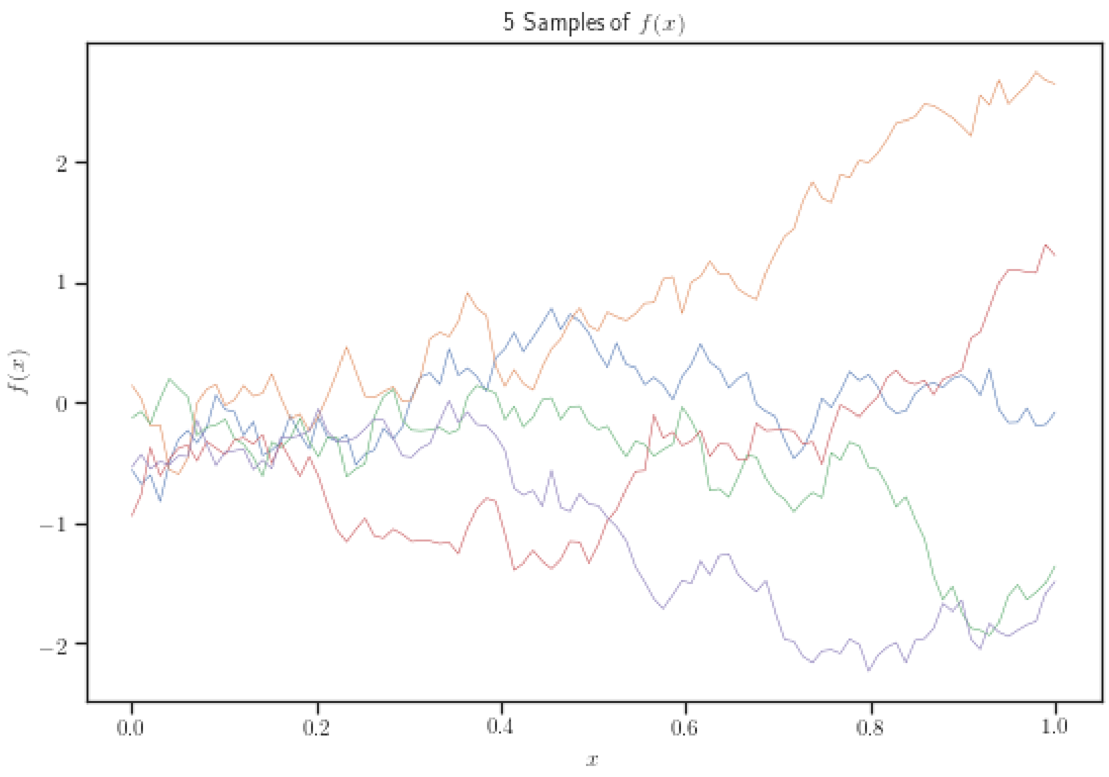
$$k_2(x,x') = s_2^2 \exp\left\{-\frac{(x-x')}{\ell_2}\right\},$$

with variance:

$$s_2^2 = k_2(x,x) = \mathbb{V}[f_2(x)] = 0.1,$$

and lengthscale $\ell_2 = 0.1$. We chose the variance to be 0.1 so that with (about) 95% probability the values of $f_2(x)$ are between $-2\sqrt{0.1}$ and $2\sqrt{0.1}$.

[10]:
```
# Define the covariance functions
k1 = GPy.kern.RBF(1)
k1.variance = 2
k1.lengthscale = 0.5
k2 = GPy.kern.Exponential(1)
k2.variance = 0.1
k2.lengthscale = 0.1
# Generate new covariance as the sum of the above covariance functions
k = k1 + k2
# Define dense set of x values to plot against
xs = np.linspace(0, 1, 100)
# The mean function at xs
ms = np.zeros(xs.shape)
# Sample and plot using function
sample_and_plot(xs, ms, k)
```



### 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), lengthscale 0.1 (relative to the period), and variance 2. - $f_2(x)$ is smooth with lengthscale 0.5 and variance 1.

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

**Answer:**

For the *first* covariance function, $f_1(x)$:

The mean function should be:

$$m_1(x) = 0.$$

The covariance function should be a standard periodic:

$$k_1(x,x') = s_1^2 \exp\left(-\frac{2\sin^2\left(\frac{\pi}{p}|x-x'|\right)}{\ell_1^2}\right),$$

with period:

$$p = 0.1,$$

and with variance:

$$s_1^2 = k_1(x,x) = \mathbb{V}[f_1(x)] = 2,$$

and lengthscale $\ell_1 = 0.1$. We chose the variance to be 2 so that with (about) 95% probability the values of $f_1(x)$ are between $-2\sqrt{2}$ and $2\sqrt{2}$.

For the *second* covariance function, $f_2(x)$:

The mean function should be:

$$m_2(x) = 0.$$

The covariance function should be a squared exponential:

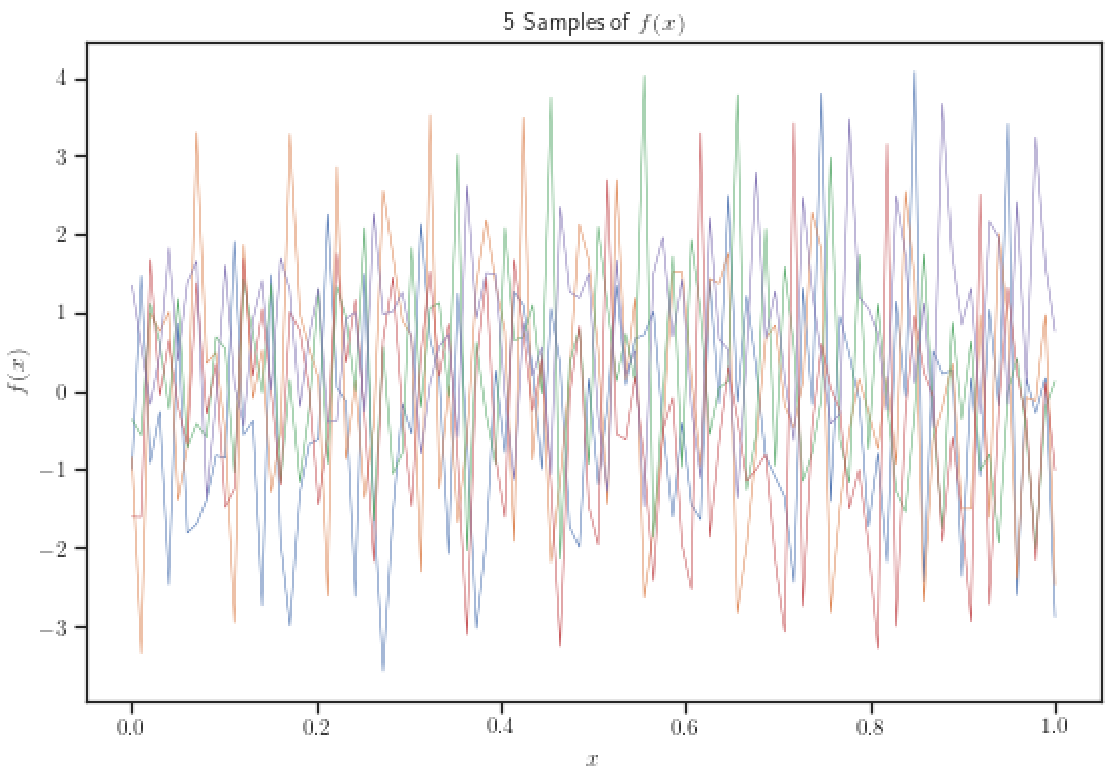
$$k_2(x,x') = s_2^2 \exp\left\{-\frac{(x-x')^2}{2\ell_2^2}\right\},$$

with variance:

$$s_2^2 = k_2(x,x) = \mathbb{V}[f_2(x)] = 1,$$

and lengthscale $\ell_2 = 0.5$. We chose the variance to be 1 so that with (about) 95% probability the values of $f(x)$ are between -2 and 2.

[11]:
```
# Define the covariance functions
k1 = GPy.kern.StdPeriodic(1)
k1.period = 0.1
k1.lengthscale = 0.1
k1.variance = 2
k2 = GPy.kern.RBF(1)
k2.lengthscale = 0.5
k2.variance = 1
# Generate new covariance as the product of the above covariance functions
k = k1 * k2
# Define dense set of x values to plot against
xs = np.linspace(0, 1, 100)
# The mean function at xs
ms = np.zeros(xs.shape)
# Sample and plot using function
sample_and_plot(xs, ms, k)
```



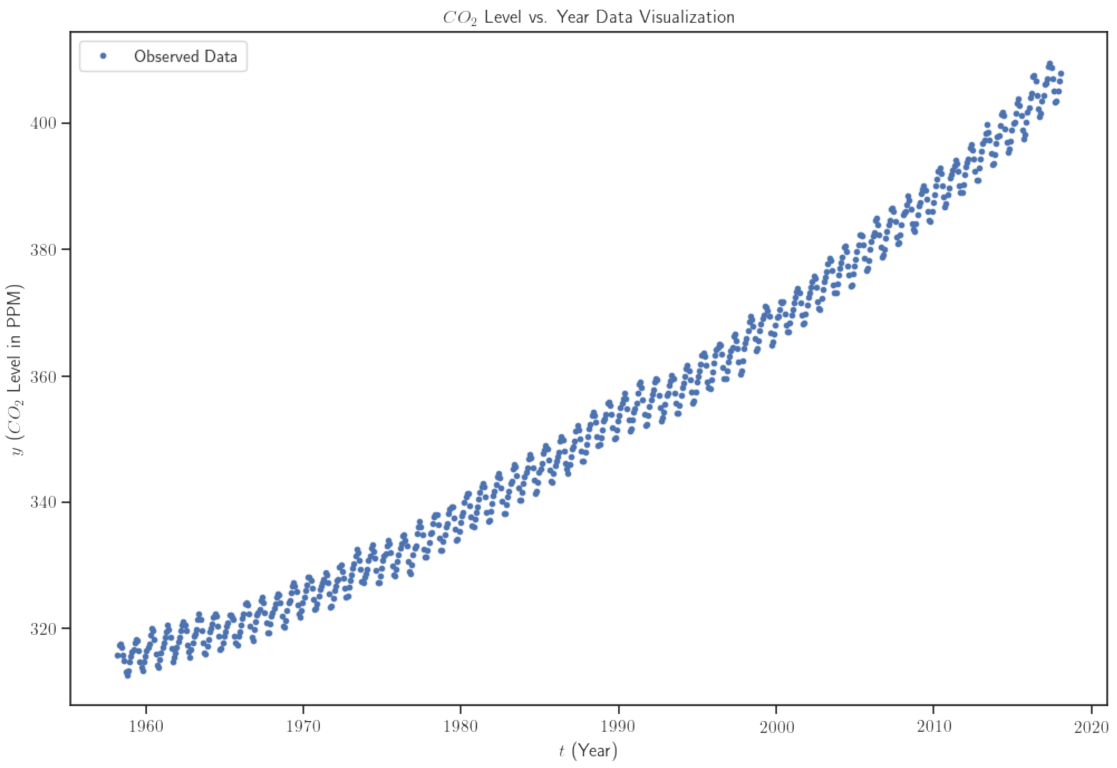
## 1.5 Problem 2

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

[12]:
```
url = "https://github.com/PredictiveScienceLab/data-analytics-se/raw/master/
↪lecturebook/data/mauna_loa_co2.txt"
download(url)
```

[13]:
```
data = np.loadtxt('mauna_loa_co2.txt')
```

[14]:
```
# load data
t = data[:, 2][:, None]  #time (in decimal dates)
y = data[:, 4][:, None]  #CO2 level (mole fraction in dry air, micromol/mol,␣
↪abbreviated as ppm)
fig, ax = plt.subplots(1, figsize = (12, 8), dpi=100)
ax.plot(t, y, '.', label='Observed Data')
ax.set_xlabel('$t$ (Year)')
ax.set_ylabel('$y$ ($CO_2$ Level in PPM)')
ax.set_title("$CO_2$ Level vs. Year Data Visualization")
plt.legend(loc='best');
```



Overall, we observe a steady growth of CO2 levels. The wiggles correspond to seasonal changes. Since the vast majority of the population inhabits the Northen hemisphere, fuel consumption goes up during the Northen 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.

