Numerical Methods in Physics
Exercises

Julius B. Kirkegaard and Mathias S Heltberg

July 24, 2024

Python, Numpy, Scipy

1.1 Loops

In this exercise we will try our hands at Newton's method. We will make our own implementation of the *Lambert W* function $W(z)$, which is defined as the solution x of

$$xe^x = z. \tag{1.1}$$

The goal of this exercise is to fill out[1]:

```
@np.vectorize
def my_lambertw(z):
    # your code here
    return x
```

Newton's method is a numerical root-finding method. We are interested in finding the root (zero) of

$$f(x) = xe^x - z. \tag{1.2}$$

Newton's method is to use the fact that the series

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{1.3}$$

[1] The line `@np.vectorize` enables your function to take numpy arrays and run it on each array entry.

converges towards a root of $f(x)$.

(a) Implement `my_lambertw` by doing a for-loop of 100 Newton iterations starting at $x_0 = 0.0$.

(b) Plot your function on [0, 150] and compare it to scipy's version (`scipy.special.lambertw`).

(c) Change the loop to a while loop that stops when the absolute change to x_n in an iteration is less than 10^{-5}. Plot the comparison again.

1.2 Linear Algebra

Numpy/scipy have excellent linear algebra libraries (`scipy.linalg` is a superset of `numpy.linalg`). Better ones exist, but these will be more than enough for our needs.

(a) Solve the following system of equations using `scipy.linalg.solve`:

$$\begin{aligned} x + 3y - 8z &= 1 \\ 3x - z &= 0 \\ 10y + 3z &= 7 \end{aligned}$$

These libraries can solve *very big* linear equations. Consider a general linear equation

$$\mathbf{A}\boldsymbol{x} = \boldsymbol{b}. \tag{1.4}$$

(b) Sample a 1000×1000 random matrix $\mathbf{A}$ and a random vector $\boldsymbol{b}$ such that each entry is normally distributed (`A = randn(1000, 1000)`

and `b = randn(1000)`) and solve the equation for $\boldsymbol{x}$. Test that $||\mathbf{A}\,\boldsymbol{x} - \boldsymbol{b}||$ is approximately zero, i.e. that the solution found is correct.
Hint: Use `timeit` to time your functions. In jupyter notebooks, `%timeit` or `%%timeit` can be used.

(c) Find the inverse of your random matrix $\mathbf{A}$ (`scipy.linalg.inv`) and solve the equation for $\boldsymbol{x}$. Which method is faster: direct solve or inverse matrix?

(d) If you had to solve many equation of the form $\mathbf{A}\,\boldsymbol{x} = \boldsymbol{b_i}$, where $\mathbf{A}$ is always the same but $\boldsymbol{b_i}$ are different, which approach would be fastest? Test your hypothesis.

1.3 Arrays

In this exercise you will be asked to form various matrices. These might seem random at first. However, it is very important to be able to manipulate array indices. Further, the matrices we choose to consider in this exercise are exactly those that later arise from finite difference methods.

Many of these exercises can be done with smart numpy array manipulation, but feel free to simply use loops. We can optimize for speed later, and further smart numpy approaches will be shown at the end of the exercise.

(a) Write code that can construct an $N \times N$ matrix $\mathbf{A}$ with entries

$$A_{ij} = \begin{cases} 0.5 & j = i + 1 \\ -0.5 & j = i - 1 \\ 0 & \text{otherwise} \end{cases} \tag{1.5}$$

for any N. Print it for size 10×10 (this is a 1D first derivative matrix).

(b) Form an $N \times N$ matrix $\mathbf{B}$ with entries

$$B_{ij} = \begin{cases} -2 & i = j \\ 1 & |i - j| = 1 \\ 0 & \text{otherwise} \end{cases} \tag{1.6}$$

and print it for size 10×10 (this is a 1D second derivative matrix).

(c) Change the first row of $\mathbf{B}$ to all zeros except put a 1 as the first entry. Likewise, change the last row to all zeros, except a 1 as the last entry. Print the matrix for size 10×10.

You are now able to construct the two most commonly used matrices for 1D boundary value problems. We now move on to 2D.

(d) Define $k = N \cdot i + j$. Make an N^2 vector $\boldsymbol{x}$ with entries

$$x_k = j \tag{1.7}$$

for $0 \leq i < N$ and $0 \leq j < N$. Likewise make an N^2 vector $\boldsymbol{y}$ with entries

$$y_k = i \tag{1.8}$$

print both for $N = 5$.
Hint: Simply make a loop over i and j, calculate k and update the entries of x and y.

These vectors set an ordering that may be used to solve 2D PDEs.

(e) Make an $N^2 \times N^2$ matrix $\mathbf{L}$ with entries

$$L_{k\ell} = \begin{cases} -4 & k = \ell \\ 1 & |x_k - x_\ell| = 1 \,\&\, y_k = y_\ell \\ 1 & |y_k - y_\ell| = 1 \,\&\, x_k = x_\ell \\ 0 & \text{otherwise,} \end{cases} \tag{1.9}$$

where $0 \le k < N^2$ and $0 \le \ell < N^2$, and x and y are the vectors defined above. Print the matrix for $N = 4$ (this is a 2D Laplacian matrix using the ordering defined above).
Hint: This exercise can be solved using two nested loops (k and ℓ). It can also be solved with a single loop over k if you think carefully about the ordering of x and y.

(f) *(optional)* Try the following code and compare the result some of the above arrays:

```
n = 10

D1 = (0.5 * np.eye(n, n, 1)
     - 0.5 * np.eye(n, n, -1)

D2 = (np.eye(n, n, 1)
     + np.eye(n, n, -1)
     - 2 * np.eye(n))
```

(g) *(optional)* Try the following code and compare the result to some of the above arrays:

```
xx, yy = np.meshgrid(np.arange(n),
                     np.arange(n))

x = xx.flatten()
y = yy.flatten()
```

(h) *(optional)* Try the following code and compare the result to some of the above arrays:

```
L = (np.kron(D2, np.eye(n))
   + np.kron(np.eye(n), D2))
```

1.4 Sparse Matrices

If a matrix mostly contains zeros (like the ones we just considered) we can store them as *sparse matrices*. These are data structures that only store non-zero entries.

(a) If we had stored Eq. (1.9) as a sparse matrix, how much space could we potentially have saved for $N = 10$, $N = 100$, $N = 1000$ in percentages? (assume that no extra space is needed to make a matrix sparse).

Sparse matrices need to use a special way to remember which numbers belong to which entries. There are many choices and they come with different advantages: some make it easy to add new elements, others make it easy to do matrix multiplication with them. Some are fast at summing across rows, and others across columns.

(b) Construct an empty 10000×10000 sparse matrix $\mathbf{A}$ using `scipy.sparse.lil_matrix`. Fill its diagonals with ones and further set $A_{i,5} = i$ for $0 \leq i < 10,000$.

(c) Convert the matrix to the `csr` format (see documentation of `scipy.sparse`).

(d) Use `scipy.sparse.linalg.spsolve` to solve the equation $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$, where $\boldsymbol{b}$ is a vector of ones (`np.ones(10000)`).

