homework_08

November 23, 2021

1 Homework 8

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 can either:
  - Type the answer using the built-in latex capabilities. In this case, simply export the notebook as a pdf and upload it on gradescope; or
  - You can print the notebook (after you are done with all the code), write your answers by hand, scan, turn your response to a single pdf, and upload on gradescope.
- The total homework points are 100. Please note that the problems are not weighed equally.

Note: Please match all the pages corresponding to each of the questions when you submit on gradescope.

1.3 Student details

- **First Name:** Alex
- **Last Name:** Shank
- **Email:** shank14@purdue.edu

[1]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
import pandas as pd
import seaborn as sns
sns.set_context('paper')
sns.set_style('white')
import scipy.stats as st
# A helper function for downloading files
import requests
import os
```

```
def download(url, local_filename=None):
    """
    Downloads the file in the ``url`` and saves it in the current working␣
 ↪directory.
    """
    data = requests.get(url)
    if local_filename is None:
        local_filename = os.path.basename(url)
    with open(local_filename, 'wb') as fd:
        fd.write(data.content)
try:
  import GPy
except:
  _=!pip install GPy
  import GPy
import scipy.stats as st
```

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 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 it. For each of the cases below: + assume that $f \sim \mathrm{GP}(m, k)$ 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:

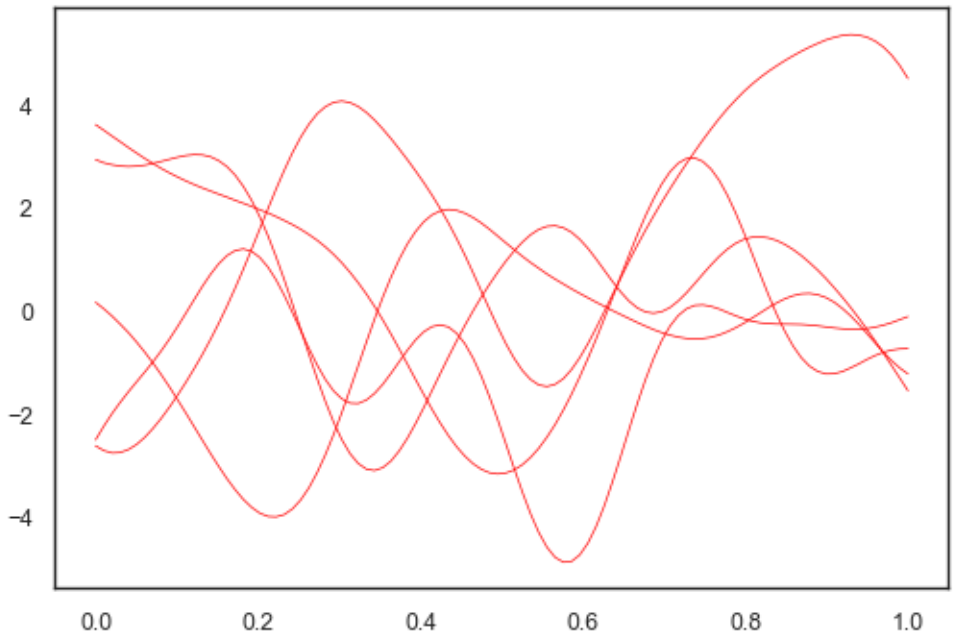
$$1.96 * \sigma = 4$$

$$s^2 = k(x, x) = \mathbb{V}[f(x)] = \sigma^2 = (\frac{4}{1.96})^2 = 4.165,$$

and lengthscale $\ell = 0.1$. We chose the variance to be 4.165 so that with (about) 95% probability the values of $f(x)$ are between -4 and 4.

[2]:
```
# Define the covariance function
k = GPy.kern.RBF(1)
k.lengthscale = 0.1
k.variance = np.square(4.0 / 1.96)
print(k.variance)
# Sample
xs = np.linspace(0, 1, 100)
# The mean function at xs
ms = np.zeros(xs.shape)
# 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(dpi=100)
ax.plot(xs, f_samples.T, 'r', lw=0.5);
```

```
index  |  rbf.variance  |  constraints  |  priors
[0]    |    4.16493128  |      +ve      |
```



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:

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

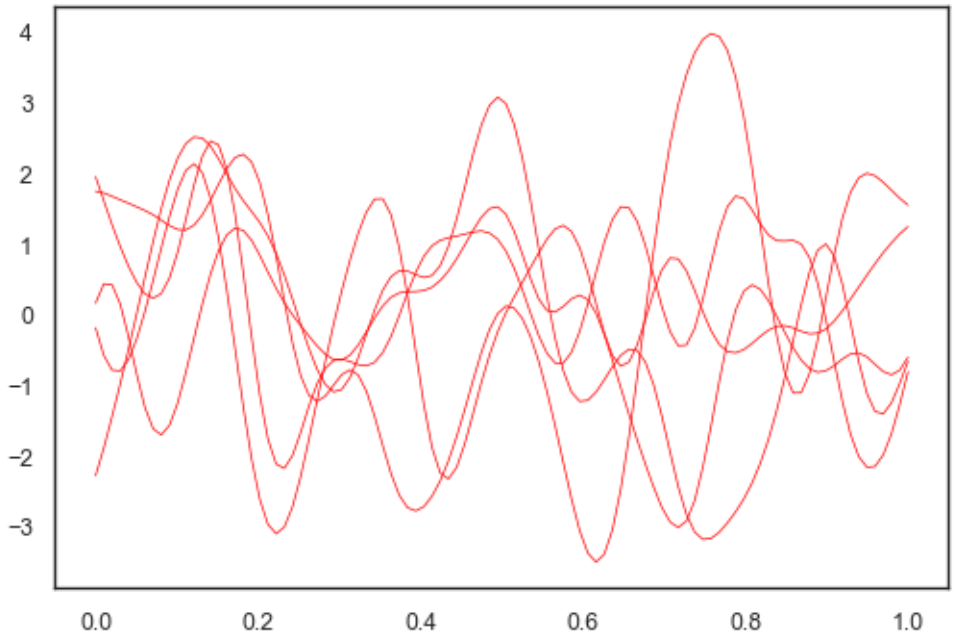
with variance:

$$1.96 * \sigma = 3$$

$$s^2 = k(x, x) = \mathbb{V}[f(x)] = \sigma^2 = (\frac{3}{1.96})^2 = 2.345,$$

and lengthscale $\ell = 0.05$. We chose the variance to be 2.345 so that with (about) 95% probability the values of $f(x)$ are between -3 and 3.

[3]:
```
# Define the covariance function
k = GPy.kern.RBF(1)
k.lengthscale = 0.05
k.variance = np.square(3.0 / 1.96)
# Sample
xs = np.linspace(0, 1, 100)
# The mean function at xs
ms = np.zeros(xs.shape)
# 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(dpi=100)
ax.plot(xs, f_samples.T, 'r', lw=0.5);
```



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 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:

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

with variance:

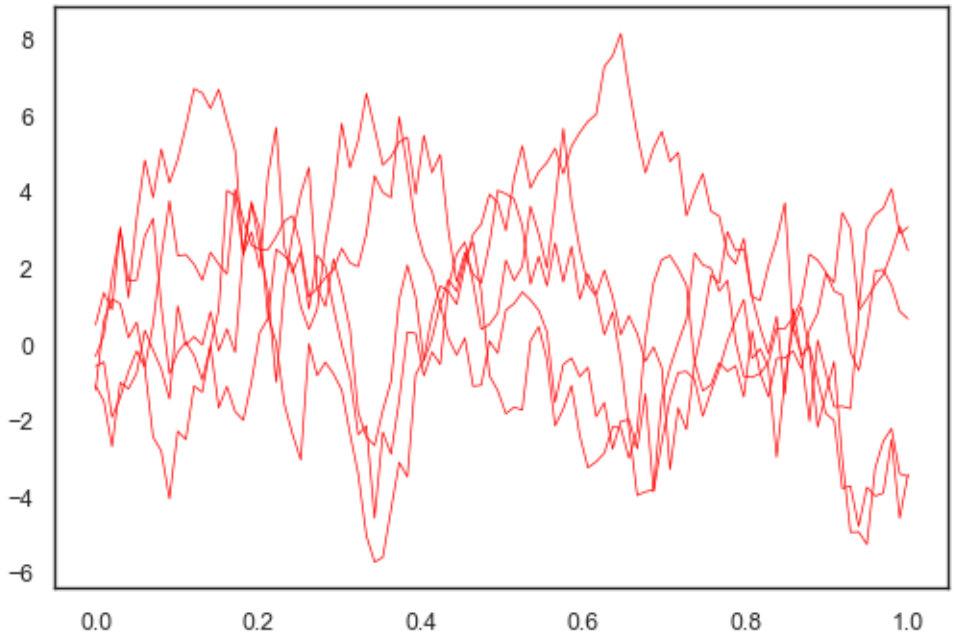
$$1.96 * \sigma = 5$$

$$s^2 = k(x, x) = \mathbb{V}[f(x)] = \sigma^2 = (\frac{5}{1.96})^2 = 6.508,$$

and lengthscale $\ell = 0.1$. We chose the variance to be 6.508 so that with (about) 95% probability the values of $f(x)$ are between -5 and 5.

[4]:
```
# Define the covariance function
k = GPy.kern.Exponential(1)
k.lengthscale = 0.1
k.variance = np.square(5.0 / 1.96)
```

