Numerical Methods for Differential Equations

Homework no. 1 Submission

Joshua Belieu

Department of Physics and Astronomy
Michigan State University

August 24, 2025

1 AI-Assisted Symbolic Manipulation in Python

1.1 Taylor series $\rightarrow$ lambdify $\rightarrow$ plots (complex-valued).

1.1.1 Using Sympy

Using the numerical package, sympy, I took a Maclaurin series expansion (MSE) of the given expression at different orders, stripped the $\mathcal{O}(h^i)$ away from the generated expression, and converted the sympy expression into an executable function. See A.1 for the code snippet. This generated :

$$\begin{aligned} y_4 = {} & x^4\left(0.833333333333333 - 1.0\mathrm{i}\right) - x^3\left(1.33333333333333 + 0.666666666666667\mathrm{i}\right) \\ & + x^2\left(-1+\mathrm{i}\right) + x\left(1+\mathrm{i}\right) + 1 \end{aligned}$$

$$\begin{aligned} y_8 = {} & x^8\left(0.833730158730159 - 0.988888888888889\mathrm{i}\right) - x^7\left(1.2984126984127 + 0.634920634920635\mathrm{i}\right) \\ & + x^6\left(-0.833333333333333 + 0.988888888888889\mathrm{i}\right) + x^5\left(1.3 + 0.633333333333333\mathrm{i}\right) \\ & + x^4\left(0.833333333333333 - 1.0\mathrm{i}\right) - x^3\left(1.33333333333333 + 0.666666666666667\mathrm{i}\right) \\ & + x^2\left(-1+\mathrm{i}\right) + x\left(1+\mathrm{i}\right) + 1 \end{aligned}$$

$$\begin{aligned} y_{12} = {} & x^{12}\left(0.833730025118914 - 0.988897707231041\mathrm{i}\right) - x^{11}\left(1.29845759179093 + 0.634963924963925\mathrm{i}\right) \\ & + x^{10}\left(-0.833730158730159 + 0.988897707231041\mathrm{i}\right) + x^9\left(1.29845679012346 + 0.634964726631393\mathrm{i}\right) \\ & + x^8\left(0.833730158730159 - 0.988888888888889\mathrm{i}\right) - x^7\left(1.2984126984127 + 0.634920634920635\mathrm{i}\right) \\ & + x^6\left(-0.833333333333333 + 0.988888888888889\mathrm{i}\right) + x^5\left(1.3 + 0.633333333333333\mathrm{i}\right) \\ & + x^4\left(0.833333333333333 - 1.0\mathrm{i}\right) - x^3\left(1.33333333333333 + 0.666666666666667\mathrm{i}\right) \\ & + x^2\left(-1+\mathrm{i}\right) + x\left(1+\mathrm{i}\right) + 1 \end{aligned}$$

1.1.2 AI-Assist

I used ChatGPT 5 and it gave somewhat broad stroke steps in what it did. At first glance I was able to see that it was trying to model the function as a basis expansion. It took further prompting to get it to show the steps in its algebra. It was then I saw the difference in the approaches between (i) and (ii). After looking at its algebra for a little bit I could see that there were some discrepancies which it corrected (I assume by finding the next equation in whatever source it was looking at). All in all, I would not want to use it for these types of problems. Please go to A.2 for reference of use.

1.1.3 Verification

I asked ChatGPT to generate its expansions as a list of python strings so that I could easily loop through them and take the difference with sympy's results. Go to A.3 for code snippet. The results are presented here:

Order 4, Sympy - AI : 0
Order 8, Sympy - AI : $-9.02276489854095e - 17 * I * x * *3$
Order 12, Sympy - AI : $1.24435168432403e - 16 * I * x * *3 * (x * *6 - 1)$
Where this is raw sympy output and I represents the complex unit. These small values are consequences of numerical instabilities and are, in effective magnitude, zero.

1.1.4 Plots

See A.1 for plotting routine.

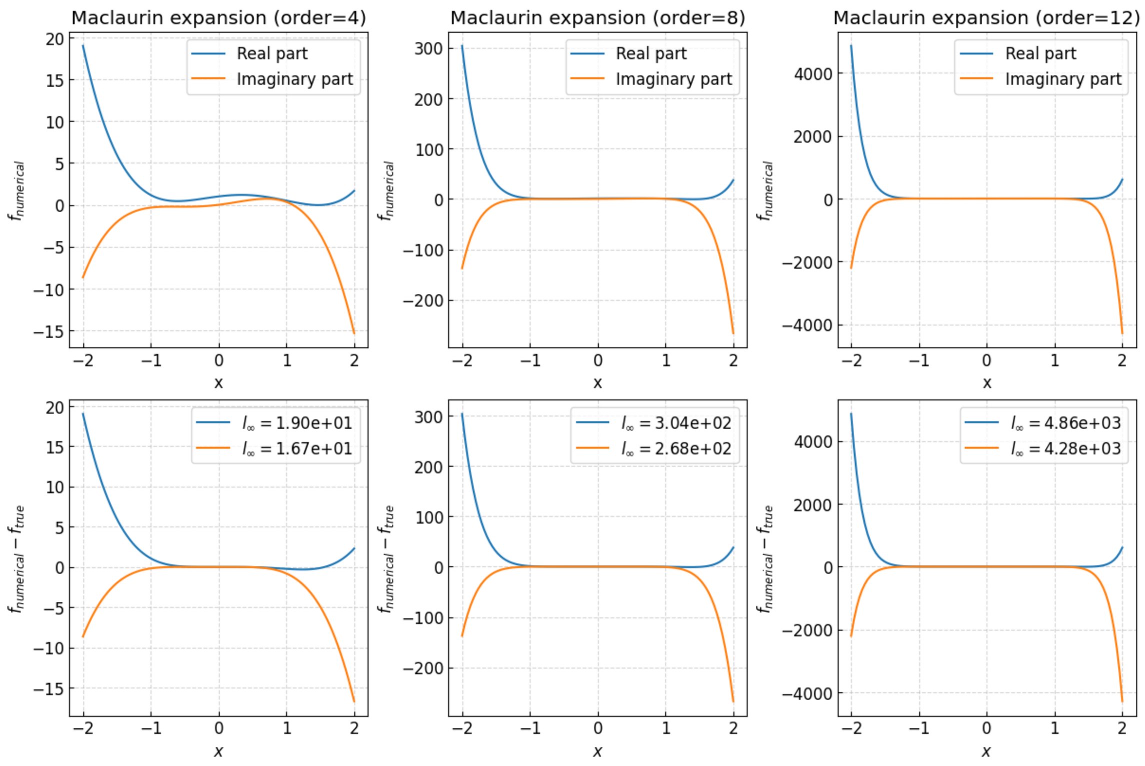

Figure 1: 1.1.4 plots with error.

The plots found in figure 1 feature the numerical expression in the top row. The bottom row shows the error where I have labeled the colored curves with their l_∞. The columns correspond to the order of the expansion.

1.2 Method of moments

1.2.1 Vandermonde system

We would like to do:

$$hf'(x_i) = \sum_{i=-3}^{1} c_{i+3} f(x_i + ih) = \sum_{i=-3}^{1} d_{i+3} f(x_i + ih) = d_0 f(x_i - 3h) + d_1 f(x_i - 2h) + d_2 f(x_i - h)$$

$$+ d_3 f(x_i) + d_4 f(x_i + h)$$

Where I can Taylor expand each function about the point x_i,

$$f(x_i \pm ih) \approx f(x_i) \pm ihf'(x_i) + \frac{(ih)^2}{2} f''(x_i) \pm \frac{(ih)^3}{6} f'''(x_i) + \frac{(ih)^4}{24} f^{(4)}(x_i) + \mathcal{O}(h^5)$$

Evaluating this for all functions in our first expression and grouping by order of derivative,

$$\begin{aligned}
hf'(x_i) =& d_0(f(x_i) - 3hf'(x_i) + \frac{(3h)^2}{2}f''(x_i) - \frac{(3h)^3}{6}f'''(x_i) + \frac{(3h)^4}{24}f^{(4)}(x_i))+ \\
& d_1(f(x_i) - 2hf'(x_i) + \frac{(2h)^2}{2}f''(x_i) - \frac{(2h)^3}{6}f'''(x_i) + \frac{(2h)^4}{24}f^{(4)}(x_i))+ \\
& d_2(f(x_i) - hf'(x_i) + \frac{(h)^2}{2}f''(x_i) - \frac{(h)^3}{6}f'''(x_i) + \frac{(h)^4}{24}f^{(4)}(x_i)) \\
& + d_3 f(x_i)+ \\
& d_4(f(x_i) + hf'(x_i) + \frac{(h)^2}{2}f''(x_i) \pm \frac{(h)^3}{6}f'''(x_i) + \frac{(h)^4}{24}f^{(4)}(x_i)) \\
& + \mathcal{O}(h^5)
\end{aligned}$$

$$hf'(x_i) \approx (d_0+d_1+d_2+d_3+d_4)f(x_i)+h(-3d_0-2d_1-d_2+d_4)f'(0)+\frac{h^2}{2}(9d_0+4d_1+d_2+d_4)f''(x_i)$$

$$+\frac{h^3}{6}(-27d_0 - 8d_1 - d_2 + d_4)f'''(x_i) + \frac{h^4}{24}(81d_0 + 16d_1 d_2 + d_4)f^{(4)}(x_i) + \mathcal{O}(h^5)$$

This defines a set of linear equations that we can solve to find a vector, $\vec{d}$. I will do this using Sympy where I have augmented some code provided by the professor (see A.3). This produces the vector:

$$\vec{d} = \begin{bmatrix} -\frac{1}{12} & \frac{1}{2} & \frac{-3}{2} & \frac{5}{6} & \frac{1}{4} \end{bmatrix}$$

1.2.2 AI-Assist

First, ChatGPT wanted to generate code for me to run which I corrected. It quickly wrote the Vandermonde system and gave quick explanations what the Vandermonde matrix represented. It did not show me how it solved the system but it did give me the same coefficients that I found. Then, it gave the moments and showed the equation that it was calculating along with the result. Finally, it adhered to the rational instruction. See A.2 for engagement.

