ProjectInstructions

September 21, 2023

1 Project guidelines

For the final assessment of our course, you must complete a small project in a group.

1.1 Choosing topics

Please choose at least **2 topics** you're interested in, and email me your choices before **Friday 6th October (evening)**. You can provide a ranking for your preferred topic choices. I will create groups of 3-4 based on topic choices over the weekend.

This is a list of proposed project topics:

- [A] Categorising data: supervised and unsupervised learning in Python
- [B] Finite difference method for the heat equation
- [C] Newton and Broyden's methods for nonlinear systems
- [D] Polynomial least squares
- [E] Loss of significance in floating-point arithmetic
- [F] The FEniCS and Dolfin libraries for numerical PDEs
- [G] Object-oriented Python for numerical ODEs

Each of these topics is introduced in more detail in the following sections. A project outline is given for each topic, including a description of the objectives and guidelines to get you started. The projects are designed to be open-ended – you can take them in any direction you like. Projects [A], [B], [C], [D] are more guided; projects [E], [F], [G] are more open-ended. I'll be happy to provide further guidance at any point if necessary.

For most of these topics, there are multiple possible directions of investigation. The choice of direction is yours – the important thing is that **you put your Python skills into practice**. Some of these directions may be, for example:

- implement an algorithm from scratch, test and investigate different use cases or parameters, try to optimise it as much as possible;
- implement two or more algorithms, investigate and compare their behaviour under similar conditions;
- use existing libraries (e.g. `scikit-learn` for machine learning, `FEniCS` for PDEs…), and focus on investigating, for instance, efficiency and/or accuracy;
- focus on a simple case, and produce interesting/insightful visualisations using Python tools;
- etc.

Alternatively, you could also propose your **own topic** – for instance revisit some of your past work, or a result or algorithm from the literature, and try reproducing it in Python. If you do so, then

you can still consult the example project briefs below to give you an idea of the scope and expected amount of work.

1.2 Support sessions

There will be 2 support sessions for the project:

- Tuesday 10th October, 11-12, 5.46 Bayes
- Tuesday 17th October, 11-12, 5.46 Bayes

Please attend these sessions with your group to make progress on your project, and ask for help/advice.

1.3 Assessment

The projects will be assessed on a pass/fail basis by a **15-min presentation** (+ Q&A), on **Tuesday 24th October, starting at 10am in 5.46 Bayes**.

Here is a suggested structure for your presentation:

- Introduce your project topic, and state the specific problem/question you have tackled. (2min)
- Explain the method you have used to solve the problem; it could be useful to illustrate it with a simple example and/or some useful visualisations produced in Python. (3-5min)
- Deep dive: choose a part of your code which you found interesting and/or challenging to implement, or maybe a part which taught you something new about Python, and walk the audience through your code to explain what you did. This doesn't necessarily have to be the "core" part of your code, or the part where the main computations happen – it could also be a utility function you wrote to visualise your results or clean up your data, for instance. (3-5min)
- Present your final results and conclude. (2-3min)

Here is what I recommend to prepare your presentation:

- The RISE extension allows you to (very easily!) create interactive slides from a Jupyter notebook. Have a look at the documentation. You'll need to install the extension on your computer with `conda install -c conda-forge rise`.
- Write the bulk of your code as functions in one or more separate modules (.py scripts) in the same folder as your notebook.
- Import your module(s) into the notebook to use your functions.

The idea is to create a notebook for your slides, and only have relatively short code snippets in the slides, that you can run live during your presentation. (If your code takes quite a long time to run, then you can run it in advance and show the results statically instead.)

For the "deep dive" part, you can exit the slides and show the code directly inside a .py module, or you can copy it over into a code cell directly in your slides – whatever is most convenient.

There is an example in the "Example" folder. In Jupyter Notebook, click "View" > "Cell toolbar" > "Slideshow" to display a bar at the top of each cell allowing you to structure and order your slides. Launch the notebook, and click the "Enter/Exit RISE slideshow" button in the toolbar at the top (the icon looks like a bar chart). In Slideshow mode, you can still edit and run the code in the code cells.