```
# Sample
xs = np.linspace(0, 1, 100)
# The mean function at xs
ms = np.zeros(xs.shape)
# 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(dpi=100)
ax.plot(xs, f_samples.T, 'r', lw=0.5);
```



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 period should be:

$$\lambda = 0.1$$

The covariance function should be a standard periodic:

$$k(x, x') = s^2 \exp\left[-\frac{1}{2}\left(\frac{\sin(\frac{\pi}{\lambda}(x - x'))}{\ell}\right)^2\right],$$

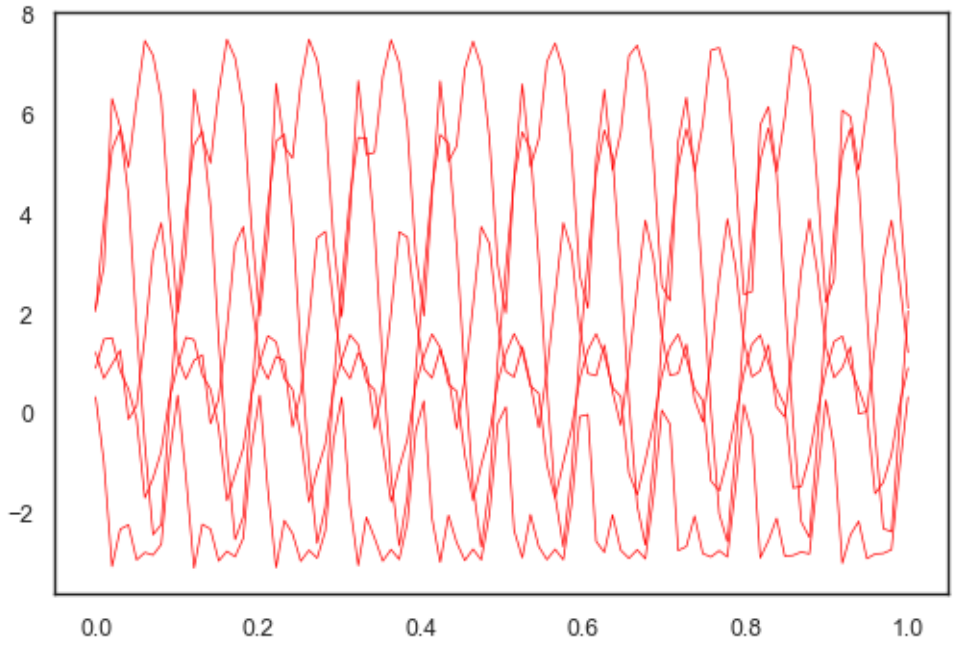
with variance:

$$1.96 * \sigma = 5$$

$$s^2 = k(x, x) = \mathbb{V}[f(x)] = \sigma^2 = (\frac{5}{1.96})^2 = 6.508,$$

and lengthscale $\ell = 0.5$. We chose the variance to be 6.508 so that with (about) 95% probability the values of $f(x)$ are between -5 and 5.

[5]:
```
# Define the covariance function
k = GPy.kern.StdPeriodic(1)
k.lengthscale = 0.5
k.variance = np.square(5.0 / 1.96)
k.period = 0.1

# Sample
xs = np.linspace(0, 1, 100)
# The mean function at xs
ms = np.zeros(xs.shape)
# 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(dpi=100)
ax.plot(xs, f_samples.T, 'r', lw=0.5);
```



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 period should be:

$$\lambda = 0.1$$

The covariance function should be a standard periodic:

$$k(x, x') = s^2 \exp\left[-\frac{1}{2}\left(\frac{\sin(\frac{\pi}{\lambda}(x - x'))}{\ell}\right)^2\right],$$

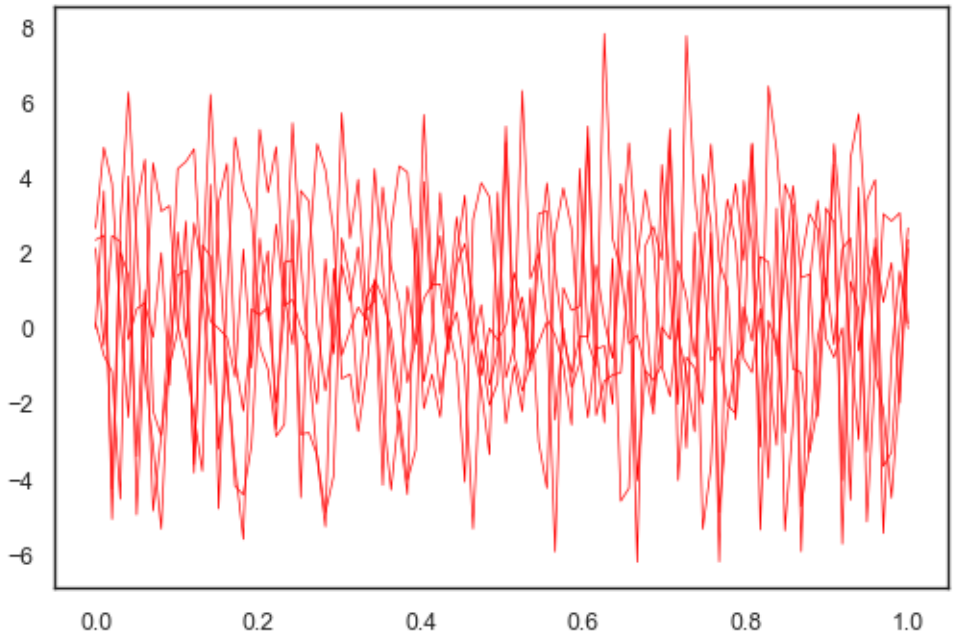
with variance:

$$1.96 * \sigma = 5$$

$$s^2 = k(x, x) = \mathbb{V}[f(x)] = \sigma^2 = (\frac{5}{1.96})^2 = 6.508,$$

and lengthscale $\ell = 0.1$. We chose the variance to be 6.508 so that with (about) 95% probability the values of $f(x)$ are between -5 and 5.

[6]:
```
# Define the covariance function
k = GPy.kern.StdPeriodic(1)
k.lengthscale = 0.1
k.variance = np.square(5.0 / 1.96)
k.period = 0.1