It's always a good idea to work with at scaled version of the inptus and the outputs. We are going to scale the times as follows:

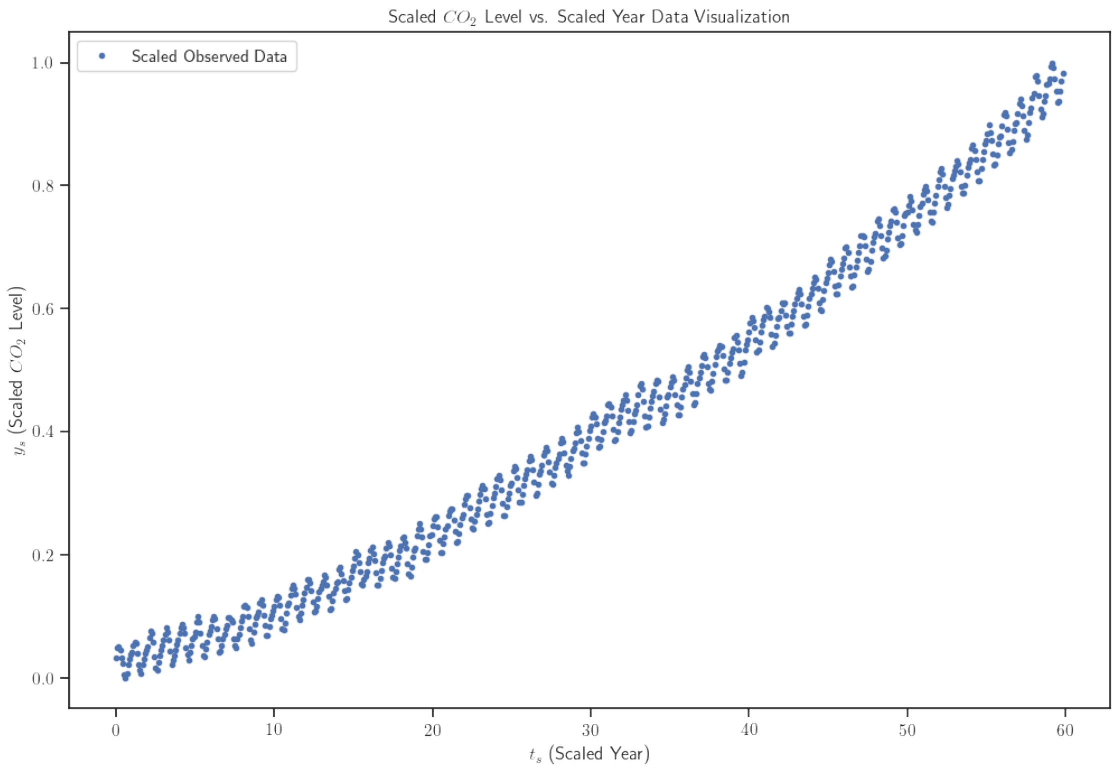
$$t_s = t - t_{\min}.$$

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

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

This takes all the $y$ between 0 and 1. Here is how the scaled data looks like:

```
[15]: t_s = t - t.min()
      y_s = (y - y.min()) / (y.max() - y.min())
      fig, ax = plt.subplots(1, figsize = (12, 8), dpi=100)
      ax.plot(t_s, y_s, '.', label='Scaled Observed Data')
      ax.set_xlabel('$t_s$ (Scaled Year)')
      ax.set_ylabel('$y_s$ (Scaled $CO_2$ Level)')
      ax.set_title("Scaled $CO_2$ Level vs. Scaled Year Data Visualization")
      plt.legend(loc='best');
```



In what follows, just work with the scaled data 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:**

```
[16]: # generate naive kernel
      naive_k = GPy.kern.RBF(1)
      # generate naive model
      naive_model = GPy.models.GPRegression(t_s, y_s, naive_k)
```

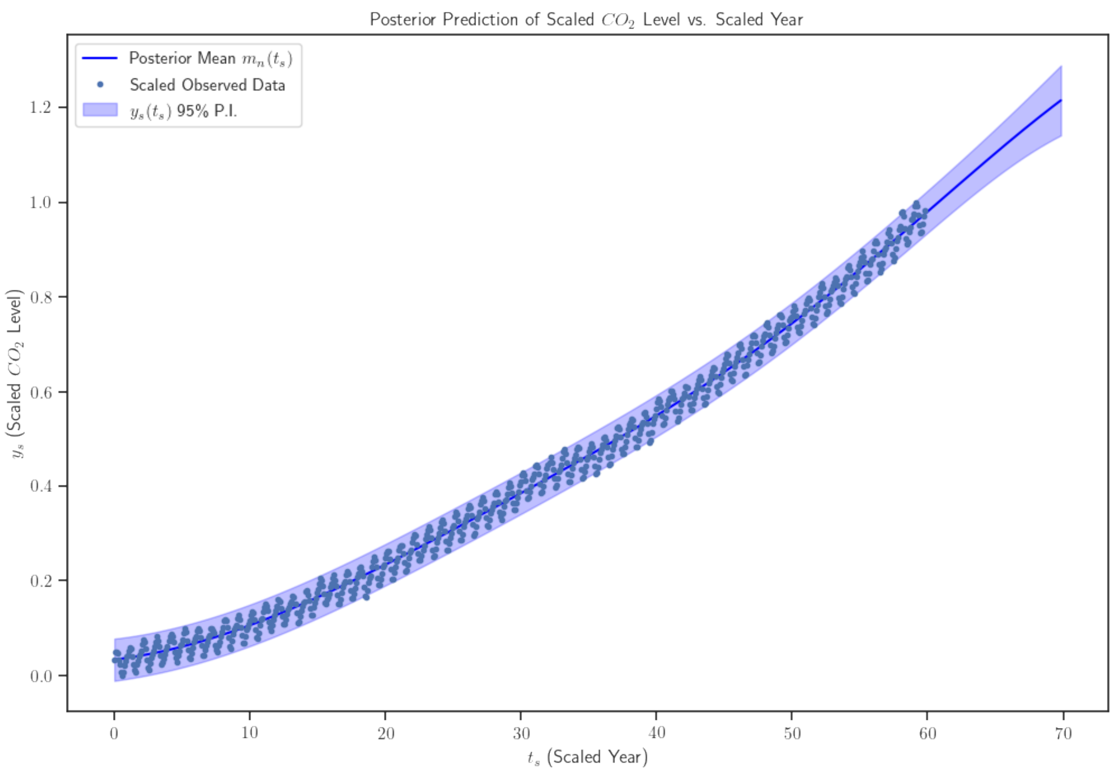
```
# find the MAP through optimization
naive_model.optimize(messages=True)
# display model overview
print(naive_model)
# reference(s): hands-on activity 22.3
```

```
HBox(children=(VBox(children=(IntProgress(value=0, max=1000), HTML(value=''))), Box(children=(
```

```
Name : GP regression
Objective : -1691.470677008756
Number of Parameters : 3
Number of Optimization Parameters : 3
Updates : True
Parameters:
  GP_regression.             |                  value  |  constraints  |
priors
  rbf.variance               |     0.4834956682760185  |      +ve      |
  rbf.lengthscale            |      32.83707387889793  |      +ve      |
  Gaussian_noise.variance    |  0.0004965143464048201  |      +ve      |
```

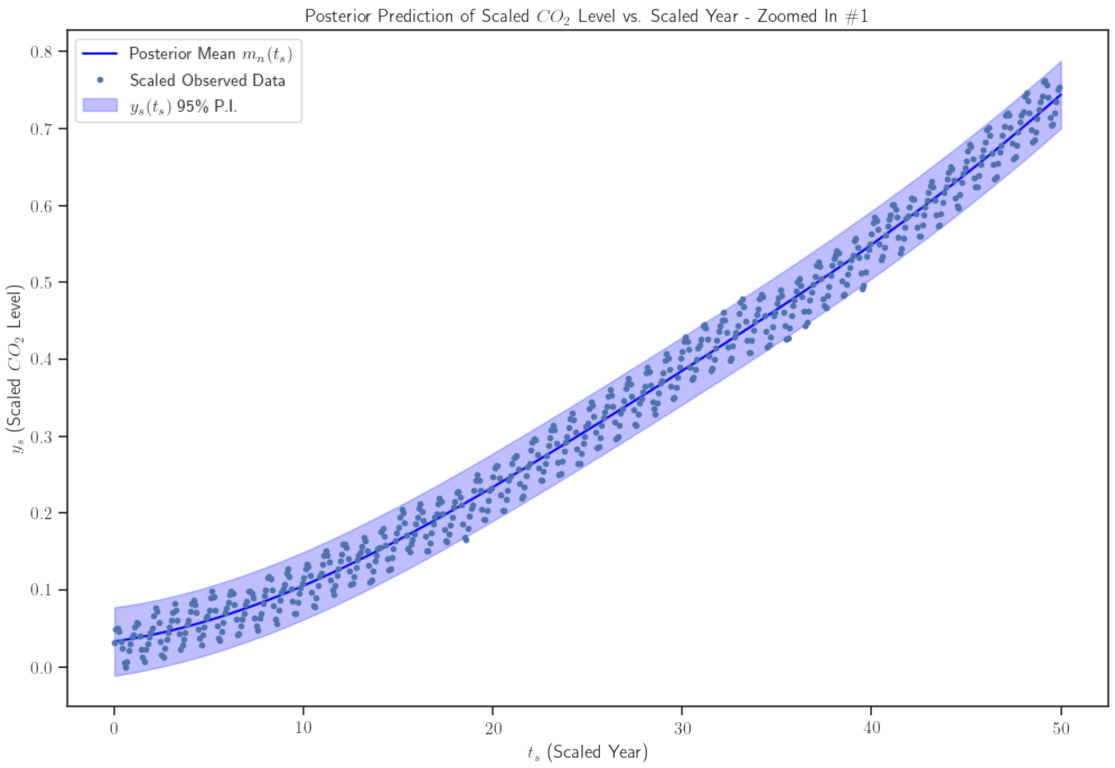
Predict everything:

[17]:
```
tss = np.linspace(0, t_s.max() + 10, 1000)[:, None]
ys, vs = naive_model.predict(tss)
ls = ys - 1.96 * np.sqrt(vs)
us = ys + 1.96 * np.sqrt(vs)
fig, ax = plt.subplots(1, figsize = (12, 8), dpi=100)
ax.plot(tss, ys, color='blue', label='Posterior Mean $m_n(t_s)$')
ax.fill_between(tss.flatten(), ls.flatten(), us.flatten(), color='blue',
                alpha=0.25, label='$y_s(t_s)$ 95\% P.I.')
ax.plot(t_s, y_s, '.', label='Scaled Observed Data')
ax.set_xlabel('$t_s$ (Scaled Year)')
ax.set_ylabel('$y_s$ (Scaled $CO_2$ Level)')
ax.set_title("Posterior Prediction of Scaled $CO_2$ Level vs. Scaled Year")
plt.legend(loc='best');
```

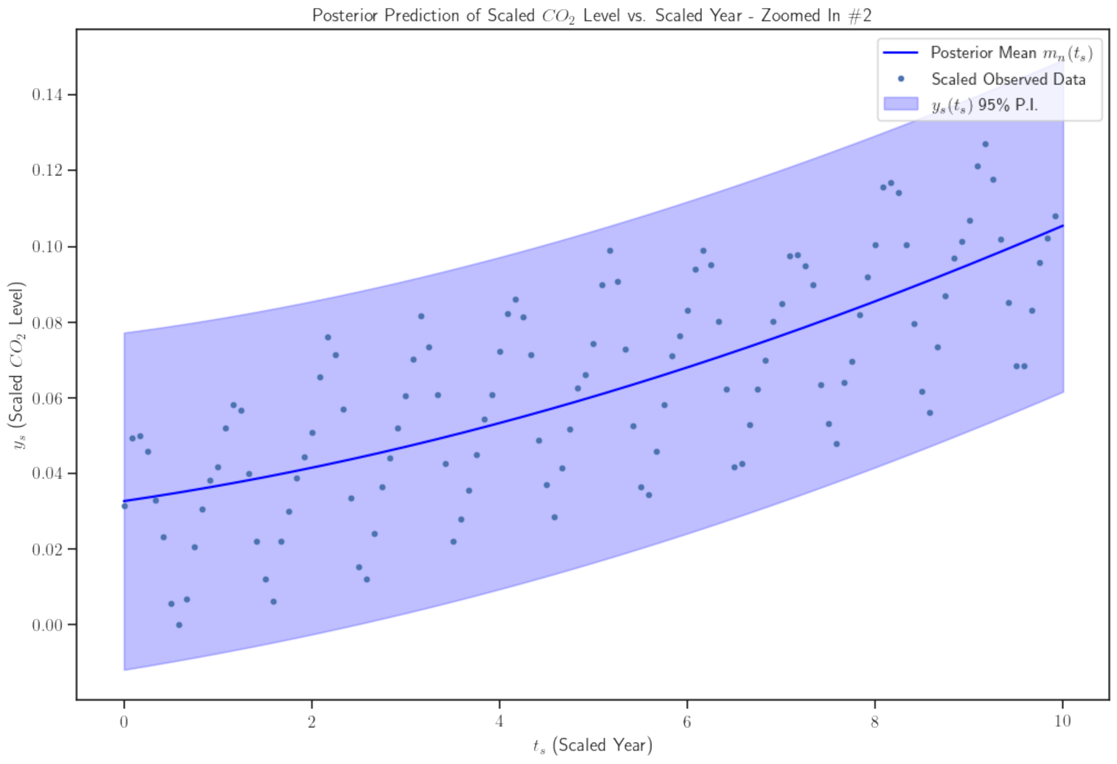


To assist with overall interpretation, visualization of the data, and selection of covariance function parameters, two additional plots are added below.

[18]:
```
# pick the ending point (in scaled years)
end = 50
# generate proper array of scaled time values
tss_second = np.linspace(0, end, 200)[:, None]
# make prediction and calculate 95% predictive interval (P.I.)
ys_second, vs_second = naive_model.predict(tss_second)
ls_second = ys_second - 1.96 * np.sqrt(vs_second)
us_second = ys_second + 1.96 * np.sqrt(vs_second)
# determine number of data points to keep for plotting
keep = len(t_s[t_s < end])
# plot results
fig, ax = plt.subplots(1, figsize = (12, 8), dpi=100)
ax.plot(tss_second, ys_second, color='blue', label='Posterior Mean $m_n(t_s)$')
ax.fill_between(tss_second.flatten(), ls_second.flatten(), us_second.flatten(),␣
 ↪color='blue',
                alpha=0.25, label='$y_s(t_s)$ 95\% P.I.')
ax.plot(t_s[:keep], y_s[:keep], '.', label='Scaled Observed Data')
ax.set_xlabel('$t_s$ (Scaled Year)')
ax.set_ylabel('$y_s$ (Scaled $CO_2$ Level)')
ax.set_title("Posterior Prediction of Scaled $CO_2$ Level vs. Scaled Year -␣
 ↪Zoomed In \#1")
plt.legend(loc='best');
```



[19]:
```
# pick the ending point (in scaled years)
end = 10
# generate proper array of scaled time values
tss_third = np.linspace(0, end, 200)[:, None]
# make prediction and calculate 95% predictive interval (P.I.)
ys_third, vs_third = naive_model.predict(tss_third)
ls_third = ys_third - 1.96 * np.sqrt(vs_third)
us_third = ys_third + 1.96 * np.sqrt(vs_third)
# determine number of data points to keep for plotting
keep = len(t_s[t_s < end])
# plot results
fig, ax = plt.subplots(1, figsize = (12, 8), dpi=100)
ax.plot(tss_third, ys_third, color='blue', label='Posterior Mean $m_n(t_s)$')
ax.fill_between(tss_third.flatten(), ls_third.flatten(), us_third.flatten(),␣
 ↪color='blue',
                alpha=0.25, label='$y_s(t_s)$ 95\% P.I.')
ax.plot(t_s[:keep], y_s[:keep], '.', label='Scaled Observed Data')
ax.set_xlabel('$t_s$ (Scaled Year)')
ax.set_ylabel('$y_s$ (Scaled $CO_2$ Level)')
ax.set_title("Posterior Prediction of Scaled $CO_2$ Level vs. Scaled Year -␣
 ↪Zoomed In \#2")
plt.legend(loc='best');
```



Notice that the squared exponential covariance caputes the long terms, but it fails to capture the seasonal fluctuations. As a matter of fact the seasonabl fluctions as treated as noise. This is clearly false. How can we fix it?

## 1.7 Part B - Improving the prior covariance

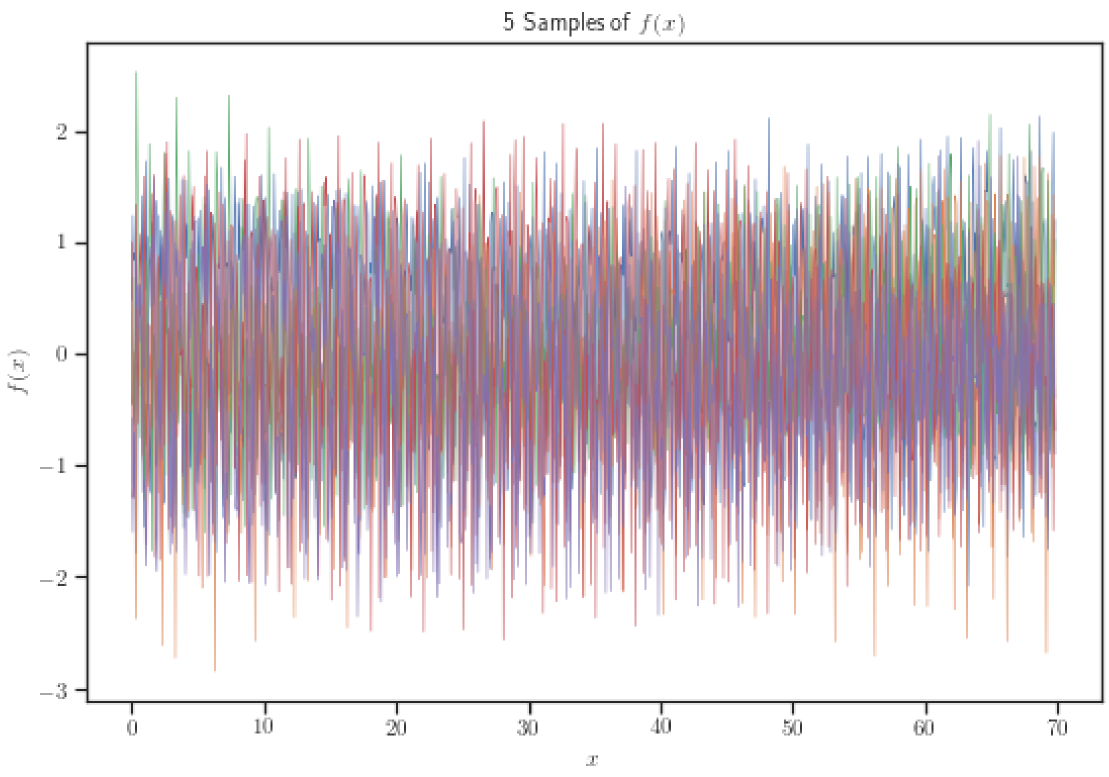
Now use the ideas of Problem 1, to come up with a covariance function that is exhibits the following characteristics clearly visible in the data (call $f(x)$ the scaled CO2 level. + $f(x)$ is smooth + $f(x)$ has a clear trend with a multi-year lengthscale (it is also an increasing trend, but we are not going to impose this) + $f(x)$ has seasonal fluctations with a period of one year + $f(x)$ exhibits small fluctiations within its period.

Use summation and multiplication of simple covariance functions to create a covariance function that exhibits these trends. Sample a few times from it.