2 Project A: Categorising data: supervised and unsupervised learning in Python

In this project, you will investigate one or both of the following machine learning problems:

- Supervised **classification**, where all items in a data set are classified into two or more labelled categories. The number of categories and their labels are known in advance.
- Unsupervised **clustering**, where the data set is partitioned into a number of *clusters*, based on the similarity between items. The number of categories or labels are *not necessarily* known in advance.

The aim of the project is to perform a classification or clustering task, using at least one method, on at least one dataset.

2.0.1 Implementation

A good reference for machine learning methods is the book *Elements of Statistical Learning*, by Hastie, Tibshirani, and Friedman. You should be able to find all the mathematical background needed for a wide variety of statistical learning methods, and plenty of ideas for evaluation and further investigation.

The scikit-learn module provides a number of algorithms for classification and clustering problems, as well as tools for pre-processing data. You may wish to use these for your project, for instance if you want to investigate the performance of different algorithms with a given dataset, or to evaluate the usefulness of different pre-processing methods. Start with one of the tutorial examples to get to grips with the module's functionality.

Alternatively, you may wish to implement an algorithm from first principles to solve either of these problems. In this case, I would probably recommend a classification algorithm which is relatively straightforward to implement: the *k-nearest-neighbours* algorithm.

k-nearest-neighbours The k-nearest-neighbours algorithm is a classic machine learning algorithm used for classification problems. The data is assumed to be separated into a *training dataset*, where the class (the label) of each item is known, and a *test dataset*, which contains the unlabelled data that you would like to classify.

A basic implementation is as follows: for each item in the test dataset, find the k nearest items in the training dataset – these are your k nearest neighbours. The class of your unlabelled item is then determined by majority voting amongst the classes of these k neighbours. In the case where two or more classes are tied in the vote, the tie is resolved by taking the class of the nearest data item in the training dataset.

There are 3 different aspects of this algorithm which you have control over:

- k, the number of nearest neighbours,
- the *distance metric*, i.e. what is meant by "nearest" – you could start with computing the Euclidean distance between items, and then experiment with other metrics,

- the voting method – majority voting is not the only possibility; for instance, you could weigh the votes according to distances, so that the closest of the k neighbours have more influence on the classification of the test item.

2.0.2 Datasets

This archive provides hundreds of different data sets which you can use for your project. Different datasets are suited for different machine learning problems – you will be able to find many different appropriate datasets for both classification and clustering. If you wish to implement an algorithm from first principles, you may wish to use a fairly simple dataset – a couple of suggestions:

- The iris dataset is a small but very well-known dataset, containing 150 labelled data points, each with 4 attributes representing 4 different measurements on iris flowers. Each data point belongs to one of three species of iris flower.
- The MNIST dataset is much larger, and also very well-known. Each data point is a 28×28 pixel greyscale image of a handwritten digit (0-9), each with 784 attributes (the intensity of each pixel). Each data point (in the training set) is labelled with the correct number.
- The wine dataset contains the results of chemical analysis of 178 different wines, made with grapes from one of three possible types of vine.
- scikit-learn comes with a few built-in datasets, including iris and wine.
- Alternatively, particularly for clustering problems, you may wish to generate your own synthetic data – again, you can do this with scikit-learn.

2.0.3 Resources

- Python Data Science Handbook - a great free online book.
- *Elements of Statistical Learning*, by Hastie, Tibshirani, and Friedman – a solid textbook reference.
- Start Here With Machine Learning – a series of guides and tutorials at different levels.

---

3 Project B: Finite difference method for the heat equation

In this project, you will produce simulations of the evolution of the temperature distribution in a medium, by computing numerical solutions to the heat equation using the **finite difference method**, under different initial and boundary conditions.