# Sample
xs = np.linspace(0, 1, 100)
# The mean function at xs
ms = np.zeros(xs.shape)
# 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(dpi=100)
ax.plot(xs, f_samples.T, 'r', lw=0.5);
```



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...

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:

$$1.96 * \sigma = 2$$

$$s^2 = k(x, x) = \mathbb{V}[f(x)] = \sigma^2 = (\frac{2}{1.96})^2 = 1.041,$$

and lengthscale $\ell = 0.5$. We chose the variance to be 1.041 so that with (about) 95% probability the values of $f(x)$ are between -2 and 2.

For the second covariance function...

The mean function should be:

$$m(x) = 0.$$

The covariance function should be an exponential:

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

with variance:

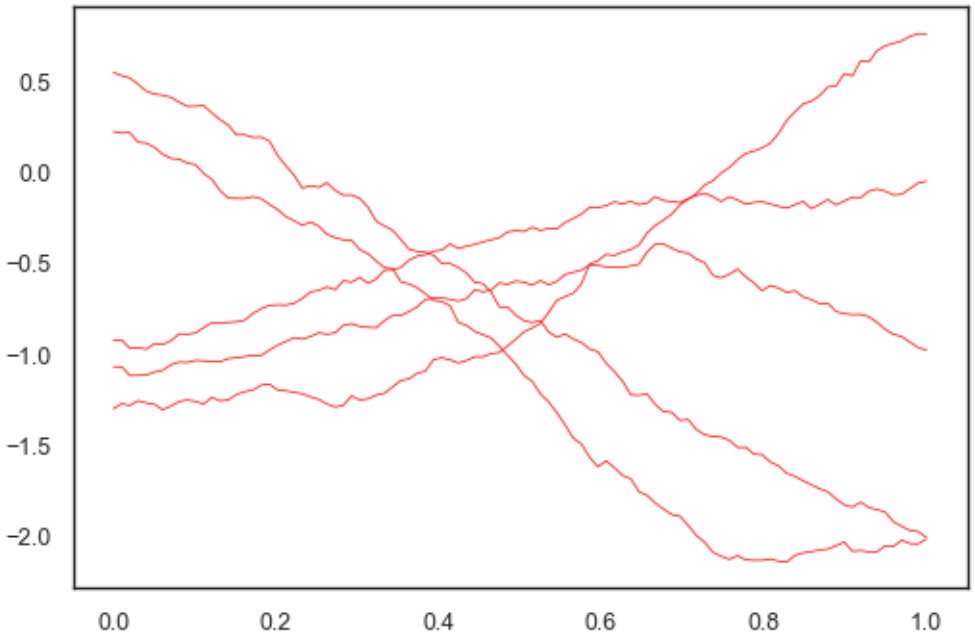
$$1.96 * \sigma = 0.1$$

$$s^2 = k(x, x) = \mathbb{V}[f(x)] = \sigma^2 = (\frac{0.1}{1.96})^2 = 0.0026,$$

and lengthscale $\ell = 0.1$. We chose the variance to be 0.0026 so that with (about) 95% probability the values of $f(x)$ are between -0.1 and 0.1.

[7]:
```
# Define the first covariance function
k1 = GPy.kern.RBF(1)
k1.lengthscale = 0.5
k1.variance = np.square(2.0 / 1.96)

# Define the second covariance function
k2 = GPy.kern.Exponential(1)
k2.lengthscale = 0.1
k2.variance = np.square(0.1 / 1.96)

# Create a new covariance function that is the sum of these two:
k_new = k1 + k2
```

```
# Sample
xs = np.linspace(0, 1, 100)
# The mean function at xs
ms = np.zeros(xs.shape)
# Find the covariance matrix. You need to add a small number
# to the diagonal to ensure numerical stability
nugget = 1e-6
K = k_new.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(dpi=100)
ax.plot(xs, f_samples.T, 'r', lw=0.5);
```



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), 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...

The mean function should be:

$$m(x) = 0.$$

The period should be:

$$\lambda = 0.1$$

The covariance function should be a standard periodic:

$$k(x, x') = s^2 \exp\left[-\frac{1}{2}\left(\frac{\sin(\frac{\pi}{\lambda}(x - x'))}{\ell}\right)^2\right],$$

with variance:

$$1.96 * \sigma = 2$$

$$s^2 = k(x, x) = \mathbb{V}[f(x)] = \sigma^2 = (\frac{2}{1.96})^2 = 1.041,$$

and lengthscale $\ell = 0.1$. We chose the variance to be 1.041 so that with (about) 95% probability the values of $f(x)$ are between -2 and 2.

For the second covariance function...

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:

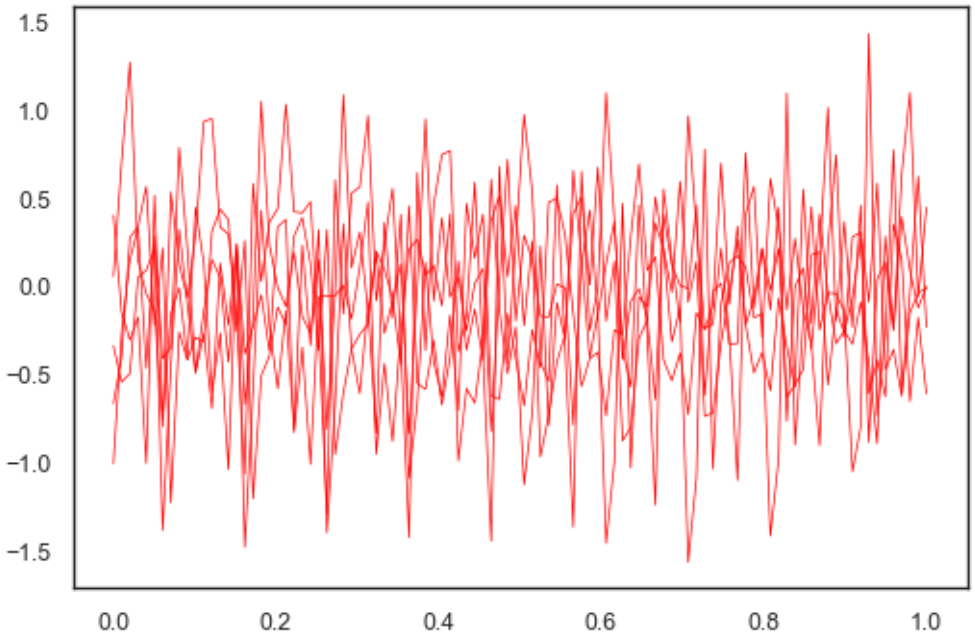
$$1.96 * \sigma = 1$$

$$s^2 = k(x, x) = \mathbb{V}[f(x)] = \sigma^2 = (\frac{1}{1.96})^2 = 0.026,$$

and lengthscale $\ell = 0.5$. We chose the variance to be 0.026 so that with (about) 95% probability the values of $f(x)$ are between -1 and 1.

[8]:
```
# Define the first covariance function
k1 = GPy.kern.StdPeriodic(1)
k1.lengthscale = 0.1
k1.variance = np.square(2.0 / 1.96)
k1.period = 0.1