1.2.3 Verify all moments and prove order 4

For each moment, I calculate the inner product of the weights and offsets raised to an order. See A.5 for code snippet. The results are found in Table 1 where we can see that the first non-zero moment, aside from the first-order derivative we want, is the fifth one. This demonstrates that the method is on order four.

Moment	0	1	2	3	4	5
$d_j s_j^{moment}$	0	1	0	0	0	6

Table 1: The moments of the system. Einstein notation is used for the label of row 1 column 0.

1.2.4 Numerical check

This is my idea; we have learned in class and in the text that when we do our approximations our leading error will have some power of h multiplied by a constant C (representing an order of derivative at some point). For example, recall from the text that,

$$D_0D_+D_-u(\bar{x}) = u'''(\bar{x}) + \frac{h^2}{4}u''''(\bar{x}) + \mathcal{O}(h^4)$$

If we define our error as the absolute difference in the left hand side (LHS) term and the first term on the right (also, I will supress the omicron from here) then we see,

$$E \equiv |D_0D_+D_-u(\bar{x}) - u'''(\bar{x})| = \frac{h^2}{4}|u''''(\bar{X})|$$

We see that our leading order 'remnant' depends on h^2. This means that if I systematically divide the error by powers of h and plot the behavior of the error, then when I divide by the power of h^2 this term will be constant. Applying this to our system results in figure 2 where we can see that dividing by h^4 is constant over the refinement levels. That is, the order is fourth. For reported errors, see Table **??**. See A.6 for code snippet.

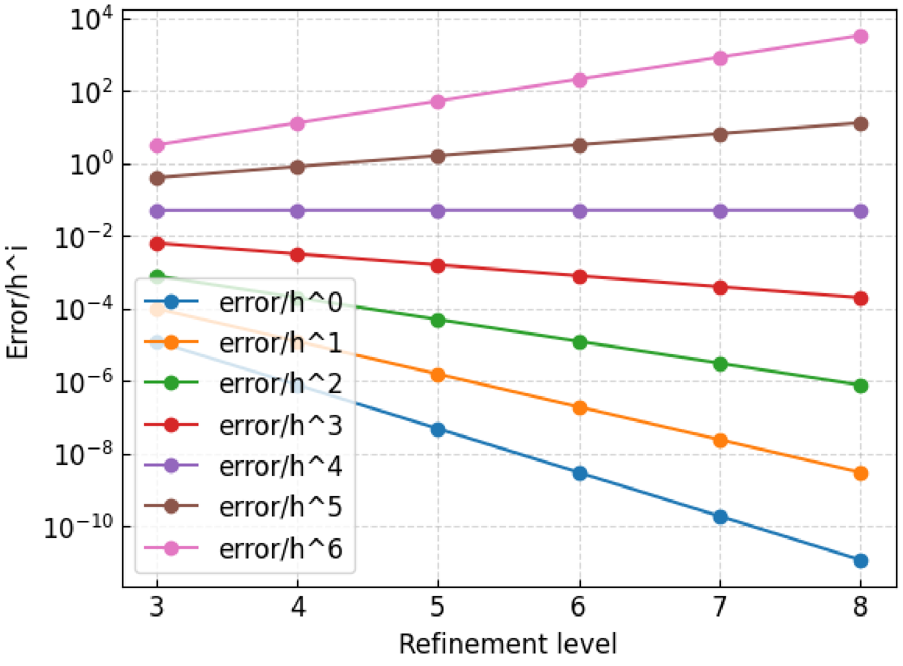

Figure 2: Systematically dividing the error by powers of h to determine the order of the method.

1.3 One-sided 6-point second derivative at a right boundary

1.3.1 6 by 6 moment system

The work here is much the same as in 1.2.1. A point of notation, my point of interest will be designated $\bar{x}$ and the function at that point will be displayed $f(\bar{x}) \equiv \bar{f}$.

idx	$h = 2^{-\text{idx}}$	approximation	error	error$/h^4$
03	1.250×10^{-1}	1.00001211643246	1.212×10^{-5}	4.9629e-2
04	6.250×10^{-2}	1.00000076152118	7.615×10^{-7}	4.9907e-2
05	3.125×10^{-2}	1.00000004766154	4.766×10^{-8}	4.9977e-2
06	1.562×10^{-2}	1.00000000297989	2.980×10^{-9}	4.9994e-2
07	7.812×10^{-3}	1.00000000018626	1.863×10^{-10}	4.9999e-2
08	3.906×10^{-3}	1.00000000001164	1.164×10^{-11}	4.9999e-2

Table 2: Approximation errors for different step sizes h.

$$h^2 \bar{f}'' = \sum_{i=-5}^{0} c_{i+5} f(\bar{x} + ih) = \sum_{i=-5}^{0} d_{i+5} f(\bar{x} + ih) = d_0 f(\bar{x} - 5h) + d_1 f(\bar{x} - 4h) + d_2 f(\bar{x} - 3h)$$
$$+ d_3 f(\bar{x} - 2h) + d_4 f(\bar{x} - h) + d_5 \bar{f}$$

Where I can Taylor expand each function about the point $\bar{x}$,

$$f(\bar{x} \pm ih) \approx \bar{f} \pm ih\bar{f}'(\bar{x}) + \frac{(ih)^2}{2}\bar{f}''(\bar{x}) \pm \frac{(ih)^3}{6}\bar{f}'''(\bar{x}) + \frac{(ih)^4}{24}\bar{f}^{(4)}(\bar{x}) \pm \frac{(ih)^5}{120}\bar{f}^{(5)}(\bar{x}) + \mathcal{O}(h^6)$$

Using this relation to sub in the relevant expressions and group by order of derivative results in the following.

$$\bar{f}'' = (d_0 + d_1 + d_2 + d_3 + d_4 + d_5)\bar{f} - h(d_0 + d_1 + d_2 + d_3 + d_4)\bar{f}' + \frac{h^2}{2}(25d_0 + 16d_1 + 9d_2 + 4d_3 + d_4)\bar{f}''$$
$$- \frac{h^3}{6}(125d_0 + 64d_1 + 27d_2 + 8d_3 + d_4)\bar{f}''' + \frac{h^4}{24}(625d_0 + 256d_1 + 81d_2 + 16d_3 + d_4)\bar{f}^{(4)}$$
$$- \frac{h^5}{120}(3125d_0 + 1024d_1 + 243d_2 + 32d_3 + d_4)\bar{f}^{(5)}$$

I will make use of the same code as described in 1.2.1 (A.4) with the results:

$$c_j = h^2 d_j \rightarrow \vec{c} = h^2 \vec{d} = h^2 \begin{bmatrix} -\frac{5}{6} \\ \frac{61}{12} \\ -13 \\ \frac{107}{6} \\ -\frac{77}{6} \\ \frac{15}{4} \end{bmatrix}$$

1.3.2 AI-Assist

I am writing this after the initial use of the LLM and I see now that the coefficients it generated do not match mine entirely. I do not know how it deviated in such a way. I copy and pasted the prompt from the homework as I had in the previous AI-Assist problems. In any case, it made good point that some rationals are non-terminating or can be so large that there decimal representation is truncated in the computer's memory. This truncation causes further error aside from the error caused by our approximation of the system. It ended with advising me to stick with rational fractions so that there is minimized truncation in the evaluation of the weights. See A.2 for engagement.

1.3.3 Verify moment conditions symbolically; state the formal order

See A.7 for code snippet which resulted in the moment vector $\vec{M}$:

$$\vec{M} \doteq \begin{bmatrix} 0 \\ 0 \\ 2 \\ 0 \\ 0 \\ 0 \\ -548 \end{bmatrix}$$

From this we can see that $q = N - m = 6 - 2 = 4$.

1.3.4 Test with an exponential near a right boundary; confirm order

Here I will employ a similar thought process as 1.2.4 and will re-tune the code snippet at A.6. I will methodically decrease the step size and test which division order is seemingly constant. My expansion point will be at $x = 0$. From figure 3 we see dividing by h^4 results in a near zero slope. Therefore, the order is four.

