ENGSCI 233

Sampled Data Lab

Date: 18 March 2022

Lab Background and Objectives

Experimental or field **data** provides an essential window into the physical world, and a measuring stick against which to calibrate[1] computer models. Unfortunately, real data are often **incomplete** or only **sporadically sampled**, in which case it is useful to be able to "fill in the gaps". In this lab, we are using interpolation to estimate the missing values. However, from the perspective of machine learning[2], this corresponds to the prediction of so-called out-of-sample values.

Your objective in this lab is to implement **cubic spline interpolation**, a method for fitting cubic polynomials to data that ensures continuity of the first and second derivative of the interpolating function. You will then use your code to **analyse** some Oil & Gas production, injection and earthquake data and assess operator culpability. Finally, we will take this opportunity to become familiar with Python's built-in **plotting** functions.

In order to complete the lab, you have been provided with the following files:

- cubic spline function templates in `sdlab_functions.py`
- a series of practice exercises and implementation tests in `sdlab_practice.py`
- a file containing instructions (and space to code) for the assessed task in `sdlab_earthquakes.py`
- a question-answer template file `sdlab_questions.txt`
- the data files `IW1.dat`, `PW1.dat`, and `PW2.dat`

Use these to complete the practice exercises, implement your cubic spline interpolation algorithm, and then perform the required analysis.

[1] If you're Engineering Science, there's more on that in ENGSCI263.
[2] Machine learning will be introduced in semester 2 in ENGSCI205

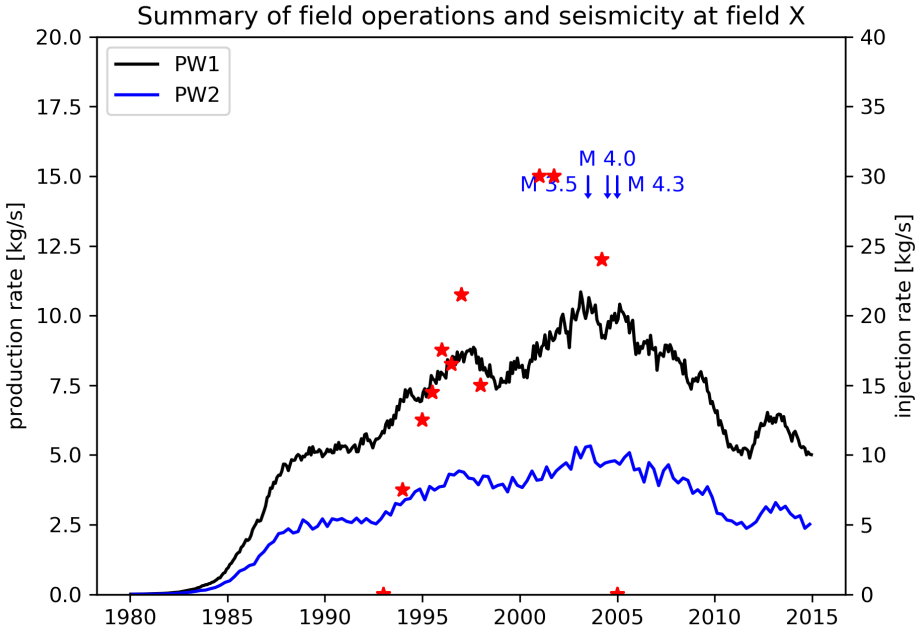
Practice Exercises

Some of these exercises are OPTIONAL, others are REQUIRED.

In your IDE, open `sdlab_practice.py` and `sdlab_functions.py` and read the commented instructions. You should pass all the asserts in `sdlab_practice.py` before attempting the assessed task.

1. Plotting data (max 20 mins)

Enable and run the first set of plotting commands by setting `run_ex1 = True`. A figure window will automatically appear when the code executes `plt.show()`.



Unlike MATLAB, Python code pauses in execution when a plot window is generated, and does not continue until you close it.

Step through and attach a brief comment (What are the command inputs? How does it alter the figure?) for the following commands.

- `plt.subplots(...)`
- `ax1.plot(...)`[3]

[3]See the Notes section on string formatting here.