# Define the second covariance function
k2 = GPy.kern.RBF(1)
k2.lengthscale = 0.5
k2.variance = np.square(1.0 / 1.96)

# Create a new covariance function that is the sum of these two:
k_new = k1 * k2

# Sample
xs = np.linspace(0, 1, 100)
```

```
# The mean function at xs
ms = np.zeros(xs.shape)
# Find the covariance matrix. You need to add a small number
# to the diagonal to ensure numerical stability
nugget = 1e-6
K = k_new.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(dpi=100)
ax.plot(xs, f_samples.T, 'r', lw=0.5);
```



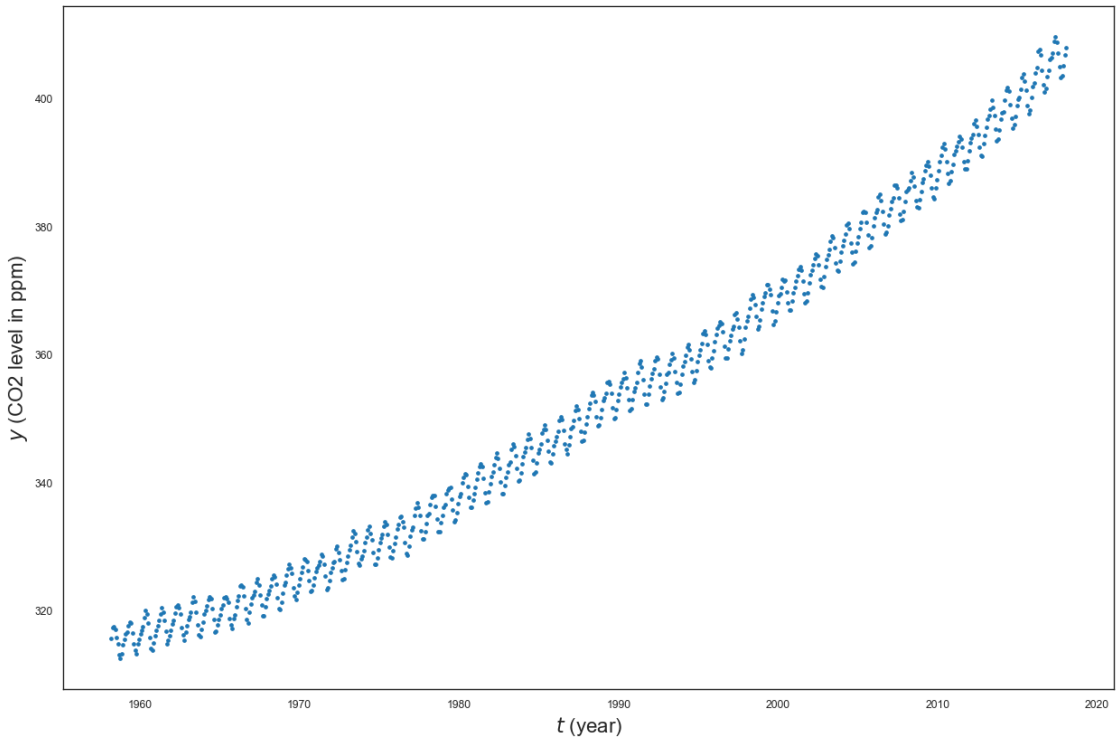
1.5 Problem 2

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.

[9]:
```
# download data
url = 'https://raw.githubusercontent.com/PredictiveScienceLab/data-analytics-se/
 ↪master/homework/mauna_loa_co2.txt'
download(url)
```

[10]:
```
data = np.loadtxt('mauna_loa_co2.txt')
```

[11]:
```
#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 = (15, 10), dpi=100)
ax.plot(t, y, '.')
ax.set_xlabel('$t$ (year)', fontsize = 16)
ax.set_ylabel('$y$ (CO2 level in ppm)', fontsize = 16);
```



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:

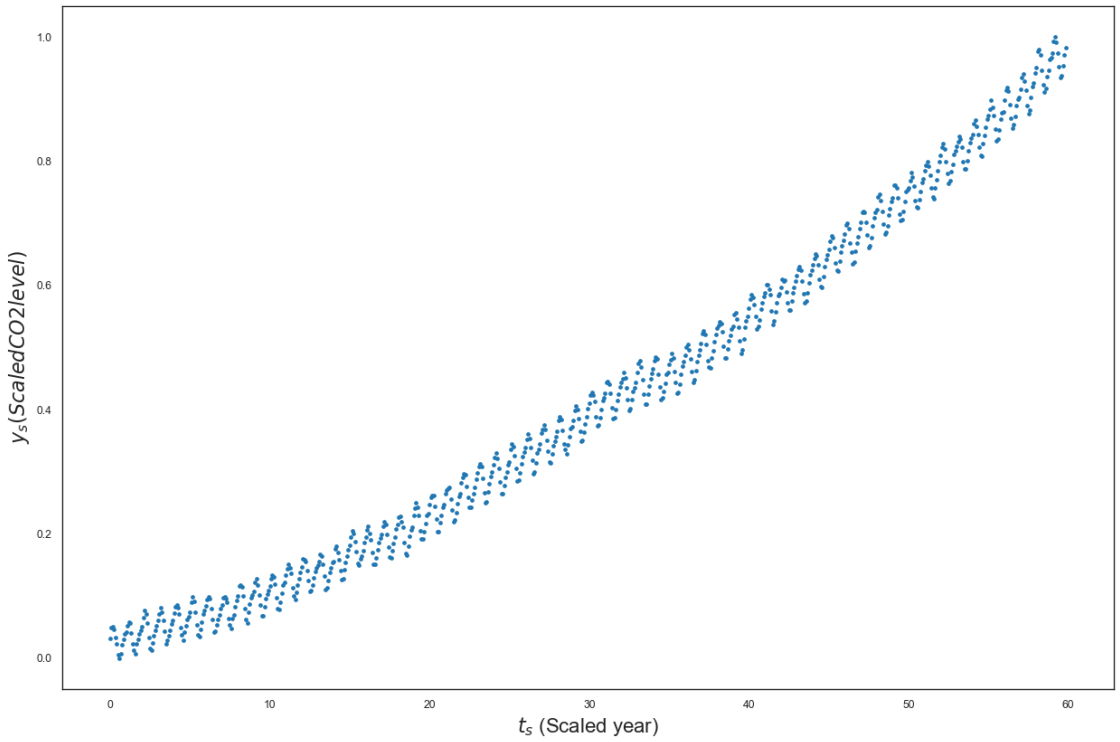
$$t_s = t - t_{\min}.$$

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 look like:

[12]:
```
t_s = t - t.min()
y_s = (y - y.min()) / (y.max() - y.min())
fig, ax = plt.subplots(1, figsize = (15, 10), dpi=100)
ax.plot(t_s, y_s, '.')
ax.set_xlabel('$t_s$ (Scaled year)', fontsize = 16)
ax.set_ylabel('$y_s (Scaled CO2 level)$', fontsize = 16);
```



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:

[13]:
```
# naive model
naive_kernel = GPy.kern.RBF(1)
naive_model = GPy.models.GPRegression(t_s, y_s, naive_kernel)
naive_model.optimize(messages=True)
```

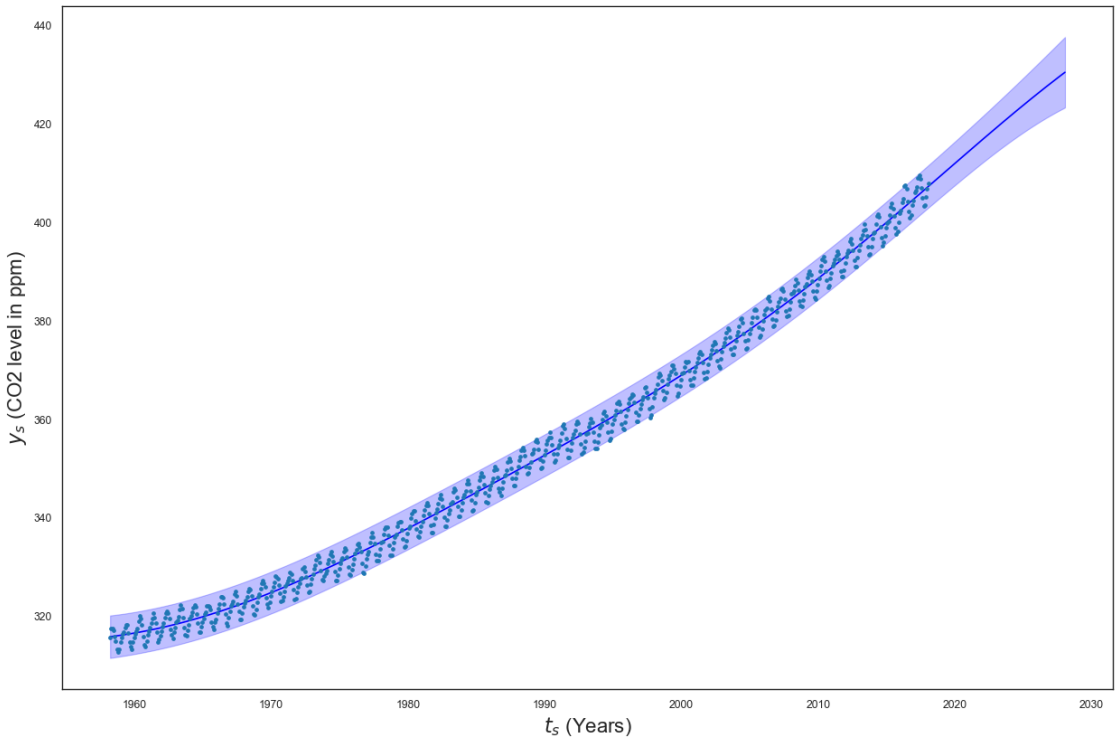
```
HBox(children=(VBox(children=(IntProgress(value=0, max=1000), HTML(value=''))),␣
↪Box(children=(HTML(value=''),)…
```

[13]:
```
<paramz.optimization.optimization.opt_lbfgsb at 0x7fa3971af130>
```

Predict everything:

[14]:
```
tss = np.linspace(0, t_s.max() + 10, 200)[:, None]
ys, vs = naive_model.predict(tss)
ls = ys - 1.96 * np.sqrt(vs)
us = ys + 1.96 * np.sqrt(vs)
```

[15]:
```
# reset original values
tss = np.linspace(0, t_s.max() + 10, 200)[:, None]
t = data[:, 2][:, None]
y = data[:, 4][:, None]