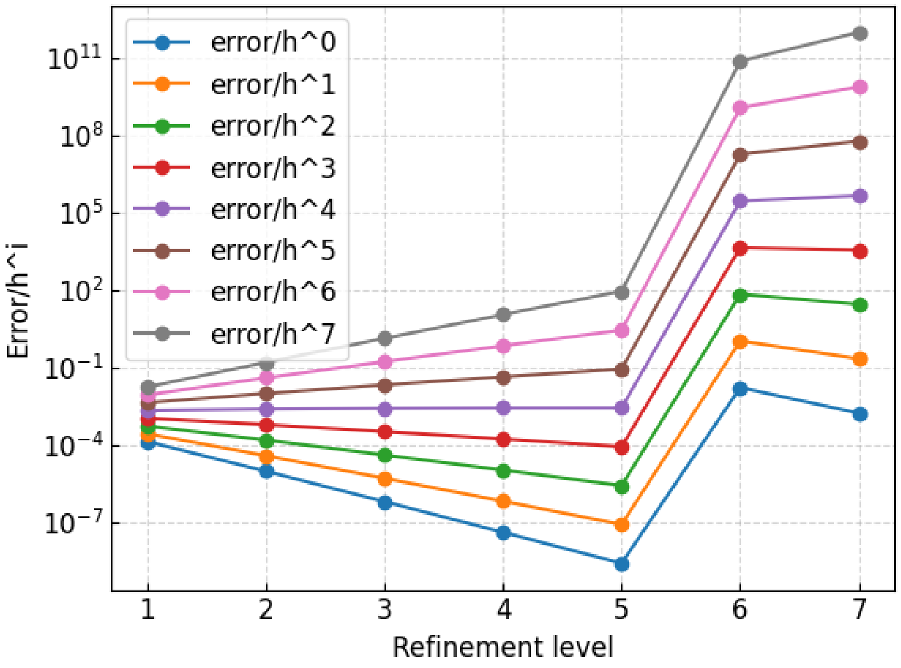

Figure 3: Systematically dividing the error by powers of h to determine the order of the method.

2 Non-Uniform Grid

2.1 Second derivative FDM with dominant error

We begin with our definition but with a slight adjustment. Due to the fact this method is written with a non-uniform grid in mind we must omit our convenient $\frac{1}{h^d}$ term in our ansatz.

$$u''(x_2) = \sum_{i=1}^{4} c_i U_i = c_1 U_1 + c_2 U_2 + c_3 U_3 + c_4 U_4 + = c_1 u(x_1) + c_2 u(x_2) + c_3 u(x_3) + c_4 u(x_4)$$
$$= c_1 u(x_2 - h_1) + c_2 u(x_2) + c_3 u(x_2 + h_2) + c_4 u(x_2 + h_2 + h_3)$$

Next we Taylor expand each function about the point $x = x_2$ like we have done on previous problems and I will group them by orders of the derivative.

$$u''(x_2) \approx (c_1+c_2+c_3+c_4)u(x_2)+(-h_1c_1+h_2c_3+(h_2+h_3)c_4)u'(x_2)+\frac{1}{2}(h_1^2c_1+h_2^2c_3+(h_2+h_3)^2c_4)u''(x_2)$$
$$+\frac{1}{6}(-h_1^3c_1 + h_2^3c_3 + (h_2 + h_3)^3c_4)u'''(x_2) + \frac{1}{24}(h_1^4c_1u''''(\xi_1) + h_2^4c_3u''''(\xi_3) + (h_3 + h_3)^4c_4u''''(\xi_4)$$

If I collect terms on the RHS up to third order in the derivative of $u(x_2)$ and refer to them as $\bar{u}$ then I see the following,

$$E \equiv |u''(x_2) - \bar{u}| = \left| \frac{1}{24}(h_1^4c_1u''''(\xi_1) + h_2^4c_3u''''(\xi_3) + (h_2 + h_3)^4c_4u''''(\xi_4) \right|$$

Although the step sizes are all different if we claim that the three step sizes are locally clustered around their average, $\bar{h} \equiv \frac{1}{N}\sum_N h_n$, then we can see,

$$E = \frac{\bar{h}^4}{24}\left|c_1c_1u''''(\xi_1) + c_3u''''(\xi_3) + 2c_4u''''(\xi_4)\right| \equiv \frac{\bar{h}^4}{24}C$$

2.2 Error behavior with finer grids

Following the directions of the problem I wrote the code found in B.1. This methodology generated figure 4. As we can see there is a strong correlation in the error and increasing step size.

2.3 Order of accuracy fit

Following the directions again resulted in the code snippet 5 which generated figure B.2. We see here a reliable fit to the data. The values of the parameters as a result of the least squares solution are:

$$K = -3.8031127737479125, \quad p = 1.973644440179718$$

3 Mixed Boundary Conditions

Consider the system:

$$u'' + u = f(x) \quad \forall x \in [0, 10]$$

With boundary conditions :

$$u'(0) - u(0) = 0, \quad u'(10) + u(10) = 0$$

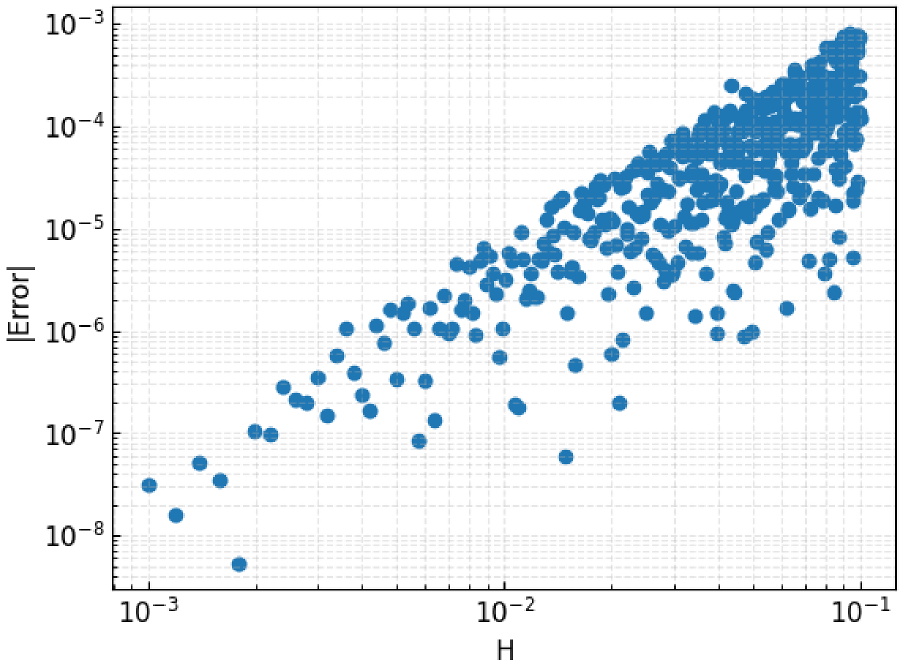

Figure 4: The max error from non-uniform step sizes on a log-log plot.

3.1 2nd order FDM as a linear system

I will begin by discretizing the O.D.E. such that there are n divisions from 0 and 10 $(n+1)$ and use a central difference method to represent the second derivative.

$$u'' + u = f(x) \Rightarrow D^2 u_i + u_i = \frac{1}{h^2}\left(u_{i-1} - 2u_i + u_{i+1}\right) + u_i = \frac{1}{h^2}\left(u_{i-1} + (-2+h^2)u_i + u_{i+1}\right) = f_i$$

Where now I can see this is a set of linear equations which can be represented by a set of matrices. Before I cast the ODE into matrix form I will discuss the treatment of the boundary values. Now, because this is meant to be second order accurate we must choose between a central difference or one sided difference with an additional point in the stencil in order to reach the desired order. For no other reason other than I was having difficulty with the central difference, I will do a one sided stencil. Beginning with the boundary at $x = 0$:

$$u'(x_0) = \frac{1}{h} c_j u(x_0 + jh) = \frac{1}{h}(c_0 u(x_0) + c_1 u(x_0 + h) + c_2 u(x_0 + 2h))$$

Doing a Taylor series expansion and grouping by order of derivative in u like we did earlier in the homework leaves us with,

$$hu'(x_0) \approx (c_0 + c_1 + c_2)u_0 + h(c_1 + c_2)u_0' + \frac{h^2}{2}(c_1 + c_2)u_0'' + \mathcal{O}(h^3)$$

Which represented as a matrix system,

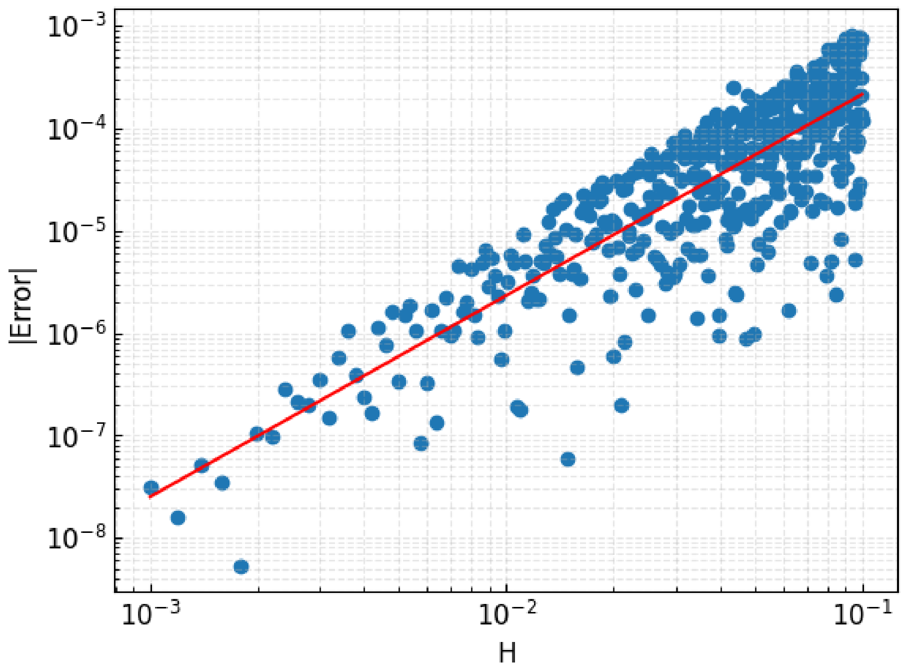

Figure 5: The linear relationship found via a least squares fit to the log of the error and step size.