(e) *(optional)* Assume each number stored (zero or otherwise) takes 8 bytes. How much (RAM) memory is needed to store an 100000×100000 matrix? How much memory is needed to store the matrix of **(b)** as a sparse matrix assuming that along with the number you need 8 bytes (two long ints) to store the location of each non-zero entry in matrix.

(f) *(optional)* Using `d2` from exercise 1.3f, try running

```
scipy.sparse.kronsum(d2, d2)
```

and compare to the result of exercise 1.3(e/h).

1.5 Fast Fourier Transforms

In this course, we will solve differential equations using Fast Fourier Transforms or just "fft" (for the spectral method). Therefore we will make a recap of the output of this algorithm.

(a) Consider the signal

$$y(x) = 1 + \cos(2x) + 2\sin(x) \tag{1.10}$$

for $x = np.linspace(0, 4np.pi, 1000)$.

Plot the signal (x vs y), then calculate the fft and plot the output array (call this cn). What frequencies does this correspond to?

(b) We describe the signal as:

$$f(x) = \sum_{n=-N}^{N} c_n e^{i2\pi nx} \tag{1.11}$$

Verify that this can be described as:

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{N} a_n \cos(2\pi nx) + b_n \sin(2\pi nx) \tag{1.12}$$

where

$$c_n = \frac{a_n - ib_n}{2} \quad \text{and} \quad c_{-n} = \frac{a_n + ib_n}{2} \tag{1.13}$$

(c) Use the output array cn and plot the estimated signal ($f(x)$) in the same plot as the true signal ($y(x)$). Are this a good match?

(d) Do the same for the non-oscillatory signal:

$$z(x) = x + x^2 + .1x^3 - .1x^4 \tag{1.14}$$

Obtain the output cn, and plot the signal as well as the resulting estimated signal. Is there a good match? Are there any dominating frequencies?

(e) Finally consider the system of two variables:

```
M, N = 100, 100
x = np.linspace(0, 4 * np.pi, M)
y = np.linspace(0, 4 * np.pi, N)
X, Y = np.meshgrid(x, y)
```

and define:

$$z(X,Y) = 1 + \cos(X) + 2\sin(2Y) + \cos(X + 3Y) \tag{1.15}$$

Use the command fft2 to obtain a 2D matrix of k vectors. Create a double loop over x and y and use sine and cosine functions to obtain the original signal.
hint: The terms should take the form as: $\cos(mx/M + ny/N)$, where m, n in the entry of the matrix.

Numerical Differentiation

2.1 Accuracy of Differentiation

Consider a simple periodic function

$$f(x) = \sin x \tag{2.1}$$

on $x \in [0, 2\pi]$. The analytical derivative is $f'(x) = \cos x$.

On a grid with N data points on $[0, 2\pi)$ we will have $\Delta x = 2\pi/N$. On such a grid, we could evaluate the derivative as

$$f'(x) \approx \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{2.2}$$

(a) Using $N = 20$ gridpoints (`np.linspace(0, 2 * np.pi, 20, endpoint=False)` evaluate the numerical derivative of $f(x)$ using the above formula. Make sure you account for periodicity. Plot the result against the true $f'(x)$.
Hint: `np.roll` might be a useful function.

For smooth functions, we can do better than the above formula.

(b) Make the same plots for the following numerical derivative schemes:

Finite Difference Coefficients for the First Derivative

	$-3\Delta x$	$-2\Delta x$	$-\Delta x$	0	Δx	$2\Delta x$	$3\Delta x$
$\mathcal{O}(\Delta x)$	0	0	0	-1	1	0	0
$\mathcal{O}(\Delta x^2)$	0	0	$-\frac{1}{2}$	0	$\frac{1}{2}$	0	0
$\mathcal{O}(\Delta x^4)$	0	$\frac{1}{12}$	$-\frac{2}{3}$	0	$\frac{2}{3}$	$-\frac{1}{12}$	0
$\mathcal{O}(\Delta x^6)$	$-\frac{1}{60}$	$\frac{3}{20}$	$-\frac{3}{4}$	0	$\frac{3}{4}$	$-\frac{3}{20}$	$\frac{1}{60}$

(c) Evaluate the maximum absolute error made by the methods above over the entire domain $[0, 2\pi)$.

(d) Make a log-log plot of the maximum absolute error of each method as a function of $N \in [10, 10^6]$ (use e.g. `N = np.logspace(1, 6, 50, dtype=int)`) and explain the plot.

(e) What is the best accuracy that you obtain with the second order method? And with the sixth order method? Can you predict the best accuracy for the first order method without evaluating higher N?

(f) How many grid points are needed for a second order method to obtain the same accuracy as a fourth order method with $N = 100$ points?

(g) Compare the curves to a plot of $\sim \Delta x^n$, where n is the order of the method as stated in the above table. Also plot $5 \cdot 10^{-16}/\Delta x$ and compare it to the best error of the methods[1].

[1] $\epsilon \approx 5 \cdot 10^{-16}$ is the machine epsilon if you do calculations using double precision (which is `numpy` default).

2.2 Deriving Schemes

In this exercise you will calculate your own finite difference coefficients.

(a) Use `scipy.linalg.solve` along with the finite difference coefficient formulas of the main text to derive the coefficients of the table in the previous exercise.

At boundaries we cannot use central derivatives.

(b) Derive numerical schemes for the first and second derivatives at x which uses $f(x)$, $f(x+\Delta x)$, $f(x+2\Delta x)$ and $f(x+3\Delta x)$. What are the order of the methods derived?

Now consider the matrix

$$\mathbf{A} = \frac{1}{\Delta x}\begin{pmatrix} -1 & 1 & 0 & 0 & 0 \\ -0.5 & 0 & 0.5 & 0 & 0 \\ 0 & -0.5 & 0 & 0.5 & 0 \\ 0 & 0 & -0.5 & 0 & 0.5 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \tag{2.3}$$

When matrix-multiplied onto a vector this will take the first derivative: $\boldsymbol{f}' \approx \mathbf{A}\boldsymbol{f}$. This matrix uses second order method for all inner points, but only first order methods at the edges.

(c) *(optional)* Build a 10×10 matrix $\mathbf{A}$ such that matrix multiplication $\mathbf{A}\boldsymbol{f}$ calculates the second derivative of $\boldsymbol{f}$ using a second order method everywhere. Use central difference schemes wherever possible, but forward/backward schemes at the edges.
Note: The central scheme $(1, -2, 1)$ is second order despite only using

three stencil points. The forward scheme for the second derivative needs four points to be second order (as derived in (**b**)).

(d) *(optional)* Update your matrix to use a fourth order method at all points.

There is nothing preventing you from having a varying Δx. This is called using an *irregular grid*.

(e) *(optional)* Derive a scheme to make the best possible approximation of the second derivative of a function at $x = 0.5$, where you only have the function values $f(0.0)$, $f(0.1)$, $f(0.25)$, $f(0.6)$, and $f(1.0)$.