# for transforming back to original y scale
def inverse_scale(f_samples_singular):
    return (f_samples_singular * (y.max() - y.min())) + y.min()

# Plot the samples (in different colors)
tss = tss + t.min()

fig, ax = plt.subplots(1, figsize = (15, 10), dpi=100)
ax.plot(tss, inverse_scale(ys), color='blue', label='Posterior mean')
ax.fill_between(tss.flatten(), inverse_scale(ls.flatten()), inverse_scale(us.
 ↪flatten()), color='blue', alpha=0.25)
ax.plot(t, y, '.', label='Observed data')
ax.set_xlabel('$t_s$ (Years)', fontsize = 16)
ax.set_ylabel('$y_s$ (CO2 level in ppm)', fontsize = 16);
```



Notice that the squared exponential covariance captures the long terms, but it fails to capture the seasonal fluctuations. As a matter of fact the seasonal fluctions are 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.

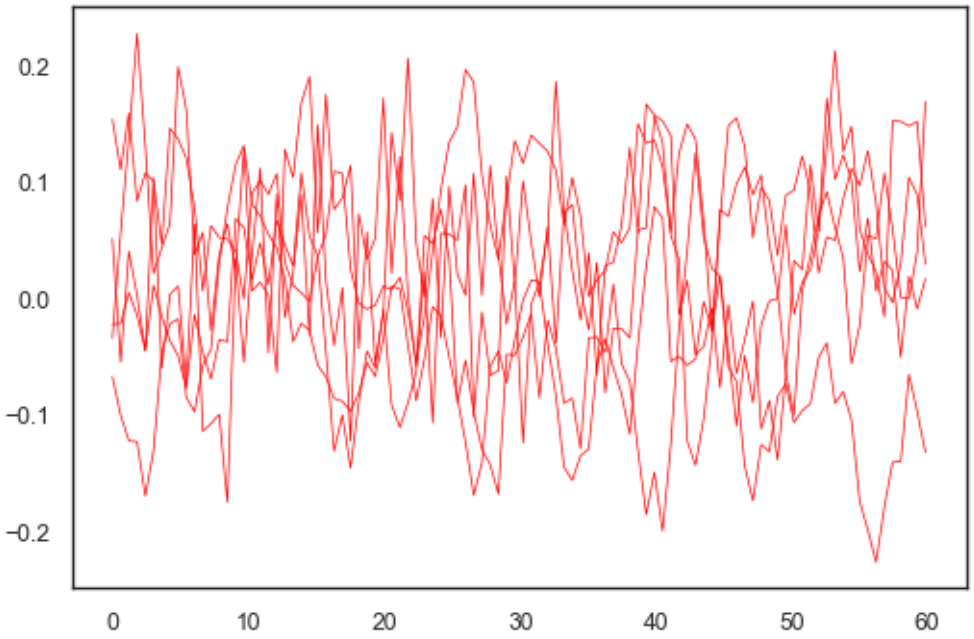
Hint: Do not attempt to fit the data in any way. Just try to find a covariance function that has the right features. We also do not care about getting the parameters 100% right at this point. The parameters will be optimized later.

Answer:

[16]:
```
# Define the first covariance function
k1 = GPy.kern.StdPeriodic(1)
k1.lengthscale = 1.0
k1.variance = np.square(0.5 / 1.96)
```

```
k1.period = 1.0

# Define the second covariance function
k2 = GPy.kern.RBF(1)
k2.lengthscale = 20.0
k2.variance = np.square(0.5 / 1.96)

# Define the third covariance function
k3 = GPy.kern.RBF(1)
k3.lengthscale = 1.0
k3.variance = np.square(0.5 / 1.96)

# Create a new covariance function that is the sum of these two:
improved_kernel = (k1 + k3) * k2
# improved_kernel = k1

# confirm the new kernel is positive semi-definite (valid covariance function)
eigenvalues = np.absolute(np.linalg.eigvals(k1.K(t_s) + k2.K(t_s)))
if len(eigenvalues[eigenvalues < 0]) == 0:
    print('No negative eigenvalues --> covariance matrix is valid!')

# Sample
xs = np.linspace(0, t_s.max(), 100)
# The mean function at xs
ms = np.zeros(xs.shape)
# Find the covariance matrix. You need to add a small number
# to the diagonal to ensure numerical stability
nugget = 1e-6
K = improved_kernel.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(dpi=100)
ax.plot(xs, f_samples.T, 'r', lw=0.5);
```

```
No negative eigenvalues --> covariance matrix is valid!
```



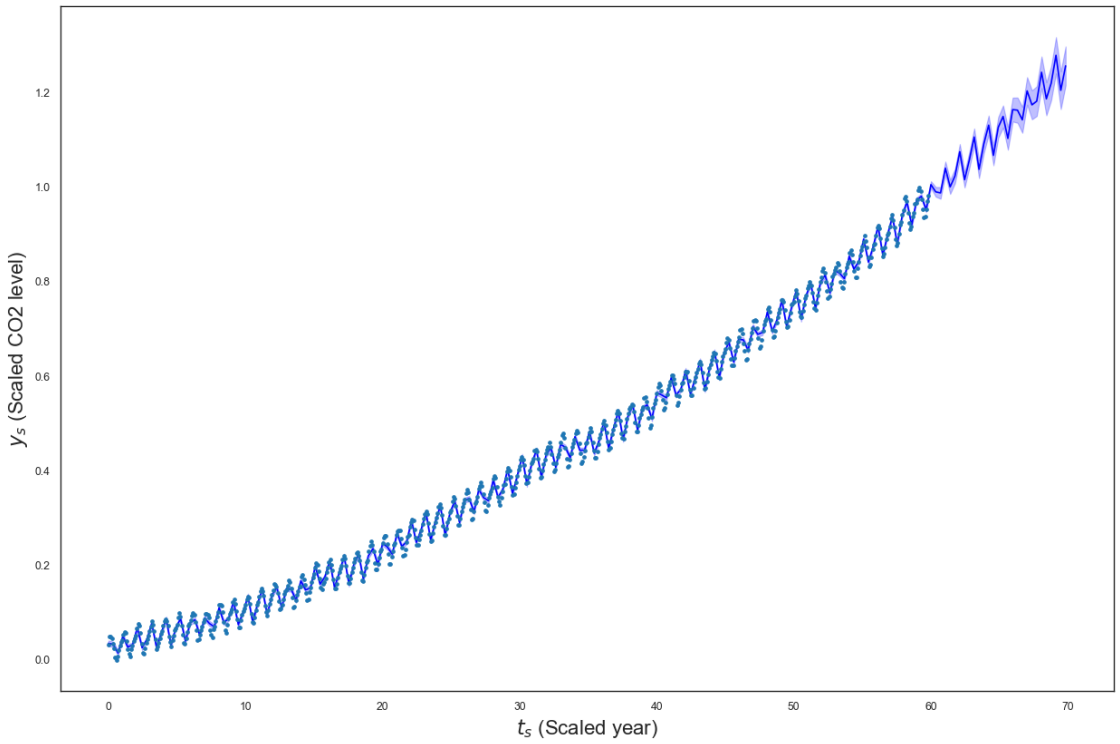
1.8 Part C - Predicting the future

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:

[17]:
```
# improved model
improved_model = GPy.models.GPRegression(t_s, y_s, improved_kernel)
improved_model.optimize(messages=True)