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 2 \\ 0 & 1 & 4 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Now, if we invert the coefficient matrix we will have our weights. Since we have done this a number of times already (and I have hopefully demonstrated my familiarity with the process and algebra) I will elect to have WolframAlpha solve this system. This results in the values $c_0 = -\frac{3}{2}, c_2 = 2, c_3 = -\frac{1}{2}$. So, our boundary condition at $x = 0$ will be of form,

$$u'(0) - u(0) \doteq u'(x_0) - u(x_0) \equiv u'_0 - u_0 \approx \frac{1}{h}(-\frac{3}{2}u_0 + 2u_1 - \frac{1}{2}u_2) - u_0 = 0$$

This expression encodes the behavior of the system at the boundary and will take the place of the top row in our eventual matrix representation of the system. Moving to the other boundary,

$$u'(x_{n+1}) = \frac{1}{h}c_j u(x_{n+1} - jh) = \frac{1}{h}(c_0 u(x_{n+1}) + c_1 u(x_{n+1} - h) + c_2 u(x_{n+1} - 2h)$$

$$hu'(x_{n+1}) \approx (c_0 + c_2 + c_3)u_{n+1} - h(c_1 - 2c_2)u'_{n+1} + \frac{h^2}{2}(c_1 + 4c_2)u''_{n+1}$$

I will do the exact same thing at the other boundary but I will not show the setup here. The important thing here is that we see that this system will differ from the other boundary by a minus sign in the row corresponding to the first derivative. Using WolframAlpha this results in the coefficients $c_0 = \frac{3}{2}, c_1 = -2, c_2 = \frac{1}{2}$. So, our last row which details the behavior at the boundary will appear,

$$u'(10) + u(10) \doteq u'(x_{n+1}) + u(x_{n+1}) \approx \frac{1}{h}(\frac{3}{2}u_{n+1} - 2u_n + \frac{1}{2}u_{n-1})$$

Ok, with these two adjustments we are ready to represent our system of equations as matrices.

$$\frac{1}{h^2}\begin{bmatrix} -\frac{3}{2h}-1 & \frac{2}{h} & -\frac{1}{2} & 0 & \cdots & & & 0 \\ 1 & -2+h^2 & 1 & 0 & \cdots & & 0 & \\ 0 & 1 & -2+h^2 & 1 & 0 & \cdots & 0 & \\ \vdots & & \ddots & & & & \vdots & \\ \vdots & & \cdots & & 1 & -2+h^2 & 1 & \\ 0 & & \cdots & 0 & \frac{1}{2h} & -\frac{2}{h} & \frac{3}{2h}+1 & \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_n \\ u_{n+1} \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_n \\ f_{n+1} \end{bmatrix}$$

3.2 L_∞ stability

With our ODE system expressed as a derivative functional, $\mathcal{L}[\phi] = \phi'' + \phi$ then, $\exists G(x;\bar{x}) \ni \mathcal{L}[G] = G'' + G = \delta(x-\bar{x})$. Which enables us to state,

$$\mathcal{L}[G] = G'' + G = 0 \quad \forall x \neq \bar{x}$$

This system has a familiar ansatz. Let $G = e^{\lambda x} \ni G'' = \lambda^2 e^{\lambda x}$ and evaluate it into the ODE to see,

$$\lambda^2 e^{\lambda x} + e^{\lambda x} = e^{\lambda x}(\lambda^2 + 1) = 0 \quad \forall x \neq \bar{x}$$

Due to the exponential being non-zero on a finite domain we see that this expression is satisfied if $\lambda = \pm i$. This tells us our system, away from $\bar{x}$ is of form,

$$G = e^{\pm ix} = iA' \sin(\pm x) + B' \cos(\pm x) = \pm iA' \sin x + B' \cos x \equiv A \sin x + B \cos x$$

Due to the delta function I will split this up into two regions,

$$G(x;\bar{x}) = \begin{cases} A \sin x + B \cos x & \forall x < \bar{x} \\ C \sin x + D \cos x & \forall x > \bar{x} \end{cases}$$

Applying our boundary conditions,

$$G(0) - G'(0) = A \sin 0 + B \cos 0 - A \cos 0 + B \sin 0 = 0 \quad \Rightarrow A = B$$

$$G(10) + G'(10) = C \sin 10 + D \cos 10 + C \cos 10 - D \sin 10 = C(\sin 10 + \cos 10) + D(-\sin 10 + \cos 10) = 0$$

$$\Rightarrow C = \gamma D, \quad \gamma \equiv \frac{-\cos 10 + \sin 10}{\cos 10 + \sin 10}$$

Next, we need to ensure continuity at the delta function,

$$G^+(\bar{x}) = G^-(\bar{x}) \to A(\sin \bar{x} + \cos \bar{x}) = D(\gamma \sin \bar{x} + \cos \bar{x})$$

To obtain the final necessary expression needed to determine these coefficients we must go back to the ODE,

$$G'' + G = \delta(x - \bar{x}) \rightarrow \int_{\bar{x}-\epsilon}^{\bar{x}+\epsilon} dx G'' + G = \int_{\bar{x}-\epsilon}^{\bar{x}+\epsilon} dx \delta(x - \bar{x}) \quad \forall \varepsilon << 1$$

From here, I apply the linearity of the integration operation to separate the two integrands on the LHS, apply the integral form of the fundamental theorem of calculus to the second derivative term, and note that the integral of a delta function over an interval that contains it's kernel is 1. This leaves us,

$$G'\big|_{\bar{x}+\epsilon} - G'\big|_{\bar{x}-\epsilon} + \int_{\bar{x}-\epsilon}^{\bar{x}+\epsilon} dx G = 1$$

Now I take the limit that $\epsilon \rightarrow 0$ and we see that the remaining integral is now bounded by the same value and equal to zero and we are left with,

$$G'^{+}(\bar{x}) - G'^{-}(\bar{x}) = A(\cos\bar{x} - \sin\bar{x}) + D(-\gamma\cos\bar{x} + \sin\bar{x}) = 1$$

Solving this using a sympy code I wrote up (C.2),

$$\begin{aligned} A = & \frac{\sin(\bar{x})\cos(10)}{2\sin(10)\sin^2(\bar{x}) + 4\sin(\bar{x})\cos(10)\cos(\bar{x}) - 2\sin(10)\cos^2(\bar{x})} \\ & - \frac{\sin(10)\sin(\bar{x})}{2\sin(10)\sin^2(\bar{x}) + 4\sin(\bar{x})\cos(10)\cos(\bar{x}) - 2\sin(10)\cos^2(\bar{x})} \\ & - \frac{\sin(10)\cos(\bar{x})}{2\sin(10)\sin^2(\bar{x}) + 4\sin(\bar{x})\cos(10)\cos(\bar{x}) - 2\sin(10)\cos^2(\bar{x})} \\ & - \frac{\cos(10)\cos(\bar{x})}{2\sin(10)\sin^2(\bar{x}) + 4\sin(\bar{x})\cos(10)\cos(\bar{x}) - 2\sin(10)\cos^2(\bar{x})} \end{aligned}$$

$$\begin{aligned} D = & - \frac{\sin(10)\sin(\bar{x})}{2\sin(10)\sin^2(\bar{x}) + 4\sin(\bar{x})\cos(10)\cos(\bar{x}) - 2\sin(10)\cos^2(\bar{x})} \\ & - \frac{\sin(\bar{x})\cos(10)}{2\sin(10)\sin^2(\bar{x}) + 4\sin(\bar{x})\cos(10)\cos(\bar{x}) - 2\sin(10)\cos^2(\bar{x})} \\ & - \frac{\sin(10)\cos(\bar{x})}{2\sin(10)\sin^2(\bar{x}) + 4\sin(\bar{x})\cos(10)\cos(\bar{x}) - 2\sin(10)\cos^2(\bar{x})} \\ & - \frac{\cos(10)\cos(\bar{x})}{2\sin(10)\sin^2(\bar{x}) + 4\sin(\bar{x})\cos(10)\cos(\bar{x}) - 2\sin(10)\cos^2(\bar{x})} \end{aligned}$$