Hint: Do not attempt to fit the data in any way. Just try to find a covariance function that has the right features. We also do not care about getting the parameters 100% right at this point. The parameters will be optimized later.

**Answer:**

[20]:
```
# define covariance function to account for seasonal fluctuations
k1 = GPy.kern.StdPeriodic(1)
k1.period = 1.0 # in years
# assume lengthscale of seasonal fluctuations is 0.1
k1.lengthscale = 0.1

# define covariance function to account for small fluctuations within the period
k2 = GPy.kern.RBF(1)
# assuming small fluctuations are approximately 0.1, or 1/10 of the seasonal␣
 ↪period
k2.lengthscale = 0.1
# assume variance of small fluctuations is small
k2.variance = 0.01

# define covariance function to account for the clear trend
k3 = GPy.kern.RBF(1)
# it appears that the trend slightly changes at ~ t_s = 40, so take it to be␣
 ↪the lengthscale
k3.lengthscale = 40.0
# assume that variance of overall trend is fixed at 1
k3.variance.constrain_fixed(1)

# generate a covariance function that exhibits all of the above characteristics␣
 ↪together
improved_k = (k1 + k2) * k3
# Define dense set of x values to plot against
xs = np.linspace(0, t_s.max() + 10, 1000)
# The mean function at xs
ms = np.zeros(xs.shape)
# Sample and plot using function
sample_and_plot(xs, ms, improved_k)
# reference(s): lecture 22 hands-on activities
# https://www.cs.toronto.edu/~duvenaud/cookbook/
```



## 1.8 Part C - Predicting the future

Use a zero mean Gaussian process with the covariance function you picked above to do Gaussian process regression and make the required prediction (ten years after the last observation).

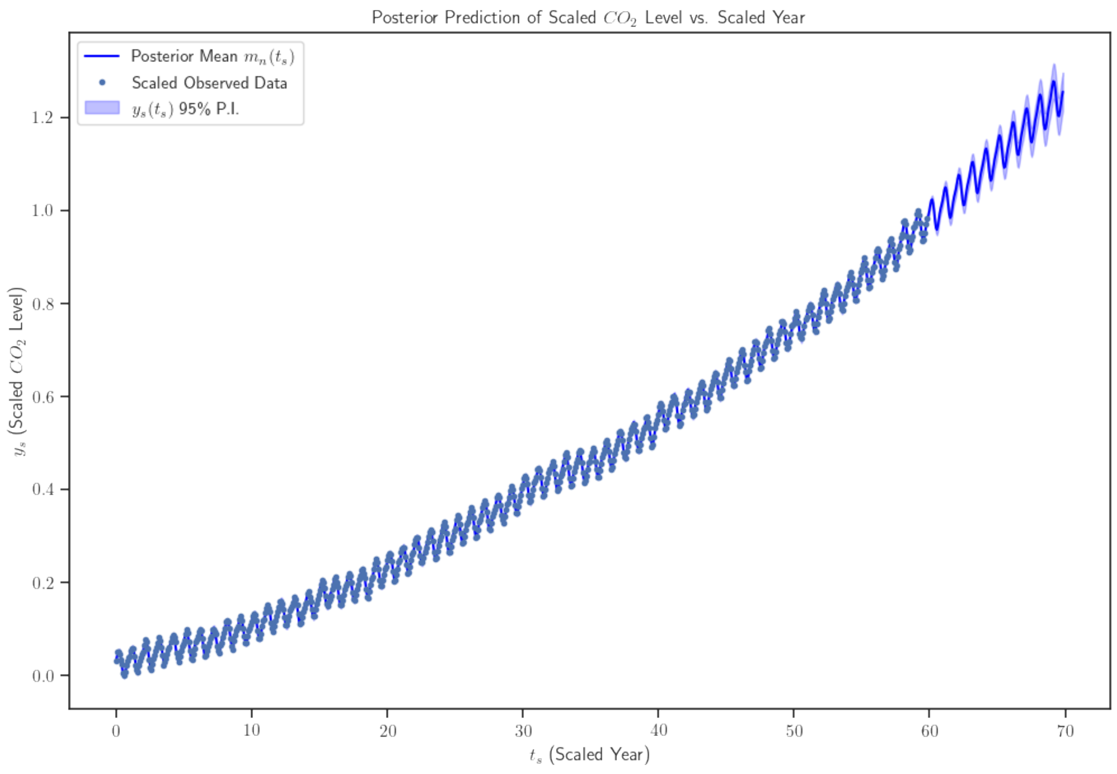
**Answer:**

[21]:
```
# generate improved model
improved_model = GPy.models.GPRegression(t_s, y_s, improved_k)
# find the MAP through optimization (with restarts)
improved_model.optimize_restarts(messages=True)
# display model overview
print(improved_model)
```

```
HBox(children=(VBox(children=(IntProgress(value=0, max=1000), HTML(value=''))), Box(children=(


Optimization restart 1/10, f = -2283.9069027770856

HBox(children=(VBox(children=(IntProgress(value=0, max=1000), HTML(value=''))), Box(children=(


Optimization restart 2/10, f = -1691.4706771782744
```

```
HBox(children=(VBox(children=(IntProgress(value=0, max=1000), HTML(value=''))), Box(children=(


Optimization restart 3/10, f = -2550.6640289414304

HBox(children=(VBox(children=(IntProgress(value=0, max=1000), HTML(value=''))), Box(children=(


Optimization restart 4/10, f = -1691.4706770090966

HBox(children=(VBox(children=(IntProgress(value=0, max=1000), HTML(value=''))), Box(children=(


Optimization restart 5/10, f = -2550.8272400880096

HBox(children=(VBox(children=(IntProgress(value=0, max=1000), HTML(value=''))), Box(children=(


Optimization restart 6/10, f = -2550.8157548896743

HBox(children=(VBox(children=(IntProgress(value=0, max=1000), HTML(value=''))), Box(children=(


Optimization restart 7/10, f = -3045.3989230259235

HBox(children=(VBox(children=(IntProgress(value=0, max=1000), HTML(value=''))), Box(children=(


Optimization restart 8/10, f = -3046.408525445854

HBox(children=(VBox(children=(IntProgress(value=0, max=1000), HTML(value=''))), Box(children=(


Optimization restart 9/10, f = -2905.660999266562

HBox(children=(VBox(children=(IntProgress(value=0, max=1000), HTML(value=''))), Box(children=(


Optimization restart 10/10, f = -1696.3365836891687

Name : GP regression
Objective : -3046.408525445854
Number of Parameters : 8
Number of Optimization Parameters : 7
Updates : True
Parameters:
  GP_regression.                   |                  value  |
constraints  |  priors
  mul.sum.std_periodic.variance    |     0.44486655802029873  |
+ve      |
  mul.sum.std_periodic.period      |      0.9994017580034343  |
+ve      |
  mul.sum.std_periodic.lengthscale  |       3.567009946352724  |
+ve      |
  mul.sum.rbf.variance             |  2.5867370220197497e-05  |
+ve        |
  mul.sum.rbf.lengthscale              |      0.6344137355286277  |
+ve        |
  mul.rbf.variance                     |                     1.0  |    +ve
fixed    |
  mul.rbf.lengthscale                  |       40.41658924203343  |
+ve        |
  Gaussian_noise.variance              |   6.679181747695667e-06  |
+ve        |
```

[22]:
```
tss = np.linspace(0, t_s.max() + 10, 1000)[:, None]
ys, vs = improved_model.predict(tss)
ls = ys - 1.96 * np.sqrt(vs)
us = ys + 1.96 * np.sqrt(vs)
fig, ax = plt.subplots(1, figsize = (12, 8), dpi=100)
ax.plot(tss, ys, color='blue', label='Posterior Mean $m_n(t_s)$')
ax.fill_between(tss.flatten(), ls.flatten(), us.flatten(), color='blue',
                alpha=0.25, label='$y_s(t_s)$ 95\% P.I.')
ax.plot(t_s, y_s, '.', label='Scaled Observed Data')
ax.set_xlabel('$t_s$ (Scaled Year)')
ax.set_ylabel('$y_s$ (Scaled $CO_2$ Level)')
ax.set_title("Posterior Prediction of Scaled $CO_2$ Level vs. Scaled Year")
plt.legend(loc='best');
```



### 1.9 Part D - Bayesian information criterion

As we have seen in earlier lectures, the Bayesian informationc criterion (BIC), see this, can be 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.

Hint: Do a `help(GPy.models.GPRegression)` and you will find a way to get both the number of parameters and the log likelihood. Ask on piazza if you can't find it - or Google it.

**Answer:**

[23]:
```
# help(GPy.models.GPRegression)
# calculate BIC value for naive model and display
BIC_naive = compute_BIC(naive_model, len(t_s))
print(f"Naive BIC: {BIC_naive:0.2f}")
# calculate BIC value for improved model and display
BIC_improved = compute_BIC(improved_model, len(t_s))
print(f"Improved BIC: {BIC_improved:0.2f}")
# reference(s): GPy.models.GPRegression documentation
# https://en.wikipedia.org/wiki/Bayesian_information_criterion
```

```
Naive BIC: -3363.21
Improved BIC: -6046.77
```

The improved model from Part C has a lower BIC score than the naïve model from Part A, therefore it is indeed better and the preferable one to use.

### 1.10 Part E - Plot samples from the posterior Gaussian process

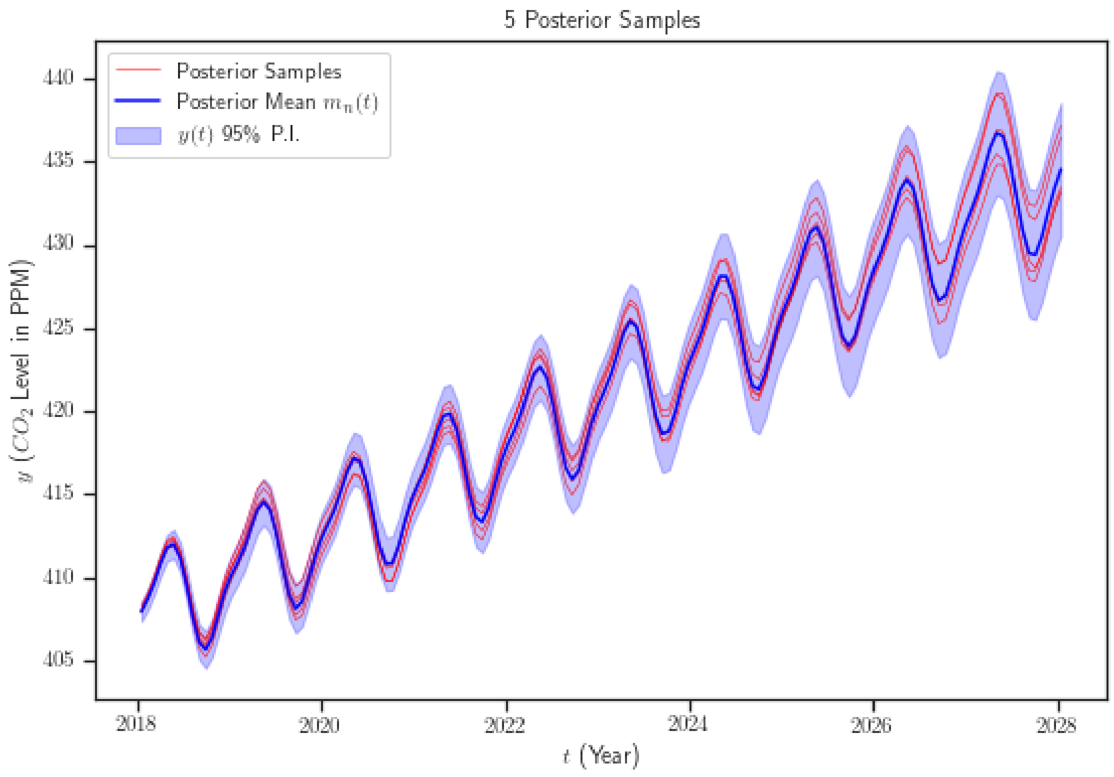
Using the model of Part C, plot 5 samples from the posterior Gaussian process between 2018 and 2028.