3.0.1 The 1D heat equation

Let the temperature distribution over a thin rod of length L be defined as $u(x, t)$, where $t \geq 0$ and $x \in [0, L]$. The evolution of this temperature distribution is governed by the *1D heat equation*,

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2},$$

where α is the thermal diffusivity of the medium. The full initial-boundary value problem is defined by this equation, plus an initial condition $u(x, 0) = f(x)$, representing the temperature distribution along the rod at time $t = 0$, and two boundary conditions, one at each end of the rod. For example, the temperature may be fixed at 0 at both ends, i.e.

$$u(0,t) = u(L,t) = 0.$$

3.0.2 Discretisation

The idea behind finite difference methods is to compute approximated solutions to an initial-boundary value problem like this one, by discretising the domain of definition of u and approximating partial derivative operators with difference operators.

Let $\Delta t > 0$ and $\Delta x > 0$ denote temporal and spatial step sizes, respectively. Consider now a discrete function u_l^n, defined over $n \in \{0, 1, 2, ...\}$ and $l \in \{0, ..., N\}$ (where $N = \frac{L}{\Delta x}$), such that u_l^n is an approximation of the temperature distribution u at time $t_0 = n\Delta t$ and $x_0 = l\Delta x$, that is

$$u_l^n \approx u(l\Delta x, n\Delta t) = u(x_0, t_0).$$

We therefore have, for example,

$$u_{l+1}^n \approx u((l+1)\Delta x, n\Delta t) = u(x_0 + \Delta x, t_0), \tag{1}$$
$$u_l^{n-1} \approx u(l\Delta x, (n-1)\Delta t) = u(x_0, t_0 - \Delta t). \tag{2}$$

Partial derivatives may be approximated by finite differences. To see this, take the definition of the derivative:

$$\frac{\partial u}{\partial t}(x, t_0) = \lim_{\Delta t \to 0} \frac{u(x, t_0 + \Delta t) - u(x, t_0)}{\Delta t}.$$

A finite difference approximation of $\frac{\partial u}{\partial t}$ can be defined by letting Δt take a small, finite value:

$$\frac{\partial u}{\partial t}(x, t_0) \approx \frac{u(x, t_0 + \Delta t) - u(x, t_0)}{\Delta t} \approx \frac{u_l^{n+1} - u_l^n}{\Delta t}.$$

This is the *forward* temporal difference, since it requires an approximation of $u(x,t)$ at a point *forward* in time, $t_0 + \Delta t$.

Similarly, a *centred* finite difference approximation of $\frac{\partial^2 u}{\partial x^2}$ is given by

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{l-1}^n - 2u_l^n + u_{l+1}^n}{(\Delta x)^2}.$$

Using these approximations, we can write a finite difference *scheme* to approximate the 1D heat equation – the *forward-time, centred-space (FTCS) scheme*:

$$\frac{u_l^{n+1} - u_l^n}{\Delta t} = \alpha \frac{u_{l-1}^n - 2u_l^n + u_{l+1}^n}{(\Delta x)^2}.$$

3.0.3 Computing numerical solutions

We can use our FTCS finite difference scheme to compute numerical solutions to the heat equation. The idea is to rearrange it into a *recursion* in time, so that the temperature distribution at a given time step can be directly computed from the distribution at previous time steps. The FTCS scheme can be rearranged to give u_l^{n+1} in terms of u_{l-1}^n, u_l^n, and u_{l+1}^n, all known from the previous step:

$$u_l^{n+1} = \frac{\alpha \Delta t}{(\Delta x)^2} \left(u_{l+1}^n + u_{l-1}^n\right) + \left(1 - \frac{2\alpha \Delta t}{(\Delta x)^2}\right) u_l^n.$$

The initial condition gives the temperature distribution at initial time – that is, we can set $u_l^0 = u(l\Delta x, 0)$. The temperature distribution at future time steps can now be computed recursively, each time for all points l of the rod.

Compute the approximate solution to the 1D heat equation with the FTCS scheme, using the following parameters:

- $\alpha = 0.1$
- $L = 1$
- $u_0^n = u_N^n = 0$ (fixed boundary conditions)
- $\Delta t = 10^{-4}$
- $N = 100$

Initialise the temperature distribution with random values between -1 and 1, and run the simulation for 1000 time steps. Plot the solution dynamically, at every iteration (or every mth iteration, with your choice of m) in the loop, to visualise how the temperature distribution changes over time along the rod – this may be helpful.