tss = np.linspace(0, t_s.max() + 10, 200)[:, 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 = (15, 10), dpi=100)
ax.plot(tss, ys, color='blue', label='Posterior mean')
ax.fill_between(tss.flatten(), ls.flatten(), us.flatten(), color='blue',␣
↪alpha=0.25)
ax.plot(t_s, y_s, '.', label='Scaled observed data')
ax.set_xlabel('$t_s$ (Scaled year)', fontsize = 16)
ax.set_ylabel('$y_s$ (Scaled CO2 level)', fontsize = 16);
```

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



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.

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:

[18]:
```
def computeBIC(num_params, log_like, n):
    return num_params*np.log(n) - 2*log_like

n = len(t_s)

naive_num_params = naive_model.num_params
naive_log_like = naive_model.log_likelihood()
```

```
bic_naive = computeBIC(naive_num_params, naive_log_like, n)

improved_num_params = improved_model.num_params # improved_model.
improved_log_like = improved_model.log_likelihood()
bic_improved = computeBIC(improved_num_params, improved_log_like, n)

print('Naive model\'s BIC    : {}'.format(bic_naive))
print('Improved model\'s BIC : {}'.format(bic_improved))
print('\nWe should use the improved model\'s BIC, because it is smaller.')
```

```
Naive model's BIC    : -3369.7856313015495
Improved model's BIC : -6079.668108674042

We should use the improved model's BIC, because it is smaller.
```

1.10 Part E - Plot samples from the posterior Gaussian process

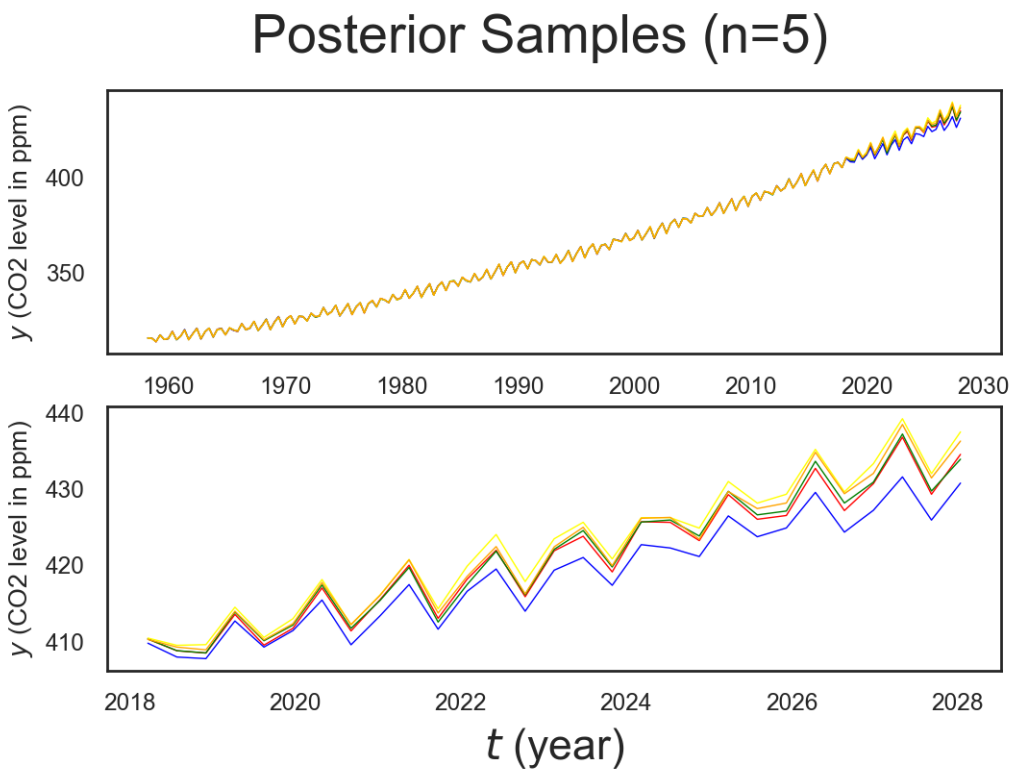
Using the model of Part C, plot 5 samples from the posterior Gaussian process between 2018 and 2028.

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

Answer:

[19]:
```
# reset original values
t = data[:, 2][:, None]
y = data[:, 4][:, None]

# for transforming back to original y scale
def inverse_scale(f_samples_singular):
    return (f_samples_singular * (y.max() - y.min())) + y.min()

# Take the function samples
xs = np.linspace(0, t_s.max()+10, 200)
f_samples = improved_model.posterior_samples_f(xs[:,None], size=5)
f_samples = f_samples.T
f_samples = f_samples.reshape(5,200)

# Plot the samples (in different colors)
xs = xs + t.min()
to_keep = len(xs[xs < 2018])

# entire posterior sample
fig, axs = plt.subplots(2, 1, dpi=200)
axs[0].plot(xs, inverse_scale(f_samples[0, :]), 'red', lw=0.5);
axs[0].plot(xs, inverse_scale(f_samples[1, :]), 'blue', lw=0.5);
axs[0].plot(xs, inverse_scale(f_samples[2, :]), 'green', lw=0.5);
axs[0].plot(xs, inverse_scale(f_samples[3, :]), 'yellow', lw=0.5);
axs[0].plot(xs, inverse_scale(f_samples[4, :]), 'orange', lw=0.5);
```

```
# zoom in on 10 year prediction
axs[1].plot(xs[to_keep:], inverse_scale(f_samples[0, to_keep:]), 'red', lw=0.5);
axs[1].plot(xs[to_keep:], inverse_scale(f_samples[1, to_keep:]), 'blue', lw=0.
 ↪5);
axs[1].plot(xs[to_keep:], inverse_scale(f_samples[2, to_keep:]), 'green', lw=0.
 ↪5);
axs[1].plot(xs[to_keep:], inverse_scale(f_samples[3, to_keep:]), 'yellow', lw=0.
 ↪5);
axs[1].plot(xs[to_keep:], inverse_scale(f_samples[4, to_keep:]), 'orange', lw=0.
 ↪5);