Hint: You need to use `GPy.models.GPRegression.posterior_samples_f`.

**Answer:**

[24]:
```
# define the minimum year to display posterior sample data for
minYr = 2018
# define number of posterior samples to take
num_samples = 5
# take posterior samples
f_post_samples = improved_model.posterior_samples_f(tss, num_samples)
```

```
# print(np.shape(f_post_samples))
# get range for plotting in time domain
# convert t back to original scale using known conversion equation
newt = tss + t.min()
# determine number of data points that do not need to be plotted
ignore = len(newt[newt < minYr])
# plotting results
fig, ax = plt.subplots(figsize=(9, 6))
# for each sample:
for i in range(num_samples):
    # plot all relevant sample points that are after the time period to ignore
    ax.plot(newt[ignore:],
            calc_y(f_post_samples[:, :, i], y)[ignore:],
            color='red', lw=0.5)
ax.plot([], [], color='red', lw=0.5, label="Posterior Samples")
# plot the mean
ax.plot(newt[ignore:], calc_y(ys, y)[ignore:], color='blue', label='Posterior␣
 ↪Mean $m_n(t)$')
# plot the 95% epistemic predictive interval (P.I.)
ax.fill_between(
    newt[ignore:].flatten(),
    calc_y(ls, y)[ignore:].flatten(),
    calc_y(us, y)[ignore:].flatten(),
    color='blue',
    alpha=0.25,
    label='$y(t)$ 95\% P.I.'
)
ax.set_xlabel('$t$ (Year)')
ax.set_ylabel('$y$ ($CO_2$ Level in PPM)')
ax.set_title(str(num_samples) + " Posterior Samples")
plt.legend(loc="best");
# reference(s): hands-on activity 22.1
```



### 1.11 Problem 3 - Using Bayesian Global optimization to calibrate an expensive physical model

This is Example 3.1 of (Tsilifis, 2014).

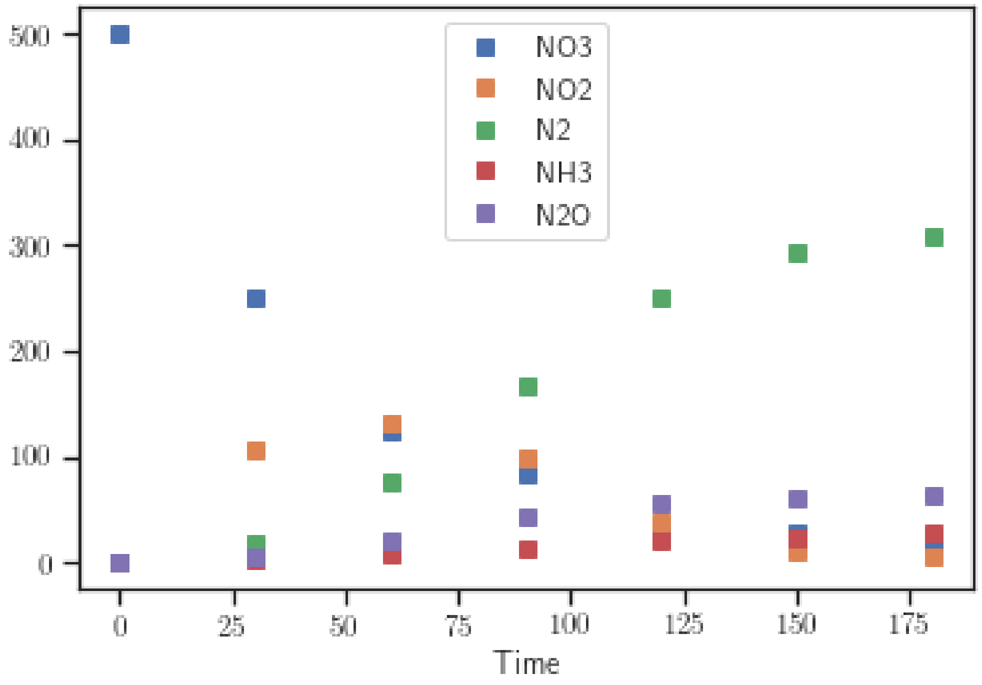
Consider the catalytic conversion of nitrate ($NO_3^-$) to nitrogen ($N_2$) and other by-products by electrochemical means. The mechanism that is followed is complex and not well understood. The experiment of (Katsounaros, 2012) confirmed the production of nitrogen ($N_2$), ammonia ($NH_3$), and nitrous oxide ($N_2O$) as final products of the reaction, as well as the intermediate production of nitrite ($NO_2^-$). The data are reproduced in Comma-separated values (CSV) and stored in catalysis.csv. The time is measured in minutes and the conentrations are measured in mmol $\cdot$ $L^{-1}$. Let's load the data into this notebook using the Pandas Python module:

[25]:
```
url = "https://github.com/PredictiveScienceLab/data-analytics-se/raw/master/
↪lecturebook/data/catalysis.csv"
download(url)
```

[26]:
```
# Load the data
import pandas as pd
catalysis_data = pd.read_csv('catalysis.csv')
catalysis_data
```

```
[26]:    Time     NO3     NO2      N2    NH3    N2O
      0     0  500.00    0.00    0.00   0.00   0.00
      1    30  250.95  107.32   18.51   3.33   4.98
      2    60  123.66  132.33   74.85   7.34  20.14
      3    90   84.47   98.81  166.19  13.14  42.10
      4   120   30.24   38.74  249.78  19.54  55.98
      5   150   27.94   10.42  292.32  24.07  60.65
      6   180   13.54    6.11  309.50  27.26  62.54
```

```
[27]: catalysis_data.plot(style='s', x=0);
```



The theory of catalytic reactions guarantees that the total mass must be conserved. However, this is not the case in our dataset:

```
[28]: catalysis_data.sum(axis=1)
```

```
[28]: 0    500.00
      1    415.09
      2    418.32
      3    494.71
      4    514.28
      5    565.40
      6    598.95
      dtype: float64
```

This inconsistency suggests the existence of an intermediate unobserved reaction product X. (Katsounaros, 2012) suggested that the following reaction path shown in the following figure.

The dynamical system associated with the reaction is:

$$\begin{aligned}
\frac{d\left[\mathrm{NO}_3^-\right]}{dt} &= -k_1\left[\mathrm{NO}_3^-\right],\\
\frac{d\left[\mathrm{NO}_2^-\right]}{dt} &= k_1\left[\mathrm{NO}_3^-\right] - (k_2+k_4+k_5)[\mathrm{NO}_2^-],\\
\frac{d[\mathrm{X}]}{dt} &= k_2\left[\mathrm{NO}_2^-\right] - k_3[X],\\
\frac{d[\mathrm{N}_2]}{dt} &= k_3\left[\mathrm{X}\right],\\
\frac{d[\mathrm{NH}_3]}{dt} &= k_4\left[\mathrm{NO}_2^-\right],\\
\frac{d[\mathrm{N}_2O]}{dt} &= k_5\left[\mathrm{NO}_2^-\right],
\end{aligned}$$

where $[\cdot]$ denotes the concentration of a quantity, and $k_i > 0$, $i = 1, ...5$ are the *kinetic rate constants.*

In this problem, I am going to guide you through the calibration of the parameters of this model so that we match the observations. These problems are also known as *inverse problems.* The problem can, and should, be formulated in a Bayesian way. However, in this homework problem we are going to do it using a classical loss-minimization approach. We will discuss the Bayesian approach for calibrating the same model in a later lecture.

Before you proceed, please read a little bit about the "classical theory of inverse problems:"

### 1.11.1 Classical theory of inverse problems

Suppose that you have a model (any model really) that predicts a quantity of interest. Let's assume that this model has parameters that you do not know. These parameters could be simple scalars (mass, spring constant, dumping coefficients, etc.) or it could be also be functions (initial conditions, boundary values, spatially distributed constitutive relations, etc.) Let's denote all these parameters with the vector $x$. Assume that:

$$x \in \mathcal{X} \subset \mathbb{R}^d.$$

Now, let's say we perform an experiment that measures a *noisy* vector:

$$y \in \mathcal{Y} \subset \mathbb{R}^m.$$

Assume that, you can use your model *model* to predict $y$. It does not matter how complicated your model is. It could be a system of ordinary differential or partial differential equations, or something more complicated. If it predicts $y$, you can always think of it as a function from the unknown parameter space $\mathcal{X}$ to the space of $y$'s, $\mathcal{Y} \subset \mathbb{R}^m$. That is, you can think of it as giving rise to a function:

$$f : \mathcal{X} \to \mathcal{Y}.$$

The **inverse problem**, otherwise known as the **model calibration** problem is to find the best $x \in \mathcal{X}$ so that:

$$f(x) \approx y.$$

### 1.11.2 Formulation of Inverse Problems as Optimization Problems

Saying that $f(x) \approx y$ is not an exact mathematical statement. What does it really mean for $f(x)$ to be close to $y$? To quantify this, let us introduce a *loss metric*:

$$\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R},$$

such that $\ell(f(x), y)$ is how much our prediction is off if we chose the input $x \in \mathcal{X}$. Equiped with this loss metric, we can formulate the mathematical problem as:

$$\min_{x \in \mathcal{X}} \ell(f(x), y).$$

**The Square Loss** The choice of the metric is somewhat subjective (it depends on what it means to be wrong in your problem). However, a very common assumption is that to take the *square loss*:

$$\ell(f(x), y) = \frac{1}{2} \parallel f(x) - y \parallel_2^2 = \frac{1}{2} \sum_{i=1}^{m} (f_i(x) - y_i)^2.$$

For this case, the inverse problem can be formulated as:

$$\min_{x \in \mathcal{X}} \frac{1}{2} \parallel f(x) - y \parallel_2^2.$$

**Solution Methodologies** We basically have to solve an optimization problem. For the square loss function, if $f(x)$ is linear, then you get the classic least squares problem which has a known solution. Otherwise, you get what is known as *generalized least squares*. There are many algorithms that you could use this problem. Several are implemented in scipy.optimize. If you are able to implement your model as a simple python function, then you can use them. Alternatively, and this is what we are going to do here, we could use Bayesian global optimization instead. The absolutely, essential thing that you need to provide to these methods is the function they are optimizing, i.e.,

$$L(x, y) = \ell(f(x), y).$$

### 1.11.3 Back to the catalysis model

Let's now formulate the calibration problem for the catalysis model. We proceed in several steps.