- `ax1.arrow(...)`
- `ax1.text(...)`
- `ax1.set_ylim(...)`
- `ax1.legend(...)`
- `ax1.set_ylabel(...)`
- `plt.show()`
- `plt.save_fig(...)`

Practice saving a figure to the disk by changing the value of the boolean variable `save_figure`. Verify that the newly created file, `sdlab_plot.png`, corresponds to the same figure.

Have a go at:

1. Changing the colour of one of the lines.
2. Changing the type of marker used for the injection.
3. Changing a legend or axis label.
4. Changing one of the earthquake text labels.

When you are satisfied that you understand the plotting commands (you will use these again later), disable this exercise by setting `run_ex1 = False`.

2. Cubic Spline Interpolation

Review Section 1.3 of the `sampled_data.ipynb` notebook. We wish to set up and solve the matrix equation $A\,\mathbf{x} = \mathbf{b}$, where $\mathbf{x}$ is a vector of spline polynomial coefficients. For example, for three data points, and two subintervals, the matrix equation

looks like

$$A\mathbf{x} = \mathbf{b} \quad \Rightarrow \quad \begin{bmatrix} . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \end{bmatrix} \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ a_2^{(0)} \\ a_3^{(0)} \\ a_0^{(1)} \\ a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \end{bmatrix} = \begin{bmatrix} . \\ . \\ . \\ . \\ . \\ . \\ . \\ . \end{bmatrix}$$

where A is an 8×8 coefficient matrix and $\mathbf{b}$ is an 8×1 RHS vector. Entries of A and $\mathbf{b}$ are populated using the equations derived in Section 1.3.1: (i) that the polynomial for each subinterval **must pass through** its bounding points; (ii) that the first and second derivatives **must be continuous** where two subintervals join; and (iii) two additional boundary conditions.

For this task, you should complete the templated implementation of cubic spline interpolation in `sdlab_functions.py`, specifically the functions `spline_coefficient_matrix()`, `spline_rhs()`, and `spline_interpolate()`. A series of `assert` statements have been included in `sdlab_practice.py` to help you check these functions are working correctly. In order to run these commands, set `run_ex2 = True`.

Part i. `spline_coefficient_matrix(...)`

The purpose of this function is to construct and populate the matrix A. Start by considering the spline example from class: three data points at `(1,2)`, `(2,5)`, and `(3,4)`, hence two subintervals (and two cubic polynomials), and eight unknown polynomial coefficients.

Consider the **first subinterval** and the requirement that its cubic polynomial must pass through the **lefthand point** of the subinterval.

$$p_0(x) = a_0^{(0)} + a_1^{(0)}(x - x_0) + a_2^{(0)}(x - x_0)^2 + a_3^{(0)}(x - x_0)^3,$$

which when substituting $x = x_0 = 1$ and $y = y_0 = 2$ reduces to

$$2 = a_0^{(0)} + a_1^{(0)} \times 0 + a_2^{(0)} \times 0 + a_3^{(0)} \times 0.$$

We can rewrite the equation to explicitly include all the unknowns

$$2 = 1 \times a_0^{(0)} + 0 \times a_1^{(0)} + 0 \times a_2^{(0)} + 0 \times a_3^{(0)} + 0 \times a_0^{(1)} + 0 \times a_1^{(1)} + 0 \times a_2^{(1)} + 0 \times a_3^{(1)},$$

which as a row of the matrix is

$$A\mathbf{x} = \mathbf{b} \quad \Rightarrow \quad \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \end{bmatrix} \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ a_2^{(0)} \\ a_3^{(0)} \\ a_0^{(1)} \\ a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \end{bmatrix} = \begin{bmatrix} 2 \\ . \\ . \\ . \\ . \\ . \\ . \\ . \\ . \end{bmatrix}$$

Derive the next equation[4] and enter it in the matrix.

STOP. Think numerically about the algorithm you just used.

- How could you **generalise** the procedure FOR N subintervals?
- How will you use **indices** to ensure that coefficients are placed in the **correct rows and columns** of the matrix?
- How much of A will have been populated once you have considered all subintervals?