3.0.4 Further investigation

Here are some ideas for further investigation – you absolutely don't need to try all of them, and you could also try something I haven't mentioned here.

Different parameters Try different initial distributions, and different sets of boundary conditions. You could have, for instance, zero temperature at one end, and a fixed, positive temperature at the other – find out what the temperature distribution settles to. Try starting with a random temperature distribution, or a distribution with one or more peaks, where the rod would have been pre-heated at specific locations. Try different materials, with different values of α.

You can also include point heat sources, distributed sources, and even moving sources – in this case, you could use linear interpolation to "spread" the source between two grid points, depending on the position $x_s(n\Delta t)$ of the source at time step n. Some of the resources listed below detail how to include source terms.

Stability condition A finite difference scheme is **stable** if it has no solutions which grow exponentially over time. This is to say that *all* discrete solutions of the form $u_l^n = e^{n\lambda\Delta t} e^{i\beta\Delta x}$ must satisfy $\mathrm{Re}(\lambda) \leq 0$ ($\lambda \in \mathbb{C}, \beta \in \mathbb{R}, i = \sqrt{-1}$). Substituting this test solution into the scheme and enforcing the condition on λ will lead to an inequality relating α, Δt, and Δx – this is the **stability condition**. Find it, and verify it by running simulations with parameter values which violate it – the solution should explode.

Other schemes Other finite difference operators may be derived to approximate $\frac{\partial u}{\partial t}$ at different orders of accuracy, by using Taylor series expansions of u around t_0 truncated at different orders. For example, the *backwards* temporal difference is given by

$$\frac{\partial u}{\partial t}(x, t_0) \approx \frac{u(x, t_0) - u(x, t_0 - \Delta t)}{\Delta t} \approx \frac{u_l^n - u_l^{n-1}}{\Delta t}.$$

The backward-time, centred-space (BTCS) scheme is therefore given by:

$$\frac{u_l^n - u_l^{n-1}}{\Delta t} = \frac{u_{l-1}^n - 2u_l^n + u_{l+1}^n}{(\Delta x)^2}.$$

Another scheme, called the *Crank-Nicolson (CN) scheme*, relies on discretising the temporal derivative using the trapezoid rule, and is given by

$$\frac{u_l^{n+1} - u_l^n}{\Delta t} = \frac{1}{2}\left(\frac{u_{l-1}^{n+1} - 2u_l^{n+1} + u_{l+1}^{n+1}}{(\Delta x)^2} + \frac{u_{l-1}^n - 2u_l^n + u_{l+1}^n}{(\Delta x)^2}\right).$$

These two schemes are *unconditionally stable* (you could try to prove this). However, they are *implicit*: if you try to rearrange them into a temporal recursion, you will see that the unknowns to solve for at each time step depend on each other. The way to compute the temperature distribution at the next time step is therefore to write the recursion as a linear system with N equations and N unknowns, and solve it using, for example, `np.linalg.solve()`. The unknowns for the BTCS scheme are the $u_l^n, l \in \{0, \dots, N\}$, and the unknowns for the CN scheme are the $u_l^{n+1}, l \in \{0, \dots, N\}$.

Other PDEs It should be *relatively* straightforward to extend this to the 2D heat equation, to simulate the evolution of the temperature distribution over, say, a rectangular plate. You could also try to compute solutions to the wave equation, for instance, in 1D or 2D.

3.0.5 Resources

- *Finite Difference Schemes and Partial Differential Equations*, J. Strikwerda – a great textbook, available online through the library. In particular, Section 6.3 presents a number of finite difference schemes for the 1D heat equation.
- Solving the Heat, Laplace and Wave equations using finite difference methods – lecture notes by Prof. Anthony Peirce, University of British Columbia
- Thermal diffusivity of different materials
- The diffusion equation – lecture notes by Dr Colm Connaughton at Warwick University

---

4 Project C: Newton and Broyden's methods for nonlinear systems

Consider the following system of n nonlinear equations in n independent variables

$$\mathbf{f}(\mathbf{x}) = \mathbf{0},$$

where $\mathbf{x} \in \mathbb{R}^n$, and $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ is a nonlinear vector-valued function. Suppose that $\mathbf{f}$ is differentiable, and that this system has a unique solution $\mathbf{x}^\star \in \mathbb{R}^n$.