# make pretty
fig.suptitle('Posterior Samples (n=5)', fontsize=20)
axs[1].set_xlabel('$t$ (year)', fontsize = 16)
axs[0].set_ylabel('$y$ (CO2 level in ppm)');
axs[1].set_ylabel('$y$ (CO2 level in ppm)');
```



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:

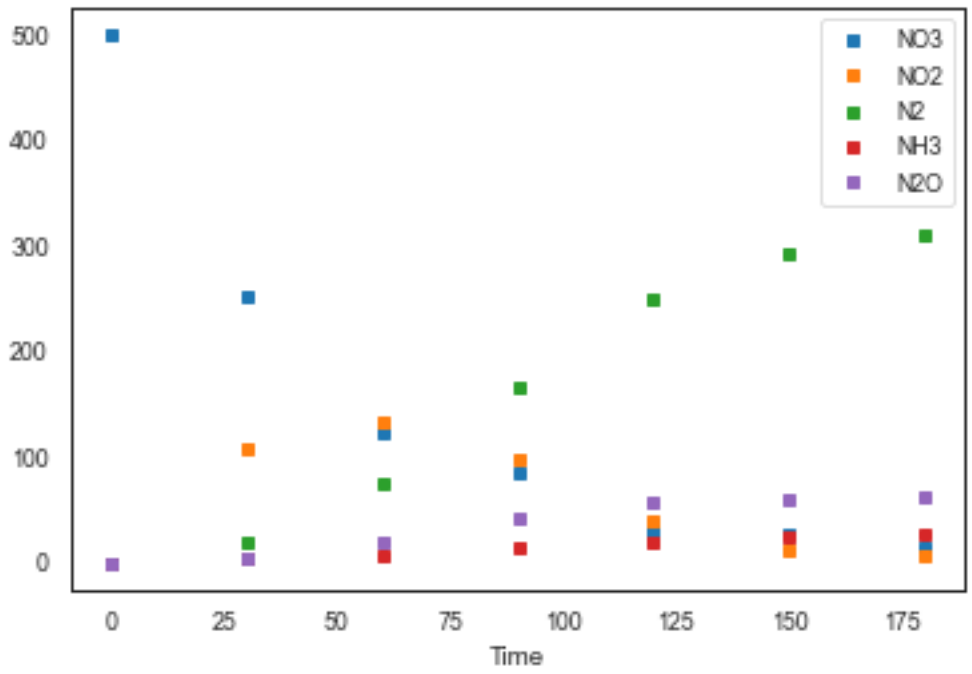
[20]:
```
# Download the file
url = 'https://raw.githubusercontent.com/PredictiveScienceLab/data-analytics-se/
 ↪master/homework/catalysis.csv'
# download(url)
```

[21]:
```
# Load the data
import pandas as pd
catalysis_data = pd.read_csv('catalysis.csv')
catalysis_data
```

[21]:

	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

[22]:
```
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:

[23]:
```
catalysis_data.sum(axis=1)
```

[23]:
```
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*:

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

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:

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

the vector:

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

and the matrix:

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

With these definitions, the dynamical system becomes:

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

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:

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

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 proparly 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:

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

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:

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

$j = 1, \ldots, 6$.

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

[24]:
```
catalysis_data[1:]
```

[24]:

	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:

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

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:

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

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

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:

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

+ And flatten that function:

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

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

[25]:
```
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:

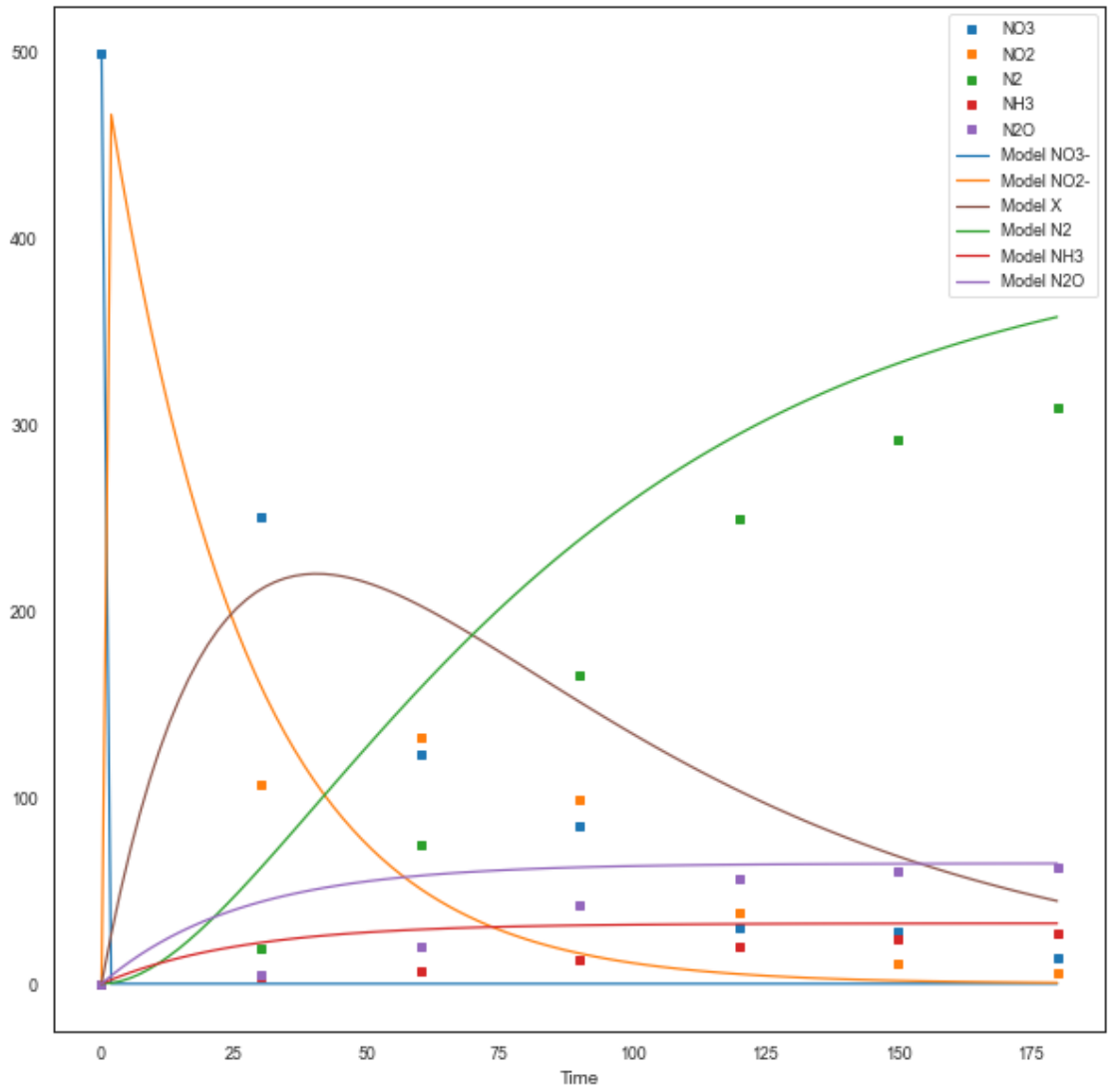
[26]:
```
import scipy.optimize

# 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: 0.20861275269214927
hess_inv: array([[ 3.03342126e-05,  8.19445261e-05,  1.51890059e-02,
         1.79633545e-05, -1.64382310e-04],
       [ 8.19445261e-05,  2.79463008e-04,  5.11119850e-02,
         7.83565525e-05, -5.59632244e-04],
       [ 1.51890059e-02,  5.11119850e-02,  1.25953973e+01,
         1.43516996e-02, -1.02467890e-01],
```

```
       [ 1.79633545e-05,  7.83565525e-05,  1.43516996e-02,
         2.62327578e-05, -1.57166384e-04],
       [-1.64382310e-04, -5.59632244e-04, -1.02467890e-01,
        -1.57166384e-04,  1.12163947e-03]])
     jac: array([3.68407313e-02, 2.47284733e-02, 2.45701522e-05,
3.25641204e-02,
       7.83492811e-03])
  message: 'Desired error not necessarily achieved due to precision loss.'
     nfev: 546
      nit: 28
     njev: 89
   status: 2
  success: False
        x: array([ 9.94193122,  1.70744145,  0.98875077, -0.81164003,
-0.12236107])
```