With the Green's function determined I will note that we use it to obtain the eigenfunction of the system via

$$u(x) = \int G(x;\bar{x}) f(\bar{x}) d\bar{x}$$

Which, for a finite spacing, h, will appear as

$$u(x_i) = \sum_j hG(x_i;x_j)f(x_j) \doteq \vec{u} = B\vec{f}$$

Where I have grouped the lattice spacing and Green's function matrix representation into the matrix B. Comparing this to our original system $A\vec{u} = \vec{f} \Rightarrow \vec{u} = A^{-1}\vec{f}$ we see that $A^{-1} = B = hG$. With this we can see in the max norm in the limit that h goes to zero that,

$$\lim_{h\to 0} ||A^{-1}||_\infty = \lim_{h\to 0} ||hG||_\infty = \lim_{h\to 0} \left\{ \max_{1\leq i\leq N+1} \sum_i |hG_i| \right\} = \max \int |G| d\bar{x} \equiv C$$

Note that G_i represents the ith row of G. Taking the limit as h goes to zero causes this discrete sum to take the form of an integral. We know there exists some finite value due to our work incorporating the boundary conditions and continuity in our Green's function. Therefore, $||A^{-1}||_\infty$ is bounded on the interval.

3.3 Exact solution

Our system now looks like,

$$u'' + u = -e^x$$

My ansatz will be a sum of a homogenous and particular solution, $u = u_h + u_p$, where the homogenous solution satisfies $u'' + u = 0$. Thankfully our work with the Green's function illuminated us to the ansatz,

$$u_h = A\sin x + B\cos x$$

Now, the particular solution must address the function on the RHS so I will first guess that the function itself will suffice,

$$u_p = Ce^x$$

and so,

$$u = A\sin x + B\cos x + Ce^x \quad u' = A\cos x - B\sin x + Ce^x \quad u'' = -(A\sin x + B\cos x) + Ce^x$$

Evaluating these into the ODE reveals,

$$2Ce^x = -e^x \Rightarrow C = -\frac{1}{2}$$

With this out of the way we need to use our boundary values to determine the last remaining set of coefficients. Beginning at $x = 0$:

$$A\cos 0 - B\sin 0 - \frac{1}{2}e^0 - A\sin 0 - B\cos 0 + \frac{1}{2}e^0 = 0 \implies A = B, \quad \text{at } x = 10,$$

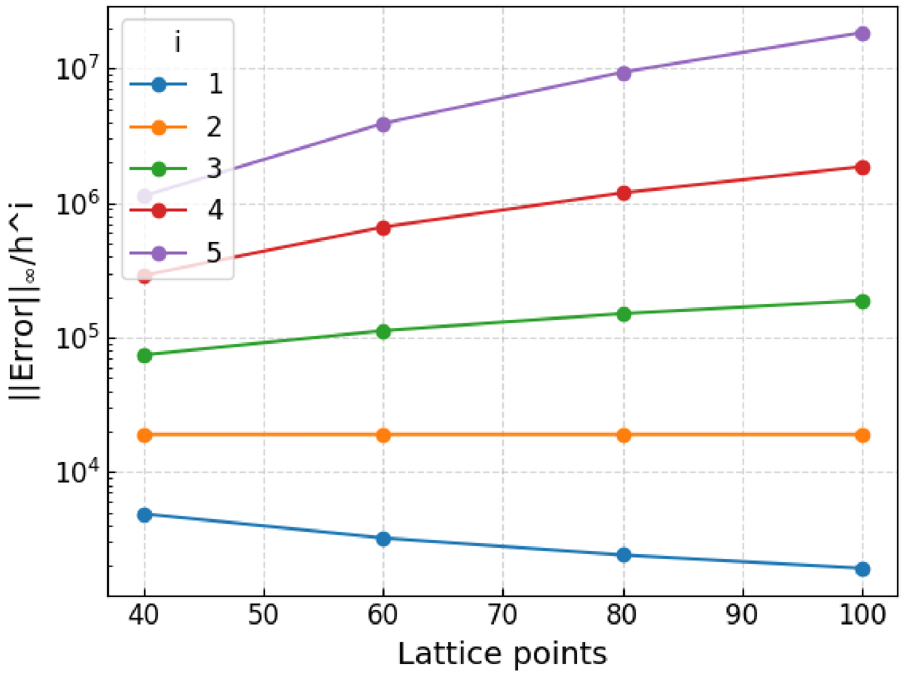

Figure 6: The max error at different lattice points (step sizes) divided by step size to some power.

$$A(\cos 10 - \sin 10) - \frac{1}{2}e^{10} + A(\sin 10 + \cos 10) - \frac{1}{2}e^{10} = 2A\cos 10 - e^{10} = 0$$

$$\Rightarrow A = \frac{e^{10}}{2\cos 10} = \frac{1}{2}\sec 10e^{10} \quad \text{and lo,}$$

$$u(x) = \frac{1}{2}\sec 10e^{10}(\sin x + \cos x) - \frac{1}{2}e^{x}$$

3.4 2nd order accuracy verification

This problem was implemented with a different scheme that distinguishes itself from prior code snippets, see C.1. We see that the curve with slope nearest 0 is the one that is being divided by h^2. See figure 6.

4 Variable diffusivity

4.1 Show that A appearing in (2.72) of LeVeque's notes is negative definite for $\kappa > 0$

Assumption: $\kappa(x) \in \mathbb{R} > 0$.

Inspecting the matrix, A, I see that it is a real symmetric tri-diagonal matrix and therefore has the following properties:

1. $A_{i,j} = 0 \quad \forall |i - j| > 1$

2. $A_{i,i+1} = A_{i+1,i} > 0$
3. $A_{i,i} = -(A_{i+1,i} + A_{i-1,i}) < 0 \quad \forall i \neq 0, d$

Then, if A has dimension d and $\forall \vec{U} \neq \vec{0}$ we can see,

$$U^T AU = U^T(AU) = \sum_{j=0}^{d} U^T A_{i,j} U_j = \sum_{i=0}^{d}\sum_{j=0}^{d} U_i A_{i,j} U_j = \sum_{i=0}^{d} U_i A_{i,i} U_i + \sum_{i=0}^{d-1} U_i A_{i,i+1} U_{i+1} + \sum_{i=0}^{d-1} U_{i+1} A_{i+1,i} U_i$$

$$= \sum_{i=0}^{d} U_i^2 A_{i,i} + 2\sum_{i=0}^{d-1} U_i U_{i+1} A_{i,i+1} = U_d^2 A_{d,d} + \sum_{i=0}^{d-1} U_i^2 A_{i,i} + 2U_i U_{i+1} A_{i,i+1}$$

$$= U_d^2 A_{d,d} - \sum_{i=0}^{d-1} U_i^2 (A_{i-1,i} + A_{i+1,i}) - 2U_i U_{i+1} A_{i,i+1} = U_d^2 A_{d,d} - \sum_{i=0}^{d-1} U_i^2 (A_{i,i-1} + A_{i,i+1}) - 2U_i U_{i+1} A_{i,i+1}$$

$$\equiv \gamma - \sum_{i=0}^{d-1} (U_i^2 - 2U_i U_{i+1}) A_{i,i+1}, \quad \gamma \equiv U_d^2 A_{d,d} - \sum_{i=0}^{d-1} U_i^2 A_{i,i-1}$$

Where after the fourth equality I made use of property 1 and split the sum into a sum of main diagonal elements, supra-diagonal elements, and sub-diagonal elements, respectively. Then, I used property two to collect the sub- and supra- diagonal sums into an "off-diagonal" sum. Then I expanded one term in the main diagonal sum so that both sums would have the same index. Making use of property 3 I then assocaited the common A element terms and defined a variable for ease of writing (γ). Now, I will complete the square in the explicit sum:

$$= \gamma - \sum_{i=0}^{d-1} (U_i^2 - 2U_i U_{i+1} \pm U_{i+1}) A_{i,i+1} = \gamma - \sum_{i=0}^{d-1} ((U_i - U_{i+1})^2 - U_{i+1}^2) A_{i,i+1} = \gamma - \sum_{i=0}^{d-1} (U_i - U_{i+1})^2 A_{i,i+1} + \sum_{i=0}^{d-1} U_{i+1}^2 A_{i,i+1}$$

I have separated that last sum becuase I will re-index via $j = i + 1$ resulting in,

$$\sum_{i=0}^{d-1} U_{i+1}^2 A_{i,i+1} = \sum_{j=1}^{d} U_j^2 A_{j-1,j} = \sum_{j=1}^{d} U_j^2 A_{j,j-1} \equiv \sum_{i=1}^{d} U_i^2 A_{i,i-1}$$

Notice that I relabeled j back to i in the last equality as it is just a dummy variable. Putting this back into our relation, re-subbing γ and moving some terms around we see,

$$U_d^2 A_{d,d} - \sum_{i=0}^{d-1} (U_i - U_{i+1})^2 A_{i,i+1} - \sum_{i=0}^{d-1} U_i^2 A_{i,i-1} + \sum_{i=1}^{d} U_i^2 A_{i,i-1}$$

We see an almost perfect cancellation of terms in the last two sums. Leaving only what remains,

$$U_d^2 A_{d,d} - \sum_{i=0}^{d-1} (U_i - U_{i+1})^2 A_{i,i+1} - U_0^2 A_{0,1} + U_d^2 A_{d,d-1}$$

Finally, I look back at matrix A in Levque's notes and I see that on the last row we can express $A_{d,d} = -(A_{d,d-1} + \kappa_d)$. Where κ_d corresponds to the $\kappa_{m+1/2}$ in Leveque's notes. In this form we can see the cancelation in the last term and we are left with,

$$U^T AU = -\sum_{i=0}^{d-1}(U_i - U_{i+1})^2 A_{i,i+1} - U_0^2 A_{0,1} - U_d^2 \kappa_d$$

As we see, all vector components are square and therefore positive, the off-diagonal elements are given as positive, and the κ_d term is given as positive. Thus, with the negative signs we can say $U^T AU < 0$.