**Step 1: Making our life easier by simplifying the notation** Note that this is actually a linear system. To simplify our notation, let's define:

$$\begin{aligned} z_1 &:= \left[\mathrm{NO}_3^-\right], \\ z_2 &:= \left[\mathrm{NO}_2^-\right], \\ z_3 &:= [\mathrm{X}], \\ z_4 &:= [\mathrm{N}_2], \\ z_5 &:= [\mathrm{NH}_3], \\ z_6 &:= [\mathrm{N}_2O], \end{aligned}$$

the vector:

$$z = (z_1, z_2, z_3, z_4, z_5, z_6),$$

and the matrix:

$$A(k_1, \dots, k_5) = \begin{pmatrix} -k_1 & 0 & 0 & 0 & 0 & 0 \\ k_1 & -(k_2 + k_4 + k_5) & 0 & 0 & 0 & 0 \\ 0 & k_2 & -k_3 & 0 & 0 & 0 \\ 0 & 0 & k_3 & 0 & 0 & 0 \\ 0 & k_4 & 0 & 0 & 0 & 0 \\ 0 & k_5 & 0 & 0 & 0 & 0 \end{pmatrix} \in \mathbb{R}^{6\times 6}.$$

With these definitions, the dynamical system becomes:

$$\dot{z} = A(k_1, \dots, k_5)z,$$

with initial conditions

$$z(0) = z_0 = (500, 0, 0, 0, 0, 0) \in \mathbb{R}^6,$$

read directly from the experimental data. What we are definitely going to need is a solver for this system. That's easy. Let's denote the solution of the system at time $t$ by:

$$z(t; k_1, \dots, k_5).$$

**Step 2: Scale the unknown parameters to your best of your abilities** The constraints you have on your parameters, the better. If you do have constraints, you would have to use constrained optimization algorithms. The way you scale things depend on the problem. Here we would think as follows:

- $k_i$ has units of inverse time. It is propary appropriate to scale it with the total time which is 180 minutes. So, let's just multiply $k_i$ with 180. This makes the resulting variable dimensionless:

$$\hat{x}_i = 180k_i.$$

- $k_i$ is positive, therefore $\hat{x}_i$ must be positive. So, let's just work with the logarithm of $\hat{x}_i$:

$$x_i = \log \hat{x}_i = \log 180k_i.$$

- define the parameter vector:

$$x = (x_1, \dots, x_5) \in \mathcal{X} = \mathbb{R}^5.$$

From now on, we will write

$$A = A(x),$$

for the matrix of the dynamical system, and

$$z = z(t; x),$$

for the solution at $t$ given that the parameters are $x$.

**Step 3: Making the connection between our model and the experimental measurements** Our experimental data include measurements of everything except $z_3$ at times six (6) time instants:

$$t_j = 30j \text{ minutes},$$

$j = 1, \dots, 6$.

Now, let $Y \in \mathbb{R}^{5\times 6}$ be the experimental measurements:

[29]:
```
catalysis_data[1:]
```

[29]:
```
   Time     NO3     NO2      N2    NH3    N2O
1    30  250.95  107.32   18.51   3.33   4.98
2    60  123.66  132.33   74.85   7.34  20.14
3    90   84.47   98.81  166.19  13.14  42.10
4   120   30.24   38.74  249.78  19.54  55.98
5   150   27.94   10.42  292.32  24.07  60.65
6   180   13.54    6.11  309.50  27.26  62.54
```

You can think of the measurements as vector by flattening the matrix:

$$y = \operatorname{vec}(Y) \in \mathbb{R}^{30}.$$

Note that `vec` is the vectorization operator.

What is the connection between the solution of the dynamical system $z(t, x)$ and the experimental data? It is as follows:

$$\begin{array}{rcl}
z_1(30j; x) & \longrightarrow & Y_{j1}, \\
z_2(30j; x) & \longrightarrow & Y_{j2}, \\
z_4(30j; x) & \longrightarrow & Y_{j3}, \\
z_5(30j; x) & \longrightarrow & Y_{j4}, \\
z_6(30j; x) & \longrightarrow & Y_{j5},
\end{array}$$

for $j = 1, \dots, 6$.

We are now ready to define a function:

$$f : \mathcal{X} \to \mathcal{Y} = \mathbb{R}^{30}_{+},$$

as follows: + Define the matrix function:

$$F : \mathcal{X} \to \mathbb{R}^{5\times 6},$$

by:

$$\begin{array}{rcccl}
F_{j1}(x) & = & z_1(30j; x) & \longrightarrow & Y_{j1}, \\
F_{j2}(x) & = & z_2(30j; x) & \longrightarrow & Y_{j2}, \\
F_{j3}(x) & = & z_4(30j; x) & \longrightarrow & Y_{j3}, \\
F_{j4}(x) & = & z_5(30j; x) & \longrightarrow & Y_{j4}, \\
F_{j5}(x) & = & z_6(30j; x) & \longrightarrow & Y_{j5},
\end{array}$$

- And flatten that function:

$$f(x) = \text{vec}(F(x)) \in \mathbb{R}^{30}.$$

Now, we have made the connection with our theoretical formulation of inverse problems crystal clear.

**Step 4: Programming our ODE solver and the loss function**

[30]:
```
import scipy.integrate

def A(x):
    """
    Return the matrix of the dynamical system.
    """
    # Scale back to the k's
    k = np.exp(x) / 180.
    res = np.zeros((6,6))
    res[0, 0] = -k[0]
    res[1, 0] = k[0]
    res[1, 1] = -(k[1] + k[3] + k[4])
    res[2, 1] = k[1]
    res[2, 2] = -k[2]
    res[3, 2] = k[2]
    res[4, 1] = k[3]
    res[5, 1] = k[4]
    return res


def g(z, t, x):
    """
    The right hand side of the dynamical system.
    """
    return np.dot(A(x), z)


# The initial conditions
z0 = np.array([500., 0., 0., 0., 0., 0.])


# The times at which we need the solution (experimental times)
t_exp = np.array([30. * j for j in range(1, 7)])

# The experimental data as a matrix
Y = catalysis_data[1:].values[:, 1:]

# The experimental as a vector
y = Y.flatten()

# The full solution of the dynamical system
def Z(x, t):
    """
    Returns the solution for parameters x at times t.
    """
    return scipy.integrate.odeint(g, z0, t, args=(x,))


# The matrix function F (matches to Y)
def F(x, t):
    res = Z(x, t)
    return np.hstack([res[:, :2], res[:, 3:]])


# The function f (matches to y)
def f(x, t):
    return F(x, t).flatten()
```

```
# Finally, the loss function that we need to minimize over x:
def L(x, t, y):
    return 0.5 * np.sum((f(x, t) / 500. - y / 500.) ** 2) # We scale for␣
 ↪numerical stability
```

**Step 5: Minimize the loss function** Let's optimize with scipy.optimize:

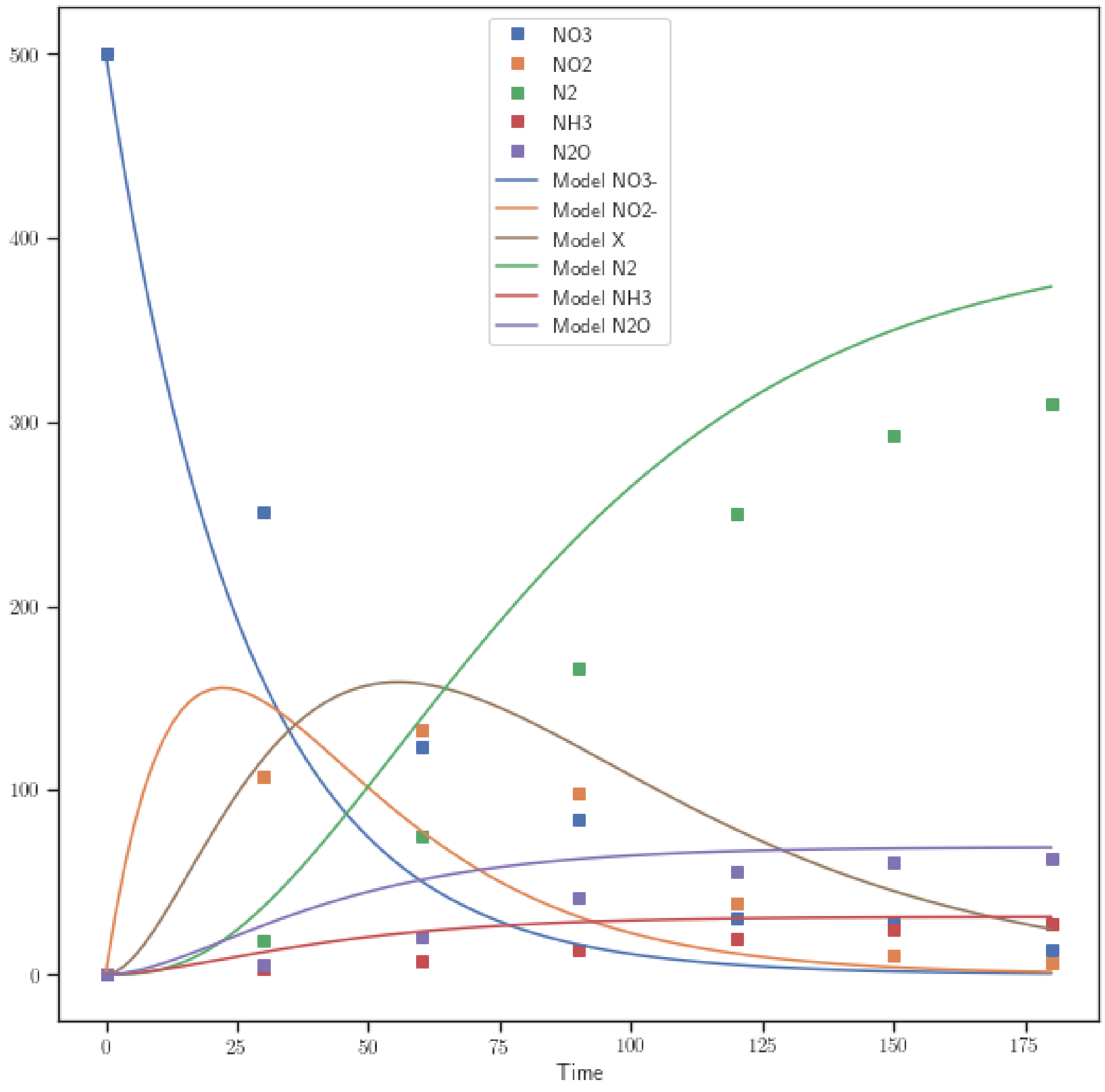
[31]:
```
import scipy.optimize

# Initial guess for x
x0 = -2.0 + 2.0 * np.random.rand(5)

# Optimize
res = scipy.optimize.minimize(L, x0, args=(t_exp, y))

print(res)
```

```
      fun: 0.16530885870677303
 hess_inv: array([[  7.556,   5.076,   1.849,  32.151,  11.174],
       [  5.076,  32.261,   8.227, 116.528,  50.631],
       [  1.849,   8.227,  10.149,  39.751,  22.752],
       [ 32.151, 116.528,  39.751, 523.655, 205.528],
       [ 11.174,  50.631,  22.752, 205.528, 104.591]])
      jac: array([ 8.129e-06,  6.706e-07,  7.402e-06, -4.493e-06,  9.507e-06])
  message: 'Optimization terminated successfully.'
     nfev: 384
      nit: 49
     njev: 64
   status: 0
  success: True
        x: array([ 1.926,  2.019,  1.425, -0.526,  0.265])
```