In this project, you will implement two *iterative* methods to compute an approximate solution to a nonlinear system: **Newton's method** and **Broyden's method**. For both these methods, the idea is to start with an initial guess $\mathbf{x}^{(0)}$, and iteratively refine this guess, until convergence to the solution is achieved.

Both methods will be used to compute the solution to the following system:

$$\begin{aligned} f_1(\mathbf{x}) &= 2x_1 - x_2 + \frac{a^2}{2}(x_1 + a + 1)^3, \\ f_i(\mathbf{x}) &= 2x_i - x_{i-1} - x_{i+1} + \frac{a^2}{2}(x_i + ia + 1)^3, \qquad i = 2, \ldots, n-1, \\ f_n(\mathbf{x}) &= 2x_n - x_{n-1} + \frac{a^2}{2}(x_n + na + 1)^3, \end{aligned} \tag{3}$$

where $a = \frac{1}{n+1}$.

4.0.1 Newton's method

The next guess $\mathbf{x}^{(k+1)}$ in Newton's method is computed as

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \left(\mathbf{J}^{(k)}\right)^{-1} \mathbf{f}(\mathbf{x}^{(k)}),$$

where $\mathbf{J}^{(k)}$ is the Jacobian matrix of $\mathbf{f}(\mathbf{x}^{(k)})$. At each iteration, Newton's method therefore requires evaluating the function $\mathbf{f}(\mathbf{x}^{(k)})$, the Jacobian matrix $\mathbf{J}^{(k)}$, and solving a linear system.

Compute the solution of the test system given above using Newton's method, with the initial guess

$$x_i^{(0)} = ia(ia - 1), \qquad i = 1, \ldots, n,$$

for different values of n (up to $n \sim 2000$). Use functionality provided by the `time` module to find out the most computationally expensive operations in each iteration, and report your findings.

4.0.2 Broyden's method

A class of algorithms, called *quasi-Newton methods*, take Newton's method as a starting point. In order to reduce computational cost, these methods approximate the Jacobian matrix instead of evaluating it at every iteration. They are particularly useful when the Jacobian is computationally expensive to evaluate.

The Broyden step substitutes the Jacobian matrix in the Newton step with an approximation $\tilde{\mathbf{J}}^{(k)}$, computed iteratively from $\tilde{\mathbf{J}}^{(k-1)}$:

$$\tilde{\mathbf{J}}^{(k)} = \tilde{\mathbf{J}}^{(k-1)} + \left(\mathbf{y}^{(k)} - \tilde{\mathbf{J}}^{(k-1)}\mathbf{h}^{(k)}\right) \frac{{\mathbf{h}^{(k)}}^\mathsf{T}}{\|\mathbf{h}^{(k)}\|_2^2},$$

where

$$\mathbf{y}^{(k)} = \mathbf{f}(\mathbf{x}^{(k)}) - \mathbf{f}(\mathbf{x}^{(k-1)}), \qquad \mathbf{h}^{(k)} = \mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}.$$

An important thing to note here is that $\tilde{\mathbf{J}}^{(k)}$ is a **rank-1 update** of $\tilde{\mathbf{J}}^{(k-1)}$, and therefore its *inverse* can also be updated iteratively, using the *Sherman-Morrison formula* – see the Resources linked below.

Compute the solution of the test system using Broyden's method and the Sherman-Morrison formula. You will first need to initialise $(\tilde{\mathbf{J}}^{(0)})^{-1} = (\mathbf{J}^{(0)})^{-1}$ explicitly.

Use different values of n, up to $n \sim 2000$, and measure computation times as before.