Write commands in `spline_coefficient_matrix` ***to populate the matrix*** A ***in a way that reflects these equations.***

Splines must also have a continuous first derivative where two subintervals meet each other. Differentiating the equations of our two neighbouring subintervals, we have

$$\begin{aligned} \frac{d}{dx}p_0(x) &= a_1^{(0)} + 2a_2^{(0)}(x - x_0) + 3a_3^{(0)}(x - x_0)^2, \\ \frac{d}{dx}p_1(x) &= a_1^{(1)} + 2a_2^{(1)}(x - x_1) + 3a_3^{(1)}(x - x_1)^2. \end{aligned}$$

[4]Spline polynomial for first subinterval passes through **righthand point**.

The two derivatives must be equal at the shared point, $x = x_1$, i.e.

$$\frac{d}{dx}p_0(x_1) = \frac{d}{dx}p_1(x_1) \quad \Rightarrow \quad a_1^{(0)} + 2a_2^{(0)}(x_1 - x_0) + 3a_3^{(0)}(x_1 - x_0)^2 = a_1^{(1)}.$$

Rearranging this equation, substituting $x_0 = 1$ and $x_1 = 2$, and making explicit all the unknown spline coefficients

$$0 \times a_0^{(0)} + 1 \times a_1^{(0)} + 2 \times a_2^{(0)} + 3 \times a_3^{(0)} + 0 \times a_0^{(1)} - 1 \times a_1^{(1)} + 0 \times a_2^{(1)} + 0 \times a_3^{(1)} = 0,$$

or in matrix form

$$A\mathbf{x} = \mathbf{b} \quad \Rightarrow \quad \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ 0 & 1 & 2 & 3 & 0 & -1 & 0 & 0 \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \end{bmatrix} \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ a_2^{(0)} \\ a_3^{(0)} \\ a_0^{(1)} \\ a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \end{bmatrix} = \begin{bmatrix} 2 \\ . \\ . \\ . \\ 0 \\ . \\ . \\ . \\ . \end{bmatrix}$$

Derive the next equation[5] and enter it in the matrix.

STOP. Think numerically about the algorithm you just used.

- **How many** equations should there be for $(n - 2)$ subintervals? Is a loop required?
- How much of A will have been populated once you have considered all subintervals?

Write commands in `spline_coefficient_matrix` ***to populate the matrix*** A ***in a way the reflects the derivative continuity equations.***

Finally, you need to write two equations for what happens at the boundaries of the data, x_0 and x_{n-1}. In this lab, we shall use the **natural spline** condition, which assumes that the **second derivative** at these points is zero.

Write commands in `spline_coefficient_matrix` ***to implement the natural spline boundary condition.***

[5] Second derivative continuous where subintervals meet.

You will know your code is **working correctly** if you are able to pass the assert command in sdlab_practice.py.

```
assert norm(A - A_soln) < tol
```

Pay particular attention to the **assumed order of the equations** in A. Now onto the next function...

Part ii. `spline_rhs(...)`

The purpose of this function is to construct and populate the righthand side vector, $\mathbf{b}$. As with A, start by considering the example with three data points and two subintervals. Work through the **first** set of equations – polynomials honour the subinterval points – then the **second** set of derivative constraints, and **finally** the boundary conditions.

Write commands in `spline_rhs` ***to construct the RHS vector*** $\mathbf{b}$***.***

You will know your code is **working correctly** if you are able to pass the assert command in sdlab_practice.py.

```
assert norm(b - b_soln) < tol
```

Pay particular attention to the **assumed order of the equations** in $\mathbf{b}$.

The matrix equation, $A\,\mathbf{x} = \mathbf{b}$, is solved for you using the `numpy.linalg.solve` function (which implements **LU factorisation with partial pivoting**, just like last week). This yields a vector of polynomial coefficients, $[a_0^{(0)}, a_1^{(0)}, ...a_3^{(n-2)}]$. The **first four** give the polynomial for the **first subinterval**, the second four for the second subinterval, etc.

Your final task is to perform the actual interpolation. Onto the next function...

Part iii. `spline_interpolate(...)`

The purpose of this function is to return a set of interpolated values, y_j, given a set of interpolating locations, x_j, subinterval boundaries, x_i, and polynomials, $a_i^{(k)}$.

You are free to determine a sensible method to do this, but the main requirement is that you pass each value x_j to the correct polynomial coefficients, $a_i^{(k)}$. The x_i are there to help you determine which is the correct polynomial.

Write commands in `spline_interpolate` ***to perform the interpolation.***

You may assume that the interpolation points, x_j, AND the subinterval boundaries, x_i, are given in ascending order.