4.2 Show that (2.50) of LeVeque's notes satisfies a maximum principle in the homogenous case

Following Leveque's lead in approximating the product $\kappa(x)u'(x)$ at points halfway between the grid points,

$$\kappa(x_{i+1/2})u'(x_{i+1/2}) = \kappa(x_{i+1/2})\frac{(u_{i+1} - u_i)}{h}, \kappa(x_{i-1/2})u'(x_{i-1/2}) = \kappa(x_{i-1/2})\frac{(u_i - u_{i-1})}{h}$$

We then form a central difference by taking the difference of these two expressions,

$$(\kappa u')'(x_i) \approx \frac{1}{h}\left[\kappa_{i+1/2}\frac{(u_{i+1} - u_i)}{h} - \kappa_{i-1/2}\frac{(u_i - u_{i-1})}{h}\right] = \frac{1}{h^2}\left[\kappa_{i-1/2}u_{i-1} - (\kappa_{i-1/2} + \kappa_{i+1/2})u_i + \kappa_{i+1/2}u_{i+1}\right] = 0$$

Shifting things around and solving for u_i I see,

$$u_i = \frac{\kappa_{i-1/2}}{\kappa_{i-1/2} + \kappa_{i+1/2}}u_{i-1} + \frac{\kappa_{i+1/2}}{\kappa_{i-1/2} + \kappa_{i+1/2}}u_{i+1}$$

If we imagine that $u_{i\pm1} = 1$ then we see,

$$u_i = \frac{\kappa_{i-1/2}}{\kappa_{i-1/2} + \kappa_{i+1/2}} + \frac{\kappa_{i+1/2}}{\kappa_{i-1/2} + \kappa_{i+1/2}} = 1$$

We see now that u_i is a weighted sum of values surrounding that point where the weights are determined by the normalized values of the surrounding heat conduction coefficient values. With this in mind we could consider these weights as a single parameter, $k \in [0, 1]$, where the system now appears like,

$$u_i = ku_{i-1} + (1 - k)u_{i+1}$$

This latest relation tells us that u_i cannot be larger than the sum of the points around it and it is not guaranteed to be larger than any one surrounding point. Due to the fact that u_i depends on values to the left and right, this only applies to values on the interval (a, b). Picking any arbitrary value on the open interval therefore has some traceable maximum and minimum back to the border values. If we then "trace back" the values for any points on the interior in search of extrema we will eventually find that the extrema are characterized by,

$$\{\min\{u(a), u(b)\}, \quad \max\{u(a), u(b)\}\}$$

5 Interior & boundary closures, assembly, and solve.

On $\Omega = [0,1][0,1]$, solve

$$-\Delta u = f(x,y), f(x,y) = \begin{cases} 1, & x \in [1/3, 1/2], y \in [1/2, 2/3] \\ 0, & \text{otherwise} \end{cases}$$

Moving forward I will use H and L to abstractly refer to the dimensions of the domain but in this case they are both equal to 1. I will be using a Neumann boundary condition along the boundary at $x = L, y \in [0, H]$ (the right boundary) and Dirchlet boundary conditions for the others. I will characterize the conditions,

$$u(x_i, 0) \equiv f_1(x_i) = \frac{x_i}{L}$$
$$u_{x_i,L} \equiv f_2(x_i) = \left(\frac{L - x_i}{L}\right)$$
$$u(0, y_j) \equiv g(y_j) = \frac{y_j}{H}$$
$$\vec{\nabla} u \cdot \vec{n}\Big|_{y=y_j} = 0, \quad \vec{n} = \vec{x}$$

5.1 *Interior*: Derive and state the classic 4th-order 9-point Laplacian on a uniform grid ($h_x = h_y = h$).

The Laplacian is comprised of the sum of two mono-variate double derivatives.

$$\Delta u(x,y) = u_{xx} + u_{yy}$$

From which we can intuit that a double derivative in one looks the same as a double derivative in the other. So, I will work to derive the stencil in one representation and then 'intuite' the form of the other. So,

$$u_{xx}h^2 = c_0 u_{i-2} + c_1 u_{i-1} + c_2 u_i + c_3 u_{i+1} + c_4 u_{i+2}$$

Now we do a Taylor series expansion like we have done previously and group by orders of the derivative,

$$u_{xx}h^2 \approx (c_0 + c_1 + c_2 + c_3 + c_4)u_i + h(-2c_1 - c_2 + c_3 + 2c_4)u_i' + \frac{h^2}{2}(4c_1 + c_2 + c_3 + 4c_4)u_i''$$
$$+ \frac{h^3}{6}(-8c_1 - c_2 + c_3 + 8c_4)u_i''' + \mathcal{O}(h^4)$$

This can be represented as,

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & -2 & -1 & 1 & 2 \\ 0 & 4 & 1 & 1 & 4 \\ 0 & -8 & -1 & 1 & 8 \end{bmatrix} \vec{c} = \begin{bmatrix} 0 \\ 0 \\ 2 \\ 0 \\ 0 \end{bmatrix}$$

Solving this system tells us that $\vec{c} = [-\frac{1}{12}, \frac{4}{3}, -\frac{5}{2}, \frac{4}{3}, -\frac{1}{12}]$. And so our system will look like,

$$u_{xx}+u_{yy} = \frac{1}{h^2}\left[-\frac{1}{12}u_{i-2,j} + \frac{4}{3}u_{i-1,j} - \frac{5}{2}u_{i,j} + \frac{4}{3}u_{i+1,j} - \frac{1}{12}u_{i+1,j} - \frac{1}{12}u_{i,j-2} + \frac{4}{3}u_{i,j-2} - \frac{5}{2}u_{i,j} + \frac{4}{3}u_{i,j+1} - \frac{1}{12}u_{i,j+2}\right]$$

$$= \frac{1}{h^2}\left[-\frac{1}{12}u_{i-2,j} + \frac{4}{3}u_{i-1,j} + \frac{4}{3}u_{i+1,j} - \frac{1}{12}u_{i+1,j} - \frac{1}{12}u_{i,j-2} + \frac{4}{3}u_{i,j-2} - 5u_{i,j} + \frac{4}{3}u_{i,j+1} - \frac{1}{12}u_{i,j+2}\right]$$

$$= \frac{1}{12h^2}\left[16(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1}) - (u_{i+2,j} + u_{i-2,j} + u_{i,j-2} + u_{i,j+2}) - 60u_{i,j}\right]$$

If we expand all terms in the square brackets via a Taylor series expansion then we see many terms cancel due to the symmetry of the grouped terms. The terms that cancel will be odd ordered as the signs on the offsets carry over meanwhile the even ordered terms will sum, leaving:

$$\begin{aligned}\Delta u 12h^2 =& 16(4u_{ij} + 2\frac{h^2}{2}(u_{xx} + u_{yy}) + 2\frac{h^4}{24}(u_{xxxx} + u_{yyyy}))\\ &- (4u_{ij} + 8\frac{h^2}{2}(u_{xx} + u_{yy}) + 32\frac{h^4}{24}(u_{xxxx} + u_{yyyy})) - 60u_{ij} + \mathcal{O}(h^6)\\ =& 12h^2(u_{xx} + u_{yy}) + \mathcal{O}(h^6) \quad \text{and,}\end{aligned}$$

$$\Delta u = u_{xx} + u_{yy} + \mathcal{O}(h^4)$$

5.2 Dirichlet sides: State how to enforce given u = g at boundary nodes and how these values modify the RHS adjacent to the boundary for a 4th-order scheme.

When we select an internal point in the mesh which has one of its branches touch the boundary we require that point is known via our boundary condition. This eliminates the point of the stencil so we subtract that value and shift it to the RHS vector, $\vec{b}$.

5.3 *Neumann side*: Derive a 4th-order one-sided normal derivative to eliminate ghost points.