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

[27]:
```
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:

[28]:
```
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:

[29]:
```
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:

[30]:
```
# 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:

[31]:
```
ARD_kernel = GPy.kern.RBF(5, ARD = True)
ARD_model = GPy.models.GPRegression(X_init_train, Y_init_train, ARD_kernel)
ARD_model.likelihood.variance.constrain_fixed(1e-6)

print(ARD_model.kern.lengthscale)
```

```
  index  |  GP_regression.rbf.lengthscale  |  constraints  |  priors
  [0]    |                      1.00000000  |      +ve      |
  [1]    |                      1.00000000  |      +ve      |
  [2]    |                      1.00000000  |      +ve      |
  [3]    |                      1.00000000  |      +ve      |
  [4]    |                      1.00000000  |      +ve      |
```

1.13 Part B - Inspecting your model

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

Answer:

[32]:
```
ARD_model.optimize(messages=True)
lengthscales = ARD_model.kern.lengthscale
lengthscale_list = [(i, x) for i, x in enumerate(lengthscales)]
lengthscale_list = sorted(lengthscale_list, key=lambda tup: tup[1])

print('Rank | Parameter Number | Lengthscale')
for k, item in enumerate(lengthscale_list):
    print('{}    | {}                | {}'.format(k + 1,␣
 ↪lengthscale_list[k][0], lengthscale_list[k][1]))
```

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

```
Rank | Parameter Number | Lengthscale
1    | 0                | 2.031929924235957
2    | 3                | 3.324340510869403
3    | 4                | 3.532729266640867
4    | 1                | 3.679778729944968
5    | 2                | 14.552755498446139
```

1.14 Part C - Diagnostics

Here are some test data:

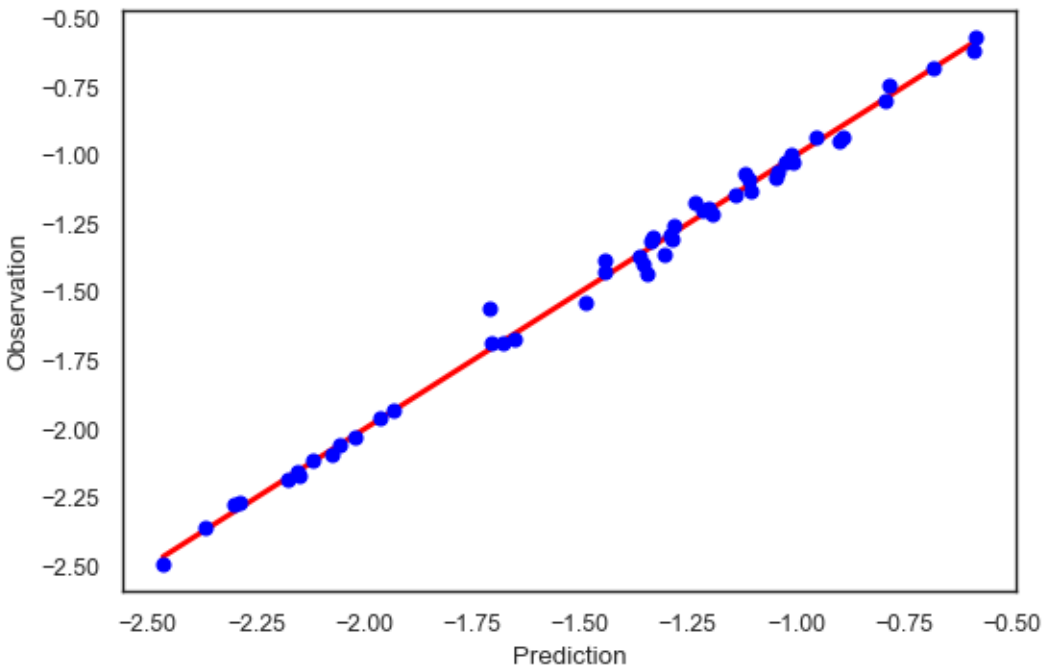
[33]:
```
# Test points
n_test = 50
X_test = -2.0 + 4.0 * np.random.rand(n_test, 5)
Y_test = np.array([h(x) for x in X_test])[:, None]
```

Do the following:

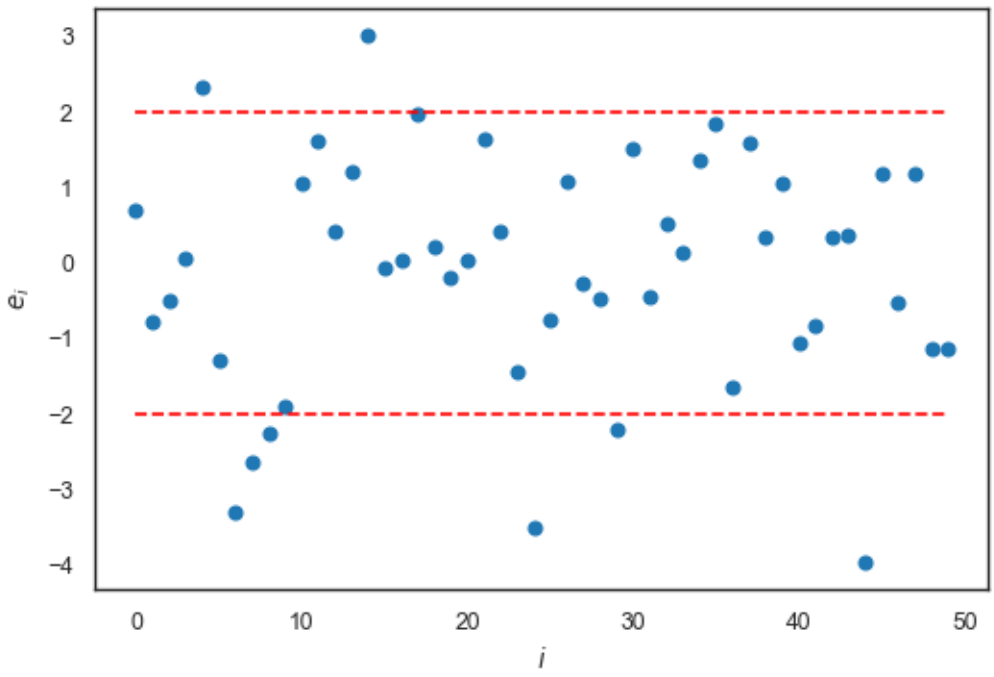
- Predictions vs observations plot
- Standarized errors plot

Answer:

[34]:
```
Y_pred_m, Y_pred_v = ARD_model.predict(X_test)

fig, ax = plt.subplots(dpi=100)
y_range = np.linspace(Y_test.min(), Y_test.max(), 50)
ax.plot(y_range, y_range, 'r', lw=2)
ax.plot(Y_test, Y_pred_m, 'bo')
ax.set_xlabel('Prediction')
ax.set_ylabel('Observation');
```



[35]:
```
Y_pred_sd = np.sqrt(Y_pred_v)
e = (Y_test - Y_pred_m) / Y_pred_sd

fig, ax = plt.subplots(dpi=100)
ax.plot(e, 'o')
ax.plot(np.arange(e.shape[0]), 2.0 * np.ones(e.shape[0]), 'r--')
ax.plot(np.arange(e.shape[0]), -2.0 * np.ones(e.shape[0]), 'r--')
ax.set_xlabel('$i$')
ax.set_ylabel('$e_i$');
```



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.

[36]:
```
# 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, ARD_model, domain)
```

```
Iteration       Current best objective  Current acquisition func. value
1               -0.17                   0.24
2               -0.17                   0.02
3               -0.17                   0.01
4               -0.17                   0.01
5               -0.17                   0.01
6               -0.17                   0.01
```

Use this code to plot your calibrated model:

[37]:
```
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();
```