2.3 Discontinuities

Consider the following function

$$f(x) = \begin{cases} e^{-x} + ax - 1 & x < 0 \\ x^2 & x > 0 \end{cases} \tag{2.4}$$

which has the derivative

$$f'(x) = \begin{cases} -e^{-x} + a & x < 0 \\ 2x & x > 0 \end{cases} \tag{2.5}$$

(a) Plot the function and its derivative for $a = 0, a = 1, a = 2$.

(b) For what values of $a \in \mathbb{R}$ is $f(0)$ well-defined? i.e. for what values of a do $\lim_{x\to 0^-} f(x) = \lim_{x\to 0^+} f(x)$? What about $f'(0)$?

To numerically avoid the troublesome point $x = 0$, we can choose a grid-spacing that does not include this. This can e.g. be achieved by `x = np.linspace(-1, 1, N)` if N is even. Choose e.g. $N = 1000$.

(c) Using the *central* second order finite difference scheme [$\mathcal{O}(\Delta x^2)$], calculate the numerical derivative of $f(x)$ and compare to the analytical $f'(x)$ for $a = 0, a = 1, a = 2$. Plot your results near $x = 0$ and discuss for what values of a you find a good approximation.

(d) Derive a suitable finite difference scheme for calculating the first derivative of the function in such a way that points from $x < 0$ are never used together with points $x > 0$.

(e) Plot your results near $x = 0$ and compare with the central scheme.

Ordinary Differential Equations

3.1 Explicit vs. Implicit

Consider the ODE

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \alpha \left(\sin t - x\right). \tag{3.1}$$

For $x(0) = 0$, this has the analytical solution

$$x(t) = \frac{\alpha}{1+\alpha^2}\left(e^{-\alpha t} - \cos t + \alpha \sin t\right). \tag{3.2}$$

(a) Solve the equation using the explicit Euler method and compare with the analytical solution for $\alpha = 0.1$ on $t \in [0, 100]$ using $\Delta t = 0.01$.

(b) Do the same for the implicit Euler method.

(c) Investigate the behaviour of the two methods for $\alpha > 200$.

3.2 Van der Pol oscillator

The equation

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} - \mu(1-x^2)\frac{\mathrm{d}x}{\mathrm{d}t} + x = 0 \tag{3.3}$$

is called the Van der Pol oscillator.

(a) Rewrite the equation to a system of two coupled first-order ODEs.

(b) Solve the equation using the Euler method for $\mu = 0.1$, $\mu = 1.0$, $\mu = 10$, $\mu = 100$, and $\mu = 250$ with initial conditions $x(0) = 2$ and $x'(0) = 0$. Solve for $t \in [0, 10\mu]$ and use $\Delta t = 0.01$.

(c) Solve the equation using SCIPY's `solve_ivp` with `method='BDF'` (this is a higher-order implicit method, "backwards differentiation formula").

(d) Plot the Euler and the BDF solution in the same plot for each value of μ.

As μ increases, the equations becomes more and more stiff. Thus implicit methods (such as BDF) become important.

(e) Solve the equation using `solve_ivp`'s `method='RK45'` and report the run time compared to that of BDF for each value of μ.

3.3 Runge–Kutta method

Independent exercise

The main text gives the following formula for the fourth order Runge–

Kutta method

$$
\begin{aligned}
k_1 &= F(f(t), t) \\
k_2 &= F(f(t) + \frac{1}{2}k_1\Delta t, t + \frac{1}{2}\Delta t) \\
k_3 &= F(f(t) + \frac{1}{2}k_2\Delta t, t + \frac{1}{2}\Delta t) \\
k_4 &= F(f(t) + k_3\Delta t, t + \Delta t) \\
f(t + \Delta t) &= f(t) + \frac{1}{6}\left[k_1 + 2k_2 + 2k_3 + k_4\right]\Delta t
\end{aligned}
\tag{3.4}
$$

(a) Implement this method to solve

$$f'(t) = 1 + \sin(t) f(t) \tag{3.5}$$

for $t \in [0, 15]$ using $\Delta t = 0.001$ and $f(0) = 0$. Also implement the Euler method for this ODE and check that it gives the same result.

(b) Now take $\Delta t = 1.0$ and solve the equation using the Runge–Kutta method. Plot the solution on top of the result with $\Delta t = 0.001$.

(c) Take $\Delta t = 0.25$ and solve the equation using the Euler method. Plot the solution on top of the result with $\Delta t = 0.001$ and comment on the result.

3.4 Dormand–Prince method*

In this exercise you are going to write your own version of the infamous 'ode45' method (aka 'RK45'). Upon completion you should feel enormously more confident using library version of this method as you will

understand all its inner working, including how errors are controlled. It is recommended to code this exercise in a function, for instance called `dopri45(F, ...)`.

Consider a general initial value problem for an ODE

$$\frac{\mathrm{d}x}{\mathrm{d}t} = F(x(t),\, t). \tag{3.6}$$

The Dormand–Prince method is an adaptive fifth order Runge–Kutta method. Its first four k-expressions are

$$\begin{aligned}
k_1 &= F(x,\, t) \\
k_2 &= F\left(x + \frac{1}{5}\, k_1 \Delta t,\, t + \frac{1}{5}\, \Delta t\right) \\
k_3 &= F\left(x + \frac{3}{40}\, k_1 \Delta t + \frac{9}{40}\, k_2 \Delta t,\, t + \frac{3}{10}\, \Delta t\right) \\
k_4 &= F\left(x + \frac{44}{45}\, k_1 \Delta t - \frac{56}{15}\, k_2 \Delta t + \frac{32}{9} k_3 \Delta t,\, t + \frac{4}{5}\, \Delta t\right)
\end{aligned}$$

The full method uses 7 k-expressions defined by the following table:

0						
$1/5$	$1/5$					
$3/10$	$3/40$	$9/40$				
$4/5$	$44/45$	$-56/15$	$32/9$			
$8/9$	$19372/6561$	$-25360/2187$	$64448/6561$	$-212/729$		
1	$9017/3168$	$-355/33$	$46732/5247$	$49/176$	$-5103/18656$	
1	$35/384$	0	$500/1113$	$125/192$	$-2187/6784$	$11/84$

where the first column is the Δt factor.

(a) Assume that f is one-dimensional and write a function that calculates all k-expressions.

Using these k-expressions a fifth order accurate estimate of $f(t + \Delta t)$ is

$$f_4(t + \Delta t) = f(t) + \left[\frac{35}{384} k_1 + \frac{500}{1113} k_3 + \frac{125}{192} k_4 - \frac{2187}{6784} k_5 + \frac{11}{84} k_6 \right] \Delta t.$$

A sixth order approximation to $f(t + \Delta t)$ is given by

$$f_5(t + \Delta t) = f(t) + \left[\frac{5179}{57600} k_1 + \frac{7571}{16695} k_3 + \frac{393}{640} k_4 - \frac{92097}{339200} k_5 + \frac{187}{2100} k_6 + \frac{1}{40} k_7 \right] \Delta t$$

(b) Calculate $f_4(t + \Delta t)$ and $f_5(t + \Delta t)$ and estimate the error of the step as $\mathcal{E} = |f_5 - f_4|$.