And here is how you can visualize the model with the "best" parameters:

[32]:
```
x = res.x
t = np.linspace(0, 180, 100)
x1 = np.array([1.359, 1.657, 1.347, -.16, -1.01])
Yp = Z(x, t)

fig, ax = plt.subplots(figsize=(10, 10))
catalysis_data.plot(ax=ax, style='s', x=0)
ax.plot(t, Yp[:, 0], color=sns.color_palette()[0], label='Model NO3-')
ax.plot(t, Yp[:, 1], color=sns.color_palette()[1], label='Model NO2-')
ax.plot(t, Yp[:, 2], color=sns.color_palette()[5], label='Model X')
ax.plot(t, Yp[:, 3], color=sns.color_palette()[2], label='Model N2')
ax.plot(t, Yp[:, 4], color=sns.color_palette()[3], label='Model NH3')
ax.plot(t, Yp[:, 5], color=sns.color_palette()[4], label='Model N2O')
```

```
plt.legend();
```



Note that the code above will not work every time... Some times it will work and sometimes it won't work. Run it 3-4 times if it accidentally works. There are several problems. Here are the three most relevant in our context: + `scipy.optimize` needs the gradient of the loss function. Since we do not provide it, it tries to get it using numerical differentiation. Numerical differentiation introduces errors... + `scipy.optimize` find a local minimum of the loss function. This may actually be a bad local minimum. + okay, this particular model is not very computationally expensive. But imagine trying to calibrate a model that takes a while for a single evaluation (e.g., a finite element model). Then, using scipy.optimize (especially without supplying the derivatives), is doomed to fail.

To overcome these difficulties, you have to use Bayesian global optimization to solve the problem. Note that in the hands-on activities, we introduced this code:

```
[33]: def ei(m, sigma, ymax, psi=0.):
          u = (m - ymax) / sigma
          ei = sigma * (u * st.norm.cdf(u) + st.norm.pdf(u))
          ei[sigma <= 0.] = 0.
          return ei

      def maximize(f, gpr, domain, num_candidates=10000,
                   alpha=ei, psi=0., max_it=6):
          """
          Optimize f using a limited number of evaluations.

          :param f:          The function to optimize.
          :param gpr:        A Gaussian process model to use for representing our state␣
       ↪of knowldege.
          :param X_design: The set of candidate points for identifying the maximum.
          :param alpha:      The acquisition function.
          :param psi:        The parameter value for the acquisition function (not used␣
       ↪for EI).
          :param max_it:     The maximum number of iterations.
          """
          af_all = []
          print('Iteration\tCurrent best objective \tCurrent acquisition func. value')
          dim = gpr.X.shape[1]
          for count in range(max_it):
              X_design = domain[:, 0] + \
                         (domain[:, 1] - domain[:, 0]) * \
                              np.random.rand(num_candidates, dim)
              m, sigma2 = gpr.predict(X_design)
              sigma = np.sqrt(sigma2)
              af_values = alpha(m, sigma, gpr.Y.max(), psi=psi)
              i = np.argmax(af_values)
              X = np.vstack([gpr.X, X_design[i:(i+1), :]])
              y = np.vstack([gpr.Y, [f(X_design[i, :])]])
              gpr.set_XY(X, y)
              # Uncomment the following to optimize the hyper-parameters
              gpr.optimize()
              idx_opt = np.argmax(gpr.Y.flatten())
              f_opt = gpr.Y[idx_opt, 0]
              print('{0:d}\t\t{1:1.2f}\t\t\t{2:1.2f}'.format(count + 1, f_opt,␣
       ↪af_values[i, 0]))
          x_opt = np.array(gpr.X[idx_opt])
          return x_opt, f_opt, gpr
```

The code *maximizes* a function, but you want to *minimize* the loss. To recast the problem as a maximization problem, you need to work with *minus the loss*. Also, the code does not allow for a function with extra parameters (like the `t_exp` and the `y` we have for `L`). Here is the function that you should be optimizing:

[34]:
```
h = lambda x: -L(x, t_exp, y)
```

## 1.12 Part A - Perform multivariate Gaussian process regression on an initial set of data

We are going to search for the best parameters $x$ within the set $[-2, 2]^5$. Consider the following two datasets consisting of parameter and minus loss pairs:

[35]:
```
# Initial training points
n_init_train = 100
X_init_train = -2.0 + 4.0 * np.random.rand(n_init_train, 5)
Y_init_train = np.array([h(x) for x in X_init_train])[:, None]
```

Use a squared exponential covariance function with automatic relevance determination to do Gaussian process regression with `X_init_train` and `Y_init_train`.

Hint: You may want to experiment by constraining the likelihood noise of your model to be very small, say $10^{-6}$. This is because the observations of the loss do not really have any noise.

**Answer:**

[36]:
```
# generate covariance kernel of proper dimension, with ARD turned on
k = GPy.kern.RBF(5, ARD=True)
# generate regression model
gp = GPy.models.GPRegression(X_init_train, Y_init_train, k)
# constraining the likelihood noise of the model
gp.likelihood.variance.constrain_fixed(1e-6);
# reference(s): lecture 22 hands-on activities
```

## 1.13 Part B - Inspecting your model

Use the lengthscale information to rank the model parameters according their effect on the calibration loss.

**Answer:**

[37]:
```
# find the MAP through optimization
gp.optimize(messages=True)
```

```
HBox(children=(VBox(children=(IntProgress(value=0, max=1000), HTML(value=''))), Box(children=(
```

[37]:
```
<paramz.optimization.optimization.opt_lbfgsb at 0x7f8f1d8812d0>
```

[38]:
```
# display the model parameters
print('Model Parameters:')
print(gp.kern.lengthscale)
# sort the parameters based on lengthscale
```

```
lengthscales = gp.kern.lengthscale
lenList = [(idx, val) for idx, val in enumerate(lengthscales)]
sortedList = sorted(lenList, key=lambda item: item[1])
# display sorted results
print('Ranked Model Parameters:')
print('Note: smaller lengthscale indicates more importance and more effect on␣
↪the calibration loss)')
print('rank | index | lengthscale')
for i, j in enumerate(sortedList):
  print(i+1, '   |', sortedList[i][0], '    |', sortedList[i][1])
# reference(s): hands-on activity 22.4
```

```
Model Parameters:
  index  |  GP_regression.rbf.lengthscale  |  constraints  |  priors
  [0]    |                          1.899  |      +ve      |
  [1]    |                          4.310  |      +ve      |
  [2]    |                         13.816  |      +ve      |
  [3]    |                          4.365  |      +ve      |
  [4]    |                          3.242  |      +ve      |
Ranked Model Parameters:
Note: smaller lengthscale indicates more importance and more effect on the
calibration loss)
rank | index | lengthscale
1    | 0     | 1.8988040935355988
2    | 4     | 3.242174653476227
3    | 1     | 4.310070319886707
4    | 3     | 4.364675375761548
5    | 2     | 13.816057950901437
```

## 1.14 Part C - Diagnostics

Here are some test data:

[39]:
```
# Test points
n_test = 50
X_test = -2.0 + 4.0 * np.random.rand(n_test, 5)
Y_test = np.array([h(x) for x in X_test])[:, None]
```

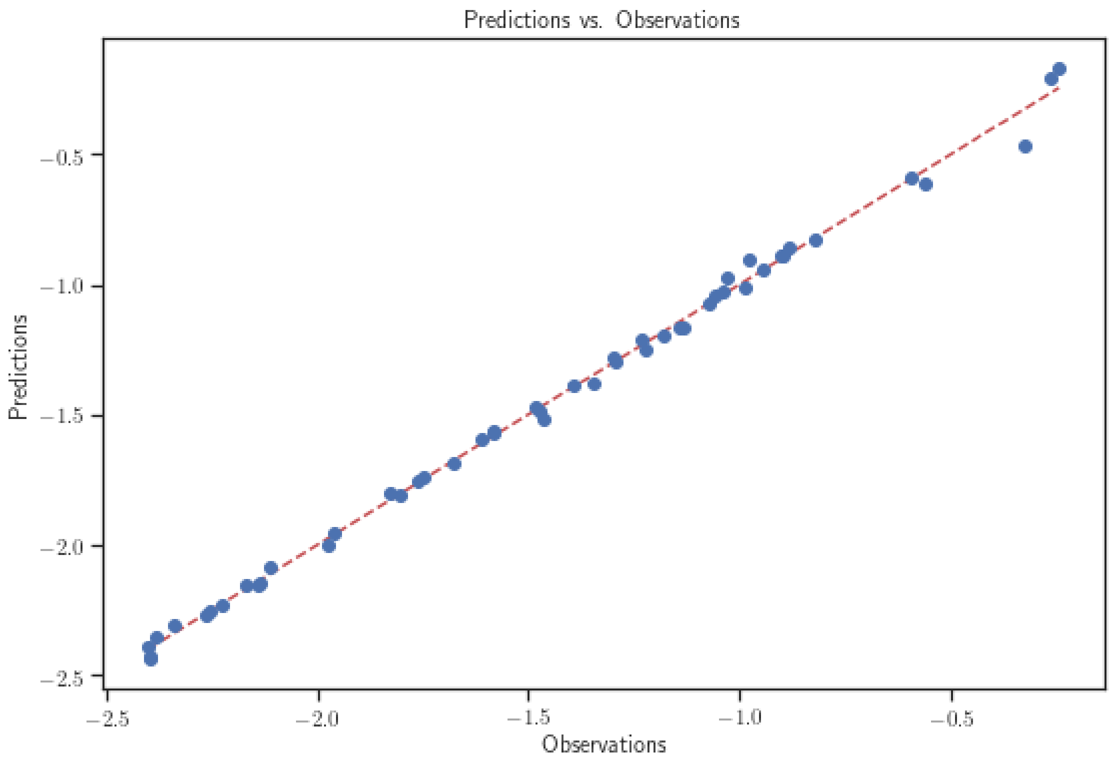
Do the following:

- Predictions vs observations plot
- Standarized errors plot

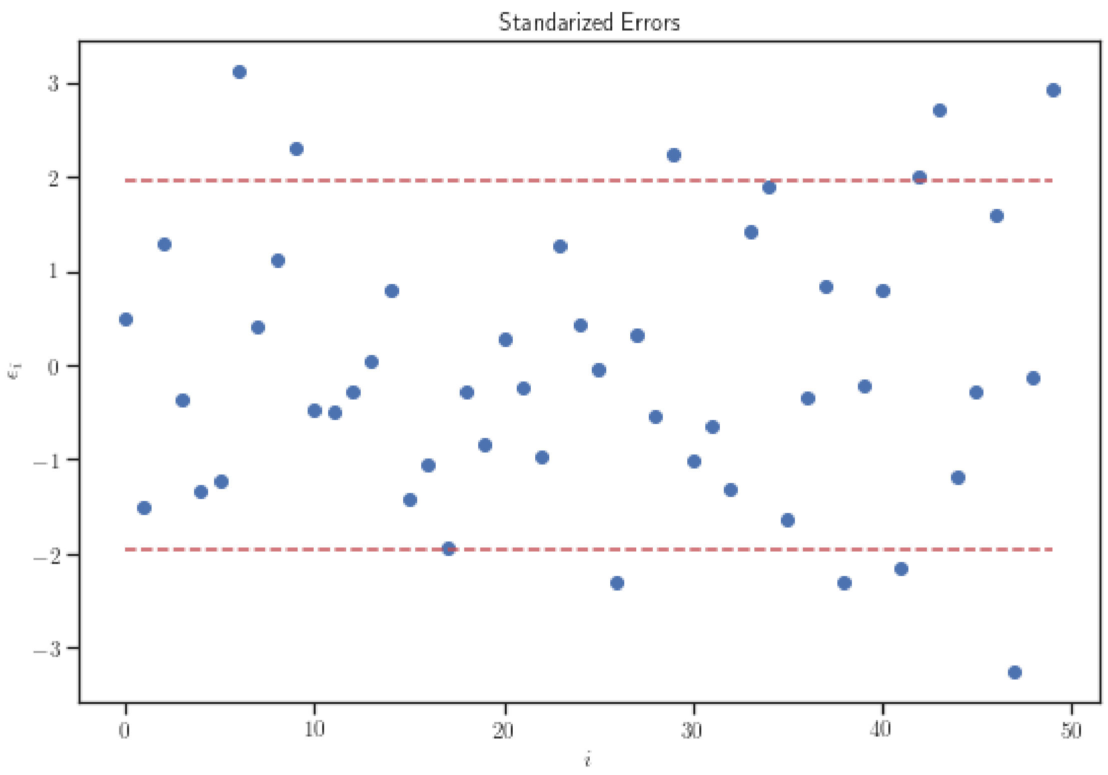
**Answer:**

[40]:
```
# Predictions vs. Ovservations Plot
# make prediction
Y_test_m, Y_test_v = gp.predict(X_test, full_cov=False)
```

```
# plot result
fig, ax = plt.subplots(figsize=(9, 6))
ys = np.linspace(np.min(Y_test), np.max(Y_test), 100)
ax.plot(ys, ys, 'r--', label='$x=y$')
ax.plot(Y_test, Y_test_m, 'o')
ax.set_xlabel('Observations')
ax.set_ylabel('Predictions')
ax.set_title("Predictions vs. Observations");
# reference(s): hands-on activity 22.2
```



[41]:
```
# Standarized Errors Plot
# compute standarized errors
eps = (Y_test - Y_test_m) / np.sqrt(Y_test_v)
# plot results
fig, ax = plt.subplots(figsize=(9, 6))
ax.plot(eps, 'o', label='Standarized Errors')
ax.plot(np.arange(eps.shape[0]), 1.96 * np.ones(eps.shape[0]), 'r--')
ax.plot(np.arange(eps.shape[0]), -1.96 * np.ones(eps.shape[0]), 'r--')
ax.set_xlabel('$i$')
ax.set_ylabel('$\epsilon_i$')
ax.set_title("Standarized Errors");
# reference(s): hands-on activity 22.2
```



## 1.15 Part D - Calibrate the model with Bayesian global optimization

Now use Bayesian global optimization with expected improvement to calibrate your model using the GP that you built above as the starting point. Do not expect this to give you a perfect model. But it will be better than nothing. We will get the best possible model in the next homework assignment.

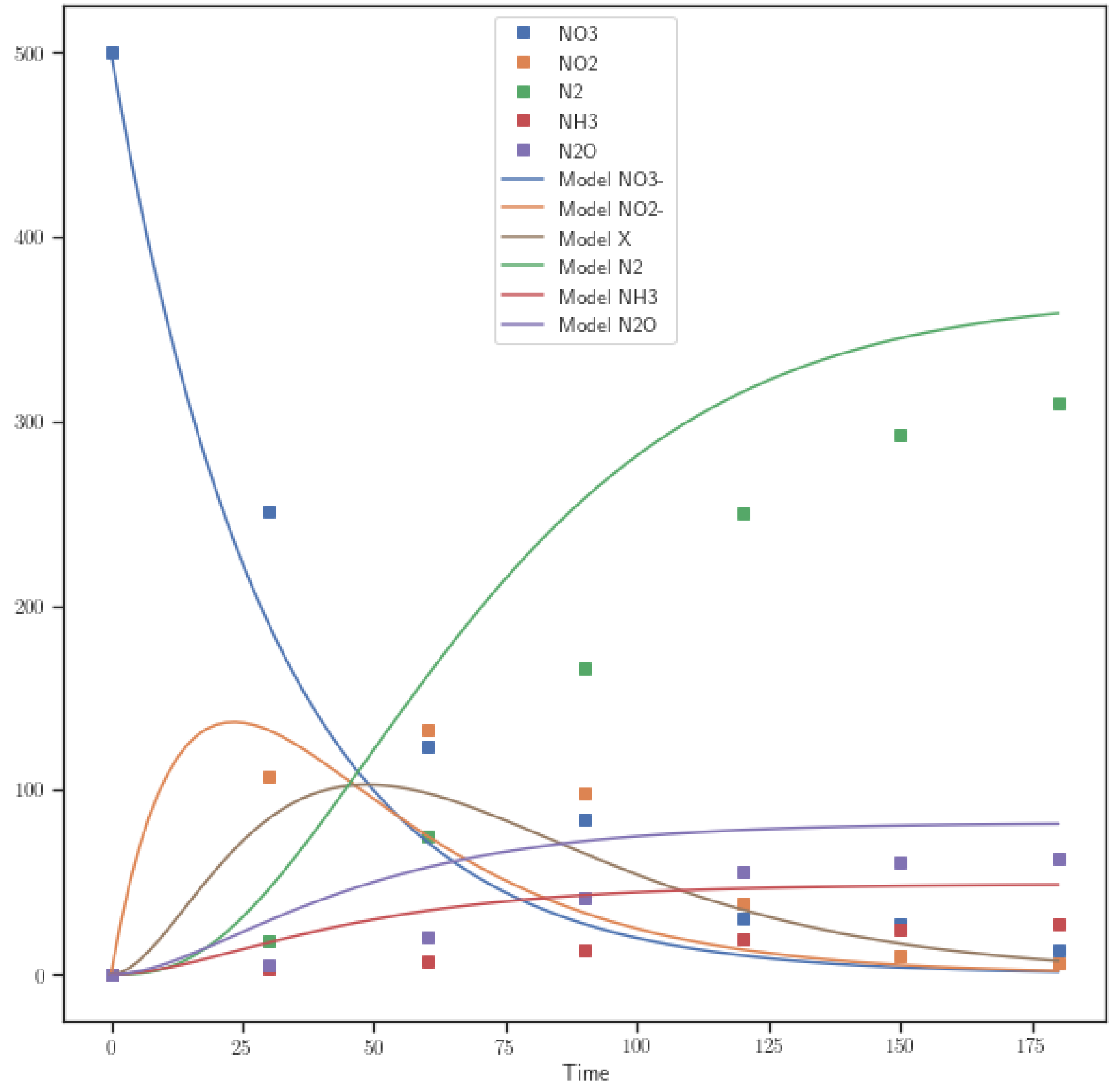
**Hint:** Here you basically need to read the docstring of `maximize` and use it correctly.

[42]:
```
# For your convenience, the `domain` argument of minimize should be:
domain = np.array([[-2, 2], [-2, 2], [-2, 2], [-2, 2], [-2, 2]])
# Run maximize here:
x_opt, f_opt, gpr = maximize(h, gp, domain)
```

```
Iteration         Current best objective  Current acquisition func. value
1                 -0.19                   0.42
2                 -0.19                   0.09
3                 -0.19                   0.05
4                 -0.17                   0.01
5                 -0.17                   0.00
6                 -0.17                   0.00
```

Use this code to plot your calibrated model:

[43]:
```
x = x_opt
t = np.linspace(0, 180, 100)
x1 = np.array([1.359, 1.657, 1.347, -.16, -1.01])
Yp = Z(x, t)

fig, ax = plt.subplots(figsize=(10, 10))
catalysis_data.plot(ax=ax, style='s', x=0)
ax.plot(t, Yp[:, 0], color=sns.color_palette()[0], label='Model NO3-')
ax.plot(t, Yp[:, 1], color=sns.color_palette()[1], label='Model NO2-')
ax.plot(t, Yp[:, 2], color=sns.color_palette()[5], label='Model X')
ax.plot(t, Yp[:, 3], color=sns.color_palette()[2], label='Model N2')
ax.plot(t, Yp[:, 4], color=sns.color_palette()[3], label='Model NH3')
ax.plot(t, Yp[:, 5], color=sns.color_palette()[4], label='Model N2O')
plt.legend();
```


NO3
NO2
N2
NH3
N2O
Model NO3-
Model NO2-
Model X
Model N2
Model NH3
Model N2O
Time