4.0.3 Further investigation

Try both methods on different nonlinear systems, perhaps systems you have previously seen in your studies or in your research.

Investigate and report the convergence properties of both methods, using different systems and different initial guesses.

There is plenty of recent literature on modified and improved Newton and Broyden methods – you could try implementing one of these.

4.0.4 Resources

- Solving nonlinear equations with Newton's method, C.T. Kelley – Chapter 4: Broyden's method
- Sherman-Morrison-Woodbury – lecture notes by David Bindel, Cornell University

---

5 Project D: Polynomial least squares

Suppose we have a set of data points $\{x^{(i)}, y^{(i)}\}_{i=1,\dots,m}$. The goal is to fit a a polynomial of degree $(n-1)$ to the data, with coefficients $b_j, j = 0, \dots, n-1$, of the form

$$y = b_0 + xb_1 + x^2 b_2 + \dots + x^{n-1} b_{n-1} = \sum_{j=0}^{n-1} x^j b_j.$$

When $m > n$, the polynomial coefficients may be estimated so that the sum of squared residuals between the LHS and the RHS is minimised over the set of data points. The polynomial may be written in matrix-vector form, for all data points, as an overdetermined system:

$$\mathbf{y} = \mathbf{X}\mathbf{b},$$

where $\mathbf{X}$ is the Vandermonde matrix $\mathbf{X} \in \mathbb{R}^{m \times n}$, with elements $\mathbf{X}_{ij} = (x^{(i)})^{j-1}$ for $j = 1, \dots, n$. This system has no exact solution – however, there exists one set of coefficients $\mathbf{b} \in \mathbb{R}^n$ which minimises

$\|\mathbf{Xb} - \mathbf{y}\|_2^2$, the sum of squared residuals. It can be shown that this minimiser solves the **normal equations**

$$\mathbf{X}^T\mathbf{Xb} = \mathbf{X}^T\mathbf{y} \qquad \Rightarrow \qquad \mathbf{b} = \mathbf{X}^\dagger\mathbf{y}, \quad \text{where } \mathbf{X}^\dagger = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T.$$

The matrix $\mathbf{X}^\dagger \in \mathbb{R}^{n\times n}$ is the *Moore-Penrose pseudo-inverse* of $\mathbf{X}$.

You can generate artificial data using a similar process as that seen in W2.4 to generate the noisy sinusoid – use some polynomial as an underlying function instead, and try to estimate the coefficients from the noisy data by solving the normal equations with, e.g., `np.linalg.solve()`.

Try polynomials of different degrees, with coefficients of different magnitudes, with more or less noisy data.

If you recall one of the past worksheet problems, you may have noticed that the matrix $\mathbf{X}^T\mathbf{X}$ is very badly conditioned, which is likely to introduce very large numerical errors when solving the normal equations. There is a way to avoid having to form and invert this matrix: compute and use the **singular value decomposition** (SVD) of $\mathbf{X}$ (see lecture notes linked in Resources).

Finally, a common issue with polynomial regression is **overfitting** – this occurs when a model is complex enough to correspond very closely to the particular set of data points you are training it with, but does not generalise well, and will not be able to predict future observations. Instead of capturing the relationship between input and output in a dataset, an overfitted model describes the statistical noise in the data.

A method to avoid overfitting is **regularisation** – here, we will look at a special case of Tikhonov regularisation (a.k.a. ridge regression for the statisticians in the audience). The regularised normal equations are given by

$$(\mathbf{X}^T\mathbf{X} + \mu\mathbf{I})\mathbf{b} = \mathbf{X}^T\mathbf{y},$$

where $\mu > 0$ is a regularisation parameter. The idea is to introduce a penalty term, so that the polynomial coefficients $\mathbf{b}$ (particularly the high-order coefficients) remain relatively small.

Rewriting the regularised normal equations using the SVD of $\mathbf{X}$ leads to a simple, explicitly computed solution $\mathbf{b}$. Try implementing this with different values of μ, and try to find the best value for a given problem, large enough that overfitting is avoided, but small enough that the model isn't underfitted.