We define a tolerance ϵ_{abs} that we want to keep the error below. If the error $\mathcal{E}$ is smaller than ϵ_{abs}, we accept the time step. Otherwise we reject it and redo the step with a smaller Δt. In any case, we change Δt since in the case where the error is much smaller than ϵ_{abs}, we can afford to take bigger time steps.

Since error estimate is fifth order, the best estimate for the optimal Δt is[1]

$$\Delta t_{\text{new}} = \left| \frac{\epsilon_{\text{abs}}}{\mathcal{E}} \right|^{1/5} \Delta t. \tag{3.7}$$

However, to avoid rejection we want to a bit conservative and choose a Δt slightly smaller than this. Also we do not want Δt to change too rapidly.

[1] "Opitmal" means one which has $\mathcal{E} \sim \epsilon$.

(c) Make a function that updates Δt to 0.9 times that of Eq. (3.7). Further ensure that Δt is never increased more than a factor two.

If the function $f(t)$ is large (in the absolute sense), relative errors become more important than absolute errors. One therefore also includes a requirement on the relative error. Defining the relative tolerance ϵ_{rel} we can define a total tolerance as

$$\epsilon = \epsilon_{\text{abs}} + \epsilon_{\text{rel}}|f(t)|. \tag{3.8}$$

(d) Update the Δt_{new} function to accept both ϵ_{abs} and ϵ_{rel} and use the full tolerance of Eq. (3.8).

(e) Finish your implementation of the ODE solver and test it on Eq. (3.5). Compare the Δt chosen by your algorithm to that of `scipy.integrate.solve_ivp`.

Since the algorithm chooses Δt you have no control of the time points that are returned. Scipy's `solve_ivp` solves this by doing interpolation to `t_eval` after integration. A different approach is to change Δt to exactly hit the grid points. If $t + \Delta t$ is larger than the next grid point t_i, then simply change Δt to $t_i - t$.

(f) *(optional)* Extend your ODE solver to take an argument `t_eval` and use the above idea to evaluate the ODE at those points.

3.5 1D Poisson Equation

Solve

$$\frac{\mathrm{d}^2 f(x)}{\mathrm{d}x^2} + e^{-x^2} = 0 \tag{3.9}$$

on $[-5, 5]$ with boundary conditions $f(-5) = 1$ and $f'(5) = 0$ using a second order finite difference scheme with $\Delta x = 0.01$.

Plot the curve and check visually that the boundary conditions are satisfied and that the curvature of the curve is highest at $x = 0$ (which is what the ODE specifies).

3.6 Diffusion–Advection

The one-dimensional Diffusion-Advection equation is

$$D\frac{\mathrm{d}^2 f(x)}{\mathrm{d}x^2} - \frac{\mathrm{d}}{\mathrm{d}x}\left(v(x) f(x)\right) = 0, \tag{3.10}$$

where $v(x)$ is some velocity field that *advects* particles around. The first term is the diffusion term that tends to spread particles.

(a) Use a second order finite difference scheme to solve the diffusion-advection with $D = 2$ and $v(x) = -\sin x$ on $[0, 25]$ with boundary conditions $f(0) = 1$ and $f(25) = 0$. Use $N = 1000$ grid points.

It is always a good idea to 'sanity check' the result of a numerical solution.

(b) Explain the shape of $f(x)$. Does it make sense compared to the physical interpretation of the diffusion-advection equation?

(c) Explain what happens for $D \to 0$ and $D \to \infty$. Plot for instance $D = 0.5$ and $D = 15$ and compare to $D = 2$. Would your code work for $D = 0$?

3.7 Beam Equation

The bending $y(x)$ of an elastic rod subject to a load $w(x)$ along its length is described by the Euler–Bernoulli beam equation:

$$y''''(x) = w(x). \tag{3.11}$$

We will consider this ODE on $x \in [0, 1]$. We fix the left end of the rod to a wall giving the boundary conditions $y(0) = 0$ and $y'(0) = 0$. The other end hangs free giving the boundary conditions $y''(1) = 0$ and $y'''(1) = 0$.

(a) Solve the ODE with $w(x) = -1.0$ using finite differences over $N = 100$ grid points and plot the result.

We now move all the load to a single point along the rod:

(b) Solve the ODE with $w(x) = -\delta(x - x_0)$ for $x_0 = 0.2$, $x_0 = 0.5$ and $x_0 = 0.9$, and plot the results.
Note: On a finite grid the Dirac delta function becomes e.g. $(0, 0, \cdots, 0, 0, 1/\Delta x, 0, 0, \cdots, 0, 0)$.

(c) Do the curves that you obtain make sense physically?

3.8 Irregular grids*

Take

$$\frac{\mathrm{d}^2 f(x)}{\mathrm{d}x^2} = -e^x f(x). \tag{3.12}$$

We will consider this ODE on $[0, 10]$ with the boundary conditions $f(0) = 1$ and $f(10) = 0$.

This, in fact, has an analytical solution in terms of Bessel functions. In Python this solution may be implemented as

```
from numpy import exp, sqrt
from scipy.special import jv, yn

def f(x):
    return ((jv(0, 2 * sqrt(exp(x)))
             * yn(0, 2 * exp(5))
             - jv(0, 2 * exp(5))
             * yn(0, 2 * sqrt(exp(x))))
            / (-jv(0, 2 * exp(5)) * yn(0, 2)
               + jv(0, 2) * yn(0, 2 * exp(5))))
```

We will limit ourselves to 750 points on the interval (it is common to have to restrict the number of grid points to allow fast computations).

(a) Plot the function using 750 linearly spaced grid points (`x = np.linspace(0, 10, 750)`) and compare it to a plot with 10,000 grid points.

(b) Derive a sixth order finite difference scheme (using eight stencil points) and use this to solve the ODE numerically on the grid of 750

points. Compare with the analytical solution. What happens if you use 1500 points?

Instead of using linearly spaced grid points, we can use irregularly spaced grid points. Often more spatial resolution is needed in certain regions. Here is a function to generate an irregularly spaced grid that has higher resolution for higher x:

```
def grid(n=750):
    x = np.linspace(0, 10, n)
    xn = 1 - np.exp(-x / 4)
    return 10 * xn / np.max(xn)
```

(c) Plot the function using 750 irregularly spaced grid points generated using the above function. Compare to the plot of 750 regularly spaced grid points.

(d) Derive a sixth order finite difference scheme for this irregularly spaced grid and use it solve the ODE. Compare with the analytical solution and the regularly spaced numerical solution.

(e) *(optional)* Do the calculations using sparse matrix representations (`scipy.sparse.lil_matrix` for construction and `.to_csr()` and `scipy.sparse.linalg.solve` for solving). Time it to compare the speed of solution.

3.9 Spectral Method

Consider the ODE

$$f(x) - \alpha \frac{\mathrm{d}^2 f(x)}{\mathrm{d}x^2} - \beta \frac{\mathrm{d}^4 f(x)}{\mathrm{d}x^4} = \cos(x)\sin(x)^5 \tag{3.13}$$

with periodic boundary condition for $x \in [0, 2\pi]$ and $\alpha = 1.0$ and $\beta = 0.1$.