You might also find it useful to use the `polyval(...)` function included in `sdlab_practice.py`.

You will know your code is **working correctly** if you are able to pass the assert command in `sdlab_practice.py`.

```
assert norm(yj - yj_soln) < tol
```

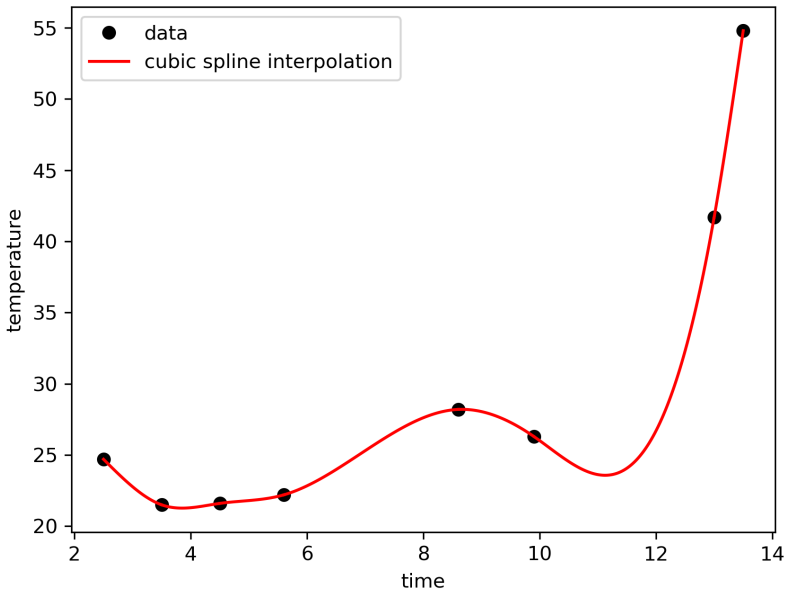
Before you move on to the analysis task...

One Final Test

As a last check to ensure that your cubic spline code is working correctly, set `run_ex3 = True` to run the final block of code in `sdlab_practice.py`.

These commands import the same temperature data used throughout `sampled_data.ipynb`, apply your interpolation algorithm, and plot the output.

You should generate a figure similar to that below.



Assessed Exercise

This exercise is REQUIRED.

Earthquakes are sometimes associated with oil and gas production (taking mass out of the ground) and injection (putting it back in) operations.

It has been suggested that injection of water at one particular site, which started midway through 1993, has been responsible for a spate of recent earthquakes there. The earthquake data are documented in lines 52–57 of `sdlab_practice.py`. The operator of the field has claimed they cannot be responsible, because injection had been ongoing for almost 10 years before any earthquakes occurred.

It has been proposed the earthquakes may be related to **net mass changes** in the field. Therefore, it is necessary to understand how this quantity has evolved over time.

Although data from the two production wells (mass extractors) - PW1 and PW2 - are reported regularly, data reporting from the injection well, IW1, is more irregular. In addition, the operator only reports **mass rate**, not **cumulative** production or injection **mass**.

Your goal is to produce a plot of NET MASS CHANGE as a function of time.

To achieve this, you will need to use both your **cubic spline interpolation** algorithm and **a Newton-Cotes integration scheme** of your choice (you may implement your own or use a Python built-in). You may wish to refer further to the hints in `sdlab_earthquakes.py`. You should write all your data processing and plotting commands in this file and then output `sdlab_earthquakes.png`. Submit both files.

On the basis of your plot, critically **assess** the operator's claim that they **cannot be responsible** for the triggered earthquakes.

Once you have completed your analysis, answer the questions in **`sdlab_questions.txt`*.***

Submission Instructions

For this lab, you should submit the Python files, `sdlab_functions.py` and `sdlab_earthquakes.py`, a png figure, `sdlab_earthquakes.png`, and the question-answer text file `sdlab_questions.txt` to Canvas. **DO NOT** modify or vary the names of these files (but don't worry if Canvas appends a `-1` or `-2` to the end). **DO NOT** submit any other files. **DO NOT** put your submission in a zip archive.

Remember, all submissions are compared against each other and those from previous years for similarity. Copying someone else's code and changing the variable names constitutes academic misconduct by both parties.