As an example, let's select a point in the mesh, $u_{i,j}$, such that $j > i = N - 2$ for a square NxN representation. That is, our stencil only touches one boundary. Our stencil will look like,

$$\begin{aligned}\Delta u_{N-2,j} =& \frac{1}{12h^2}[16(u_{N-3,j} + u_{N-1,j} + u_{N-2,j-1} + u_{N-2,j+1})\\ &- (u_{N,j} + u_{N-4,j} + u_{N-2,j-2} + u_{N-2,j+2} - 60u_{N-3,j}) - 60u_{N-2,j}\end{aligned}$$

We see explicitly that there is one point $U_{N,j}$ which is not a member of our current stencil and needs to be addressed because the value there is not given. We do no how its first derivative behaves from our Neumann boundary condition, $u_x = 0$. Using a stencil to give fourth order,

$$\begin{aligned}
hu_x =&c_0u_{N,j} + c_1u_{N-1,j} + c_3u_{N-2,j} + c_4u_{N-3,j} + c_5u_{N-4,j} \\
\approx&(c_0 + c_1 + c_2 + c_4 + c_5)u_{N,j} - h(c_1 + 2c_2 + 3c_4 + 4c_5)u'_{N,j} \\
&\frac{h^2}{2}(c_1 + 4c_2 + 9c_4 + 16c_5)u''_{N,j} - \frac{h^3}{6}(c_1 + 8c_2 + 27c_4 + 64c_5)u'''_{N,j} \\
&+ \frac{h^4}{24}(c_1 + 16c_2 + 81c_4 + 256c_5)u''''_{N,j} + \mathcal{O}(h^5)
\end{aligned}$$

Same steps, represent as a matrix, solve the system, get the vector: $\vec{c} = [\frac{25}{12}, -4, 3, -\frac{4}{3}, \frac{1}{4}]$ and we can see our stencil comes out to be,

$$\begin{aligned}
u_x &= \frac{1}{12h}\left[25u_{N,j} - 48u_{N-1,j} + 36u_{N-2,j} - 16u_{N-3,j} + 3u_{N-4,j}\right] = 0 \\
\implies u_{N,j} &= \frac{1}{25}[48u_{N-1,j} - 36u_{N-2,j} + 16u_{N-3,j} - 3u_{N-4,j}]
\end{aligned}$$

Plug this into our stencil,

$$\begin{aligned}
\Delta u_{N-2,j} =&\frac{1}{12h^2}[16(u_{N-3,j} + u_{N-1,j} + u_{N-2,j-1} + u_{N-2,j+1}) \\
&- (u_{N-4,j} + u_{N-2,j-2} + u_{N-2,j+2} - 60u_{N-3,j}) - 60u_{N-2,j} \\
&- \frac{1}{12 \cdot 25h^2}[48u_{N-1,j} - 36u_{N-2,j} + 16u_{N-3,j} - 3u_{N-4,j}]
\end{aligned}$$

And so, a correction to the stencil in terms of points we are confident in.

5.4 ***Assembly***: Build the sparse matrix A and vector b for the full grid using scipy.sparse

The code for the next set of problems are all bundled together at D.1. The formatting for the plots in this section is annoying. This is Overleaf's biggest downfall and I wish it would be better. Please follow the hyperlinks where appropriate for fast parsing.

I am having difficulty implementing the Neumann boundary condition as discussed in the last section. I do have a representation for the Laplacian to show. See figure 7. As we can see, the general form of the stencil is fine up to the Dirichlet boundary condition in which we see the diagonals take shape. There exist some cells that are not properly surrounded (some purple cells are not surrounded horizontally by yellow squares). This is addressed in the Neumann condition by replacing the row with the 5 point one sided stencil. the vector, $\vec{b}$ is seen in figure 8

5.5 ***Right-hand side***: Discretize f by nodal sampling

See figure 9.

5.6 ***Solve***: Use scipy.sparse.linalg.spsolve(A, b)

See figure 10. We see some abberant behaviour in the solution for this system when we compare to the analytic solution provided by the professor. I believe this is linked to the current implementation of the Neumann boundary condition.

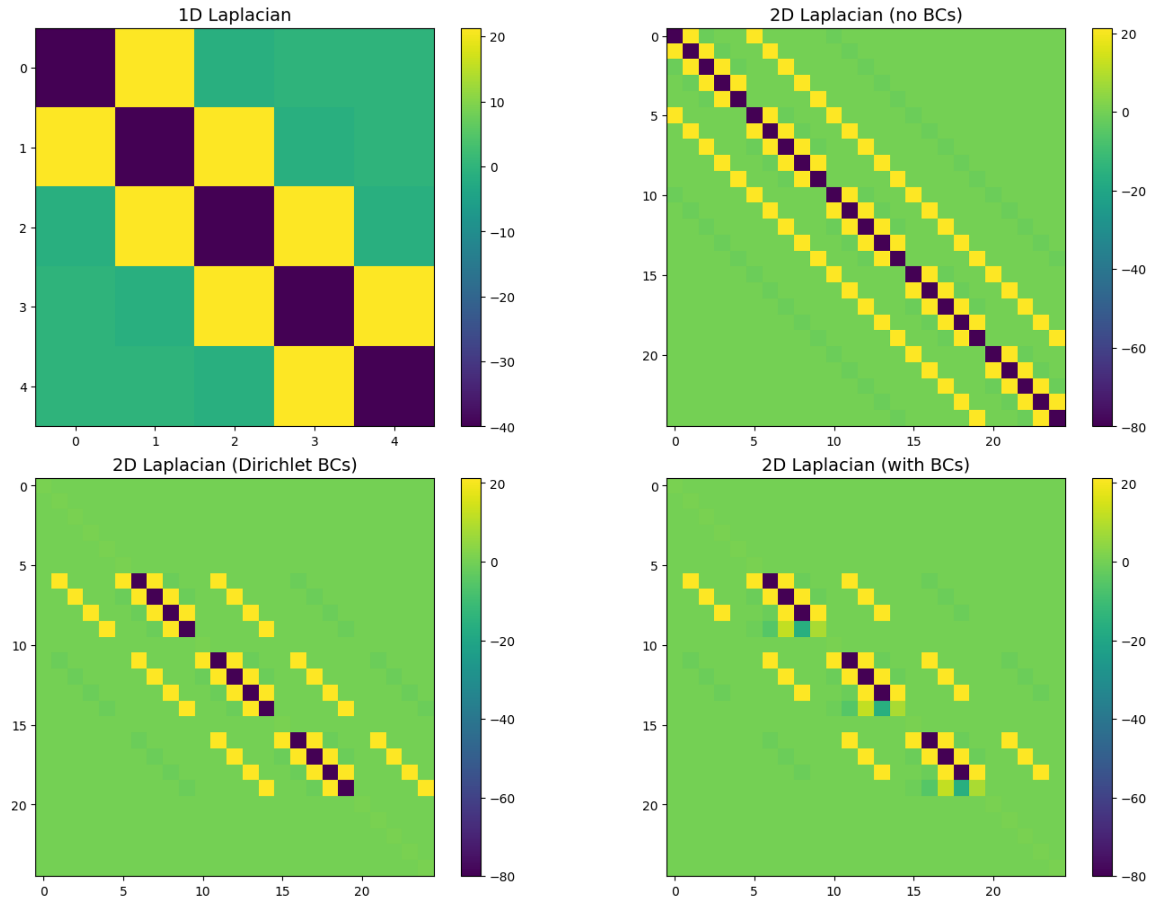

Figure 7: The Laplacian stencil in stages. In the top-left (TL) is the stencil in 1D, TR is the 2D stencil via the Kronecker product, BL 2D stencil with Dirchlet boundary conditions, and BR is the Neumann boundary condition.

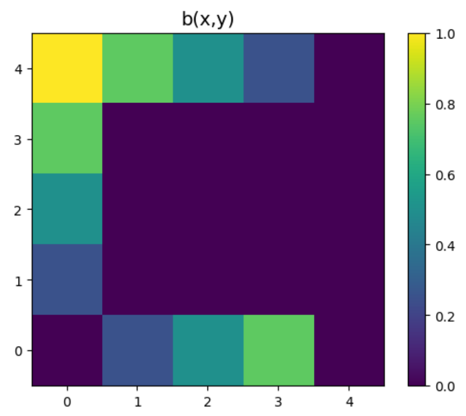

Figure 8: $\vec{b}$ for a 5×5 system.

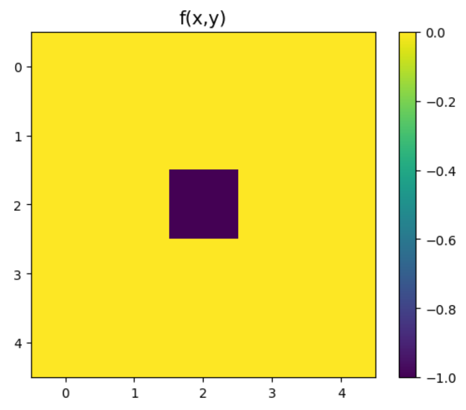

Figure 9: $\vec{f}$ for a 5×5 system.