(a) Use 1000 gridpoints and solve the ODE using the spectral method (use `np.fft.rfft`, `np.fft.rfftfreq` and `np.fft.irfft`). Use finite differences to take derivatives of your solution and check that the ODE is indeed satisfied.

Hint: Use `k = 2 * np.pi * np.fft.rfftfreq(1000, dx)`.

(b) Solve the equation again using the spectral method and only 20 grid points and compare to the solution with 1000 grid points.

(c) Solve the equation using a finite difference scheme and only 20 grid points and compare to the spectral solution.

3.10 Non-Linearities

Consider the ODE

$$\frac{\mathrm{d}^2 f(x)}{\mathrm{d}^2 x} = f(x)^3 - f(x)\frac{\mathrm{d}f(x)}{\mathrm{d}x} \tag{3.14}$$

on [1, 10] with boundary conditions $f(1) = 1/2$ and $f(10) = 1/11$. We chose this non-linear ODE very carefully such that it has an analytical solution. Small changes to the ODE will make numerical solution a necessity. The solution is $f(x) = 1/(1 + x)$.

This is actually quite a hard ODE to solve numerically using some approaches (for instance Mathemtica's `NDSolve` fails to solve it). Here, we will use a relaxation approach to iterate towards the solution. In particular, the following is a linear equation in $f_n(x)$:

$$\frac{\mathrm{d}^2 f_n(x)}{\mathrm{d}^2 x} = f_n(x) f_{n-1}(x)^2 - f_n'(x) f_{n-1}(x). \tag{3.15}$$

If f_{n-1} is close to f_n, then the solution to this is ODE is also close to a solution to the ODE of interest. If f_{n-1} is not close to f_n, then the hope is that solving this linear ODE brings us closer to the target. By iterating many times we should approach the true solution.

For instance, if $f_{n-1}(x) = x$ we simply have the linear ODE

$$\frac{\mathrm{d}^2 f_n(x)}{\mathrm{d}^2 x} = x^2 f_n(x) - x f_n'(x). \tag{3.16}$$

(a) As a warmup solve Eq. (3.16) using finite differences with $N = 100$ grid points.

(b) Take $f_0(x) = x$ (or anything else you desire, even $f(x) = 0$ should work) and solve for f_n successively using Eq. (3.15). Do 20 iterations and make a plot showing that the solutions converge towards the analytical solution.

We are not guaranteed to converge towards a correct solution. Therefore when using relaxation you should always check that the final iteration satisfies to the equations.

(c) Plot $\mathrm{mean}_x |f_n''(x) - f_n(x)^3 + f_n'(x) f_n(x)|$ and $\max_x |f_n''(x) - f_n(x)^3 + f_n'(x) f_n(x)|$ for each iteration n and show that it approaches

zero, i.e. that the equations are satisfied. Ignore the boundary points in these calculations.

Partial Differential Equations

4.1 1D Heat Equation

We begin by considering a partial differential equation with only one spatial direction. This is basically a 1D boundary value once we discretise time.

Consider the one dimensional heat equation

$$\frac{\partial f(x,t)}{\partial t} = \frac{\partial^2 f(x,t)}{\partial x^2}. \tag{4.1}$$

with boundary conditions $f(0,t) = 1$ and $\partial_x f(1,t) = 0$. This means that we keep the left end at temperature 1 and let no heat escape or enter at the right end.

The steady state of this equation is clearly $f(x,t) = 1$. We will consider the time evolution starting from $f(x,0) = e^{-5x}$.

Using implicit time discretisation we have

$$f_{t+\Delta t}(x) = f_t(x) + f''_{t+\Delta t}(x)\,\Delta t. \tag{4.2}$$

(a) Use a second order finite difference scheme to turn the above into a linear algebra problem using $N = 1000$ grid points.

(b) Solve the system using $\Delta t = 0.05$ for $t \in [0, 3]$ and plot curves for $t \in \{0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0\}$.

(c) *(optional)* At each time step we actually have a very good guess for the solution. Namely, $x_{t+\Delta t}$ will not not be very much different from

x_t assuming Δt is small. Some solvers iterate towards a solution and they can be given an initial guess. Use `scipy.sparse.linalg.bicg` to iteratively solve for $x_{t+\Delta t}$ using `x0` $= x_t$.

4.2 2D Heat Equation

Hint: The results of exercise 1.3(d–) are useful in this exercise.

In this exercise we will consider the 2D heat equation:

$$\frac{\partial f(\boldsymbol{x})}{\partial t} = c\nabla^2 f(\boldsymbol{x}) + s(\boldsymbol{x}). \tag{4.3}$$

Here $f(\boldsymbol{x})$ is the temperature field, c is the heat conductivity, and $s(\boldsymbol{x})$ is a source (or sink, if negative) of heat. We take $c = 1$.

We will begin by taking $s(\boldsymbol{x}) = 0$ and look at the steady state. In this case the heat equation becomes Laplace's equation

$$\nabla^2 f(\boldsymbol{x}) = 0. \tag{4.4}$$

The solution to this equation is the temperature distribution after a long time. We will begin by considering a 2D square whose edges are kept at fixed temperatures:

$$f(\boldsymbol{x}) = \begin{cases} 2 & \text{on the left edge} \\ 1 & \text{on the top edge} \\ 0 & \text{on the bottom edge} \\ 1 & \text{on the right edge} \end{cases} \tag{4.5}$$

These are *Dirichlet* boundary conditions.

(a) Solve Eq. (4.4) with the above boundary conditions by forming a finite difference matrix and using `scipy.linalg.solve` or `scipy.sparse.linalg.spsolve` (faster). Use a square grid with $N = 50$ points along each direction and $\Delta x = 0.1$. Ignore issues that arise due to conflicting boundary conditions at the corners. Plot the result.
Hint: if you use the ordering defined in Exercise 1.3 you can use `x` and `y` of that exercise to locate the rows in your Laplacian matrix that correspond to boundary points.
Hint: You can use `imshow(res.reshape(N, N), origin='lower')` to plot your result.

(b) *(optional)* Change the top edge boundary condition to Neumann: $\partial_y f(\boldsymbol{x}) = 0$ and solve the equation again. Plot the result.

After this warmup (pun?) we will move on to the full heat equation. We will simulate the heating of an electrical stove. Define the field

$$s(\boldsymbol{x}) = \begin{cases} 1 & \text{if } 0.8 < r(\boldsymbol{x}) < 1.2 \\ 0 & \text{otherwise,} \end{cases} \tag{4.6}$$

where r is the distance to the center of the domain.

(c) Solve Eq. (4.3) for $t \in [0, 1.5]$ using $\Delta t = 0.01$ with the same discretisation of space as used above. Take $f(\boldsymbol{x}, t) = 0$ as the boundary conditions on all edges and initial condition $f(\boldsymbol{x}, 0) = 0$.

(d) Plot a series of images/an animation of the simulation using a suitable color mapping (e.g. `imshow(..., vmin=0, vmax=0.25, cmap='hot')`).