5.0.1 Further investigation

This is straightforwardly generalisable to the multivariate case, when the data points have more than one attribute x. Try to apply the same method to perform polynomial regression on some of the datasets linked in the resources (also see the Project A description).

Versions of polynomial regression are implemented in Numpy (`np.polyfit`), scikit-learn, and many other modules. Compare your results with some of these existing implementations.

Polynomial regression is still a type of multiple *linear* regression – the model is a linear function of the unknown polynomial coefficients. More elaborate regressions can be performed by using other

basis functions than polynomials of x – you can read about these in the linked resources, and try to implement some of them to perform regression on different datasets.

5.0.2 Resources

- *Elements of Statistical Learning*, by Hastie, Tibshirani, and Friedman – Chapter 5: Basis expansions and regularization
- Least squares, pseudo-inverse, and SVD – lecture notes by Guido Gerig, New York University
- Polynomial regression: extending linear models with basis functions – scikit-learn documentation
- Dataset archive

---

6 Project E: Loss of significance in floating-point arithmetic

In this project, you will explore and demonstrate different issues related to *loss of significance* when using floating-point numbers. You will investigate at least two example problems where numerical error plays a significant role; when alternative solutions exist to minimise these issues, you will demonstrate these.

A few example problems to investigate could be (but are not limited to):

- The quadratic formula (a classic!)
- Heron's formula for the area of needle triangles
- Small step sizes in finite difference approximations
- Solving the linear system $Ax = b$ when A is ill-conditioned
- Propagation of error in large sums

6.0.1 Resources

- What Every Computer Scientist Should Know About Floating-Point Arithmetic – I have linked this resource in one of the worksheets. You should be able to find plenty of information there.
- Loss of Significance - The loss of numerical accuracy in computer calculation
- Miscalculating Area and Angles of a Needle-like Triangle, W. Kahan
- Ill-conditioned systems
- Ill-conditioned matrices
- How and How Not to Sum Floating-Point Numbers
- Error and computer arithmetic

---

7 Project F: The FEniCS and Dolfin libraries for numerical PDEs

Dolfin is a Python-based package for advanced scientific computing. Documentation is available here, along with a collection of excellent demos that illustrate the use of the package for various problems.

In this project, you will use FEniCS to compute and visualise solutions for a PDE of your choice. Here is a suggested example:

- Install FEniCS on your computer and study the FEniCS tutorial at https://fenicsproject.org/tutorial/.
- Explore the **Cahn-Hilliard equation** demo by running the code and graphing solutions.
- Modify the Cahn-Hilliard solver above to implement one of the variant methods described in [Ref5].

7.0.1 Resources

- *Introduction to Scientific Programming in Python*, freely available here.
- *Numerical Methods*, Burden and Faires, 3 copies in the library (but not online?).
- Slides from a course at BU on scientific computing in Python.
- FEniCS documentation available here.
- *Numerical methods for solving the Cahn-Hilliard equation and its applicability to related Energy-based models*, T. Tierra and F.Guillen-Gonzalez, 2013. https://core.ac.uk/download/pdf/51405273.pdf

---

8 Project G: Object-oriented Python for numerical ODEs

For this project, study the use of **classes** in Python. There is a good discussion of this in Chapters 8 and 9 of *Introduction to Scientific Programming in Python*. In Chapter 9 of that book there are some examples for numerical differentiation and integration.

Define a class-based framework for solving ordinary differential equations using numerical methods like Euler's method or a 4th order Runge-Kutta method.

8.0.1 Resources

- *Introduction to Scientific Programming in Python*, freely available here.
- *Numerical Methods*, Burden and Faires, 3 copies in the library (but not online?).
- Classes - Python tutorial
- *Python Programming and Numerical Methods - A Guide for Engineers and Scientists* - Chap. 7: Object-oriented programming
- *Python Programming and Numerical Methods - A Guide for Engineers and Scientists* - Chap. 22: Ordinary Differential Equation - Initial Value Problems
- *Applied Scientific Computing* - Differential equations