(e) *(optional)* What does our current boundary condition correspond to? Would it be better to use a (perhaps nonzero) Neumann boundary condition? What about a Robin boundary condition of the form $\partial_{\boldsymbol{x}} f(\boldsymbol{x}, t) = \alpha[f(\boldsymbol{x}, t) - f_0]$?

(f) *(optional)* We are modelling a stove as a 2D system, but in reality heat can also escape in the third dimension. Could we change the PDE to approximately account for this without moving to a full 3D simulation?

4.3 Shade sail

Independent exercise

We are designing a square sun sail for a Danish garden. It rains a lot in Denmark, so to ensure that water can drip off easily, we decide to hang it by four ropes tied to the following points

$$\begin{cases} A = (-1, -1, -1), \\ B = (1, 1, -1), \\ C = (-1, 1, 1), \\ D = (1, -1, 1). \end{cases} \tag{4.7}$$

Our four ropes are tightly bound between A-C, A-D, B-C, B-D.

(a) Plot the four ropes in a 3D plot.
You can use `ax = plt.figure().add_subplot(projection='3d')` to make a 3D plot followed by `ax.plot(x, y, z)`.

The sail's shape will be determined by Laplace's equation

$$\nabla^2 f(x, y) = 0, \tag{4.8}$$

where f here represents the z-coordinate, which we solve on the square $x, y \in [-1, 1] \times [-1, 1]$. Our boundary conditions are Dirichlet and set by the fact that the sail is tied to the ropes.

(b) Write down the boundary conditions.

(c) Solve Eq. (4.8) with the boundary conditions.

(d) Plot your solution together with the ropes.
You can use[1] `ax.plot_surface(X, Y, Z, antialiased=False)`.

4.4 2D Atoms*

The time-independent Schrodinger equation is

$$\mathcal{H}\psi = E\psi, \tag{4.9}$$

where the Hamiltonian is

$$\mathcal{H} = -\nabla^2 + V(\boldsymbol{x}) \tag{4.10}$$

in some suitable, natural units.

In this exercise we consider how orbitals would have been if atoms were two-dimensional. We intend to find the wave-function of electrons surrounding a proton ("2D hydrogen"). In suitable units we therefore have[2]

$$V(\boldsymbol{x}) = -\frac{\alpha}{|\boldsymbol{x}|}, \tag{4.11}$$

[1] Note that 3D plotting in matplotlib is a bit buggy: objects do not always appear at the correct depth.

[2] There are good arguments that in 2D the Coulomb potential should be $V(\boldsymbol{x}) = \log|\boldsymbol{x}|$, but we choose to consider $1/|\boldsymbol{x}|$.

where the negative sign signifies attraction between the electron and proton (which we position at $\boldsymbol{x} = 0$). We start with taking $\alpha = 1$.

Classical problems of orbitals are solved in infinite space. Here we will limit ourselves to a small box (this corresponds to having $V(\boldsymbol{x}) = \infty$ outside our grid — a *confined* orbit). We will therefore assume $\psi(\boldsymbol{x}) = 0$ outside the domain. At a border point you therefore have, say at the left edge, $\partial_x^2 f(x_0, y) \approx \big[f(x_0 - \Delta x, y) + f(x_0 + \Delta x, y) - 2f(x_0, y)\big]/\Delta x^2 = \big[f(x_0 + \Delta x, y) - 2f(x_0, y)\big]/\Delta x^2$.

(a) Construct a finite-difference approximation $\mathbf{H}$ of $\mathcal{H}$. Use $\Delta x = 0.01$ on a 40×40 grid.

(b) Calculate the eigenvalues and eigenvectors of $\mathbf{H}$ (`scipy.linalg.eig`). Sort the eigenvalues and plot the magnitude of the first 100 values. Discuss the plot.

(c) Plot the 2D heatmap ("orbitals") of the squared eignvectors $|\psi|^2$ for the 25 smallest eigenvalues (energies).

(d) Increase α and explain its effect on the orbitals.

(e) *(optional)* Update your code to use a sparse solver and solve for the smallest 25 eigenvalues only (`scipy.sparse.linalg.eigs` with `which='SM'`). Do you find a speed improvement? (the eigenvectors might be different as there is degeneracy in their choice, but the eigenvalues should be the same).

(f) *(optional)* We are breaking symmetry by putting a polar symmetric atom in a square box. Change the code to solve in polar coordinates.

Hints: In polar coordinates

$$\nabla^2 = \frac{\partial^2}{\partial r^2} + \frac{1}{r}\frac{\partial}{\partial r} + \frac{1}{r^2}\frac{\partial^2}{\partial \theta^2}. \tag{4.12}$$

You should solve on a grid $r \in (\Delta r, R)$ and use boundary conditions $f'(0,\theta) = 0$ and $f(R,\theta) = 0$. This means that $\partial_r^2 f(\Delta r, \theta) \approx [f(0,\theta) + f(2\Delta r,\theta) - 2f(\Delta r,\theta)]/\Delta r^2 = [f(2\Delta r,\theta) - 1f(\Delta r,\theta)]/\Delta r^2$ Use periodic boundary conditions in θ.

(g) *(optional)* Discuss how you would solve for the orbitals of two electrons orbiting a single core ("2D helium"). *Hints:* Whereas we solved for $\psi(x,y)$ before we now need to solve for $\psi(x_1,y_1,x_2,y_2)$. How should $\mathcal{H}$ be changed? What about $V(\boldsymbol{x})$ specifically? Both electrons needs to be attracted to the proton, but they must also be repelled from one another.

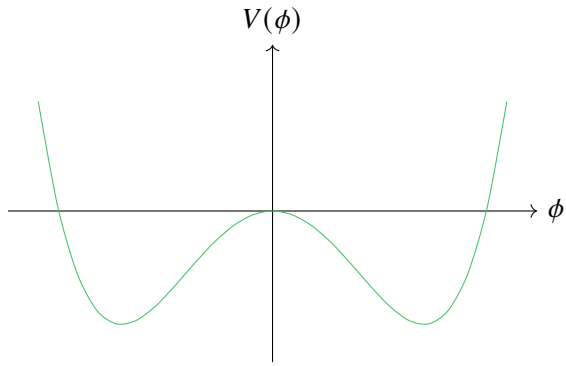
4.5 Landau–Ginzburg Model

Physical details of the Ising model are described in Exercise 5.5.

In this exercise we will consider a continuum approximation of the Ising model when $T < T_c$. That is we will simulate the evolution of a continuous spin field that tends to align neighbouring spins. In particular, we consider a field $\phi(\boldsymbol{x},t)$ that evolves according to a Landau–Ginzburg equation

$$\partial_t \phi(\boldsymbol{x},t) = \phi - \phi^3 + \nabla^2\phi \tag{4.13}$$

The terms $\phi - \phi^3$ arise from a potential $V = \frac{1}{4}\phi^4 - \frac{1}{2}\phi^2$ which have minima at $\phi = -1$ and $\phi = 1$:



The terms will thus ensure that the spins prefer these values. The Laplacian term prefers aligned fields, and will thus make neighbouring spins aligned. Noise would needed to be added to the model in order to consider $T > T_c$.

(a) Initialise a random two-dimensional field ϕ and evolve it with periodic boundary conditions according to the Landau–Ginzburg model using a finite difference scheme or a spectral method. The non-linear term ϕ^3 can be dealt with using a semi-implicit step.
Hint: Try e.g. $N = 50$, $\Delta x = 1.0$ and $\Delta t = 0.5$, or test how large a grid you can make work (depends on the method used).

(b) Plot the field ϕ for a few times steps and comment on the result. What does ϕ tend to for large t?

Stochastic Systems

5.1 Bak–Sneppen Model

We begin stochastic modelling with a model that is extremely simple, yet still has rich dynamics. The model is motivated by the evolution of species. Consider N different species. We place these on a ring and decide that they only interact with their neighbours. So species i interacts with species $i-1$ and $i+1$. With periodicity, species 0 interacts with species 1 and $N-1$.

Each species has a fitness f_i which is a number between 0 and 1: 0 meaning a really terrible species and 1 meaning highest fitness.

(a) Initialise $N = 1000$ species each with a random fitness $f_i \in [0, 1]$.

Each time step of the model is a very simple rule: find the species with the lowest fitness and mutate it and its immediate neighbours to a new random fitness $[0, 1]$ (the neighbours are also mutated since *fitness* is a context-dependent measure. The fitness of one species depends on the species it interacts with).

(b) Run the simulation for 100,000 time steps.

(c) Plot the minimum fitness as a function of time.

(d) Plot a heatmap (location i along the x-axis and time along the

y-axis) of the "age" of each species. In other words, plot a heatmap where at a given location you plot the time since a species was last replaced at that location. You should hopefully find that as time goes on, species tend to stay around longer and longer.
Hint: only plot every 100th time step along the y-axis to speed up your code.

(e) Based on the above plot, make a comment about *mass extinctions*.

5.2 Chemical Reactions

CO_2 is soluble in water, which it may further react with to form carbonic acid:

$$CO_2 + H_2O \underset{k_2}{\overset{k_1}{\rightleftharpoons}} H_2CO_3. \tag{5.1}$$

We will consider a really small system where we have just $N_{CO_2} = 10000$ molecules of CO_2, but "infinitely" many H_2O molecules. We start with no carbonic acid. The rate of the forward reaction is

$$R_{\rightarrow} = k_1 N_{CO_2} \tag{5.2}$$

and backward reaction

$$R_{\leftarrow} = k_2 N_{H_2CO_3}, \tag{5.3}$$

where $k_1 = 10^{-3}$ and $k_2 = 1.0$.

If we had an infinite number of molecules, the concentration c would follow the ODEs

$$\frac{\mathrm{d}c_{\mathrm{CO}_2}(t)}{\mathrm{d}t} = -k_1 c_{\mathrm{CO}_2}(t) + k_2 c_{\mathrm{H_2CO_3}}(t), \tag{5.4}$$

$$\frac{\mathrm{d}c_{\mathrm{H_2CO_3}}(t)}{\mathrm{d}t} = k_1 c_{\mathrm{CO}_2}(t) - k_2 c_{\mathrm{H_2CO_3}}(t). \tag{5.5}$$

(a) Solve and plot the solution to the ODEs with $c_{\mathrm{CO}_2}(0) = 1$ and $c_{\mathrm{H_2CO_3}}(0) = 0$ for $t \in [0, 20]$ using any method you wish and plot $c_{\mathrm{H_2CO_3}}(t)$.

For smaller systems, such as $N = 10000$ there will be a lot of noise. To simulate this we will use the Gillespie method.

(b) Simulate the system five times using the Gillespie algorithm and plot $N_{\mathrm{H_2CO_3}}(t)/10000$ in the same plot as $c_{\mathrm{H_2CO_3}}(t)$. Use `plt.step(..., where='post')` to make the plot.

(c) Make the same the plots for $N = 1000$, $N = 10000$, $N = 100000$, and $N = 1000000$.

(d) *(optional)* As N is increased, the simulation becomes slower and slower. For large N, however, we can use the approximate τ-stepping method. Implement this and plot the solution for $N = 1000$, $N = 10000$, $N = 100000$, and $N = 1000000$ using $\Delta t = 0.1$.

5.3 Local Voter model

Independent exercise

Consider an American neighbourhood of $N = 50$ residents. Each resident identifies as democratic (D) or republican (R).

The following four events can happen (with rates[1]):

$$\begin{cases} \text{A democratic randomly becomes republican,} & \text{rate} = 0.1D \\ \text{A republican randomly becomes democratic,} & \text{rate} = 0.1R \\ \text{A republican convinces a democratic to become republican,} & \text{rate} = 0.01DR \\ \text{A democratic convinces a republican to become democratic.} & \text{rate} = 0.01RD \end{cases}$$

(a) Explain what the rates mean. Why are the 'convincing' rates proportional to $D \cdot R$?

(b) Initialize a system with $R = 25$ and $D = 25$ and simulate the above rate system using the Gillespie method for 500,000 steps.

(c) Plot R and D as a function of time and discuss the result.

We now introduce a third type of resident: undecided (U). The rules are then updated such that when someone is convinced to leave their party, they become undecided, and thus needs to be convinced twice to be

[1] In reality, these rates are probably near-zero.

converted to the other party. The random rates are thus

$$\begin{cases} \text{A democratic randomly becomes undecided,} & \text{rate} = 0.1D \\ \text{A republican randomly becomes undecided,} & \text{rate} = 0.1R \\ \text{An undecided randomly becomes democratic,} & \text{rate} = 0.05U \\ \text{An undecided randomly becomes republican,} & \text{rate} = 0.05U \end{cases}$$

and the convincing rates

$$\begin{cases} \text{A republican convinces a democratic to become undecided,} & \text{rate} = 0.01DR \\ \text{A republican convinces an undecided to become republican,} & \text{rate} = 0.01UR \\ \text{A democratic convinces a republican to become undecided,} & \text{rate} = 0.01RD \\ \text{A democratic convinces an undecided to become democratic.} & \text{rate} = 0.01UD \end{cases}$$

(d) Initialize a system with $R = 0$, $U = 50$ and $D = 0$ and simulate the above rate system using the Gillespie method for 500,000 steps.

(e) Plot R and D as a function of time and discuss the result.

(f) (*optional*) How would a mean-field ODE description [e.g. similar to that of Eq. (5.4)] for this system behave?

5.4 Damped Harmonic Oscillator with stochastic noise

Consider the system of equations:

$$\Delta X = Y\Delta t$$
$$\Delta Y = (-\omega_0^2 X - \delta Y)\Delta t + \sigma \Delta W(t)$$

Where ω_0, δ and σ are parameters.

(a) Choose $\omega_0 = 1$, $\delta = .1$ and $\sigma = 0$. Simulate the system for $t \in [0; 100]$ using the Euler-Muryama method with $\Delta t = 0.01$. Compare this to the same system but with $\sigma = 1$. What does the stochasticity do?

(b) Now choose $\delta = 0, \sigma = 1$ and simulate for $t \in [0; 1000]$. What happens to the system? Why is this?

(c) Fix $\delta = .1$ and vary $\sigma \in [0.1; 10]$. How does the amplitude of oscillations grow?

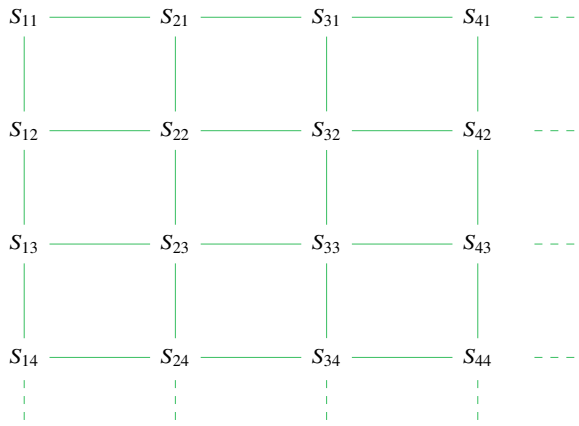
5.5 Ising Model

The Ising model is a simple model for ferromagnetism. Atoms with spin $S \in \{-1, 1\}$ are given fixed locations and their spin interactions are limited to nearest neighbours. The total energy (Hamiltonian) of the system is therefore

$$\mathcal{H} = -\sum_{\langle ij \rangle} S_i S_j, \tag{5.6}$$

where $\langle ij \rangle$ denotes neighbouring sites such that each interaction is only counted once.

In two dimensions, the simplest Ising model looks like



where, for instance, the neighbours of S_{22} are S_{21}, S_{12}, S_{32}, S_{23}.

(a) Make a function that takes an $N \times N$ matrix $\mathbf{S}$ and calculates $\mathcal{H}$ assuming a periodic system.

In the lowest possible energy state all spins are aligned, i.e. they all equal 1 or -1, which corresponds to full magnetisation. However, fluctuations at finite temperatures will drive the system away from this minimum. The likelihood of a given spin configuration is given by the Boltzmann distribution

$$p(\mathbf{S}) = \frac{e^{-\mathcal{H}/T}}{Z}, \tag{5.7}$$

where T is the temperature and Z is the partition function. The magnetisation of a specific spin configuration is simply

$$m(\mathbf{S}) = \langle S_i \rangle. \tag{5.8}$$

We shall be interested in the time-averaged magnetisation as a function temperature T. We will use the Metropolis–Hastings algorithm to simulate the time evolution of the system.

(b) Make a function that calculates the change in energy ΔE from a spin flip at a given location i.
Speed hint: this calculation can be done without evaluating the full Hamiltonian twice, as the change in energy only depends on the local spins.

(c) Make a function that randomly accepts or rejects a spin flip based on ΔE with probability $\alpha = \min(1, e^{-\Delta E/T})$.

To take a Metropolis–Hastings step, a random random spin is chosen and flipped according to the above rule.

(d) Start with a random initialisation of $\mathbf{S}$ with $N \geq 5$ depending on the speed of your code ($N = 1000$ should be possible with `numba`). Let the system run for $1000\,N^2$ time steps to equilibrate the system. Plot the spin configuration (`plt.imshow` with `interpolation='nearest'`) at different times during the simulation. Use $T = 0.5$. Comment on the result.

(e) After initialisation let the system evolve another $1000\,N^2$ time steps, and store at each 100th time step the energy and the magnetisation. Do this for temperatures between 0.1 and 5.0 (e.g. `np.linspace(0.1, 5.0, 100)`) and plot the average absolute magnetisation and energy as a function of temperature.

You should find that the system spontaneously magnetises at low temperature. In particular, there is a *phase transition* at a critical temperature T_c below which the material is ferromagnetic.

(f) Determine T_c as best you can. What ways could you improve your estimate?

(g) *(optional)* We are only updating one spin per Monte Carlo step. Do you have ideas for updating many spins per step?

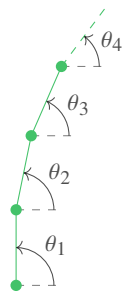
(h) *(optional)* A real magnetic has many, many spins ($\sim$ Avogadro's number). This basically corresponds to $N \to \infty$ (the thermodynamic limit). Is your estimate of T_c a good one for this? How would you find out?

Incredibly, this model has actually been solved analytically in the thermodynamic limit. The critical temperature is exactly $T_c = 2/\log(1+\sqrt{2})$.

5.6 Finite Temperature Euler Buckling

Note: For this exercise you will need to use `numba.njit` to make it run reasonably fast.

If you put enough weight on top of a spring at some point it will *buckle* meaning it will collapse under the weight. Here we will consider a simple model of a spring: a chain of $N = 15$ links that are connected by torsional springs. We will denote the angle that link i makes with the x-axis as θ_i as illustrated in the following figure:



We fix $\theta_1 = \pi/2$ and let the rest of the angles evolve freely.

For a "human-sized" spring, the temperature will not really matter for this problem: the spring forces are much larger than the thermal noise. However, if the spring instead is a DNA-string, thermal noise can definitely be felt. In this exercise we will consider the difference between a spring that is large (for which the temperature is small) and a small DNA-like spring (for which the temperature is large).

The energy of our spring consists of the energy of each torsional spring and the potential energy of the weight.

$$E = E_{\text{springs}} + E_{\text{weight}}. \tag{5.9}$$

In terms of the angles these are

$$E_{\text{springs}} = -k \sum_{i=2}^{N} \cos(\theta_i - \theta_{i-1}), \tag{5.10}$$

$$E_{\text{weight}} = w \sum_{i=1}^{N} \sin(\theta_i), \tag{5.11}$$

where k is the spring constant and w is the weight of the object. For this exercise we take $k = 1.0$.

(a) Explain the formulas for the energies.

We initialize all θ_i to $\pi/2$; i.e. all links start pointing directly up.

(b) Make a function that calculates E from an array of θ's.

To take a "Monte Carlo" step we select a random link i and change ('kick') the angle θ_i by a random amount selected from a standard normal distribution (`numpy.random.randn()`).

(c) Make a function that changes a random angle according to the above rule and calculates the change in energy ΔE from such a kick. *Remember*: we fix $\theta_1 = \pi/2$, so this link should never be chosen for a kick.

(d) Make a function that randomly accepts or rejects a kick based on ΔE with probability $\alpha = \min(1, e^{-\Delta E/T})$.

(e) We begin by considering the large spring for which $T = 0.001$. Let the system run for 1,000,000 Monte Carlo time steps to equilibrium and calculate the average height $H = \sum \sin\theta_i$ over the next 1,000,000 time steps using $w = 0.5$. *Note:* The "height" can be negative.

(f) Do the same as the above for w (logspaced) between 10^{-3} and 1.0 and plot the average height as a function of w (use `plt.semilogx`). Determine the critical load w_c after which the chain buckles.

(g) We now consider the DNA string: make the same plot for $T = 0.5$. Is the critical load still well defined?