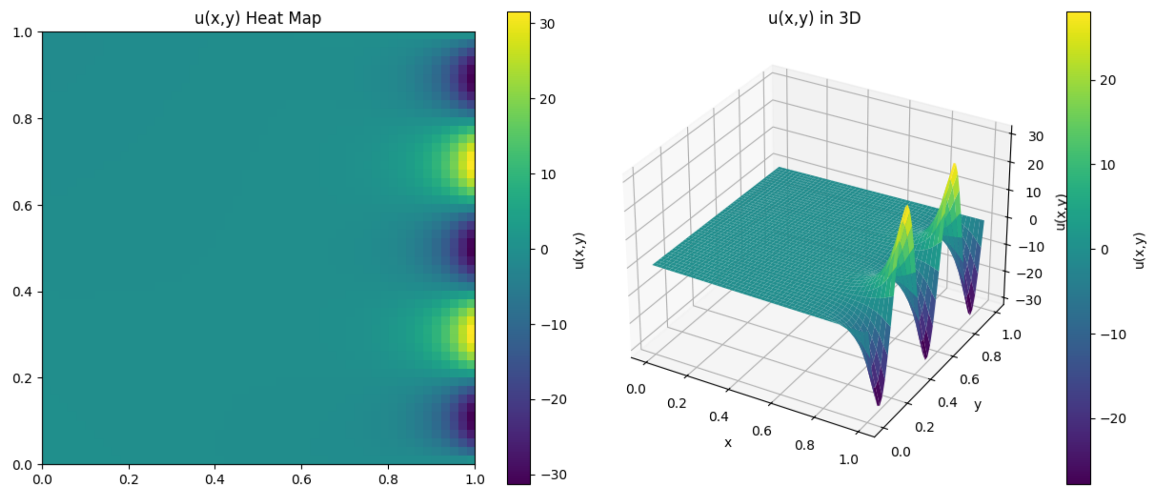

Figure 10: A heatmap and 3D projection of the solution for a $50^2 \times 50^2$ system.

As an avenue of exploration I wondered what my system looked like if I solved it for the Laplacian with homogeneous Dirichlet boundary conditions on all sides. The hope was that I could use it as a diagnostic tool to see where I went wrong. Looking at figure 11 and comparing it to the analytic solution the professor provided we see some nice qualitative agreement. However, there is still some tuning necessary which I assume is accomplished by the boundary conditions.

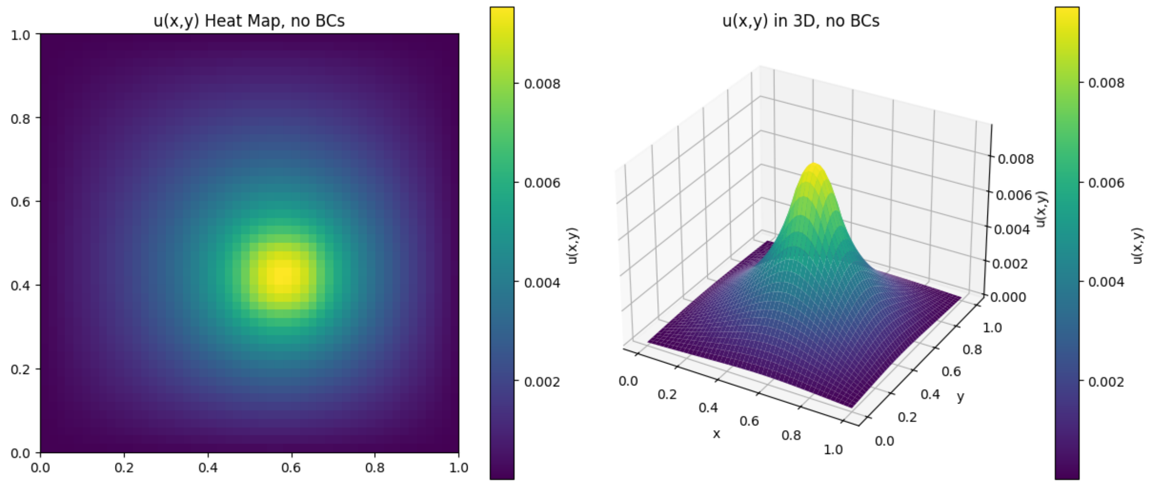

Figure 11: A heatmap and 3D projection of the solution for a $25^2 \times 25^2$ system with Dirichlet boundary conditions on all sides.

A Part 1

A.1 Taylor series → lambdify → plots

```
    """
Name: maclaurin series plotter
Purpose: this calculates the maclaurin series expansion of a passed function
         and plots the real and imaginary parts along with the difference in
         true function and the approximation (it also reports the l-inf norm
             ).
         Also, prints maclaurin series in next cell.
Author: Joshua Belieu | Fletch
Date written: 2025-09-01
Last modified: 2025-09-10
Inputs:
- x,y,z,... : sympy symbol. the dependant variable of the desired function.
- f: sympy expression. the function to approximate in sympy format.
- x_arr : numpy array. the numerical array of the dependant variable(s).
- f_arr_true : numpy array. the numerical array of the desired function.
- order_list : list of ints. the orders of the derivative to expand to
               (inclusive)
Outputs:
- plot : plot. a 2 row x len(order_list) subplot of the expressions (top row
    )
         and the difference in the true and approximate functions with l-inf
         norn (bottom row).
Dependencies:
- none.
"""

'''
establish variable and function.
'''

x = sp.symbols ( 'x' )
f = sp.exp ( (1 + 1j) * x ) / ( 1 + x ** 2 )

'''
numerical x interval for evaluation and true solution for comparison of
numerical approximation
'''

x_arr = np.linspace ( -2 , 2 , int ( 1e2 ))
f_arr_true = np.exp( (1 + 1j) * x_arr ) / ( 1 + x_arr ** 2 )

'''
list of derivative order for M.S.E.
'''

order_list = [ 4 , 8 , 12 ]

fig,axs = plt.subplots ( 2 , len ( order_list ),
                         figsize = ( 12 , 8 ),
                         constrained_layout = True )

plt.rcParams.update({'font.size': 12})
```

```python

expression_list = []

for idx , order in enumerate ( order_list ) :

    '''
    do a series expansion of f(x) at x=0 with order (inclusive). remove the
    omicron and simplify the expression. then, make it an evaluatable
        expresion
    that returns a numpy array.
    '''

    maclaurin_expansion = sp.series ( f , x , 0 , order + 1 )
    maclaurin_expansion_sympy = sp.simplify(maclaurin_expansion.removeO())
    maclaurin_expansion_func = sp.lambdify( x, maclaurin_expansion_sympy , '
        numpy' )

    expression_list.append( maclaurin_expansion_sympy )

    '''
    get numerical form, and extract max norm for real and imaginary parts.
    '''

    f_arr = maclaurin_expansion_func( x_arr )
    l_inf_real = np.max( np.abs( f_arr.real-f_arr_true.real ) )
    l_inf_imag = np.max( np.abs( f_arr.imag-f_arr_true.imag ) )

        '''
    plot!
    '''

    axs[0][idx].plot( x_arr, f_arr.real, label=f'Real part' )
    axs[0][idx].plot( x_arr, f_arr.imag, label=f'Imaginary part' )
    axs[0][idx].set_title( f'Maclaurin expansion (order={order})' )
    axs[0][idx].set_xlabel('x')
    axs[0][idx].set_ylabel(r'$f_{numerical}$')
    axs[0][idx].grid(ls='--',alpha=0.5)
    axs[0][idx].legend()

    axs[1][idx].plot( x_arr, f_arr.real-f_arr_true.real,label= fr'$l_\infty=
        ${l_inf_real:.2e}' )
    axs[1][idx].plot( x_arr, f_arr.imag-f_arr_true.imag,label=fr'$l_\infty=$
        {l_inf_imag:.2e}' )
    axs[1][idx].set_xlabel('$x$')
    axs[1][idx].set_ylabel(r'$f_{numerical}-f_{true}$')
    axs[1][idx].grid(ls='--',alpha=0.5)
    axs[1][idx].legend();

for row in axs :
    for col in row :
        col.tick_params(axis='both',which='both', direction='in')

    for expr in expression_list :

    print(expr)
```

A.2 AI-Assist

Please follow this link to see the engagement with the AI on this problem.

I used the LLM on these dates : 27.08.25 and 29.08.25

A.3 Verify

```
    """
Name: compare results
Purpose: compare the results of the sympy and LLM outputs
Author: Your Name
Date written: 2025-09-01
Last modified: 2025-09-10
Inputs:
- ai_expressions : list of strings. the generated maclaurin series from the
    LLM
                   in string format
- expression_list : list of strings. the sympy generated maclaurin series in
                    string format
Outputs:
- diff : a printout of the order of derivative, the two expressions, and
    their
         difference.
Dependencies:
- maclaurin series plotter (this generates the expression list)
"""

'''
ChatGPT 5.0 expressions from the first week of school (I forgot to take down
the exact date and I can't find it on GPT's site.)
'''

ai_expressions = [
"1 + (1+1j)*x + (-1+1j)*x**2 + (-4/3 - (2/3)*1j)*x**3 + (5/6 - 1j)*x**4",
"1 + (1+1j)*x + (-1+1j)*x**2 + (-4/3 - (2/3)*1j)*x**3 + (5/6 - 1j)*x**4 "
                "+ (13/10 + (19/30)*1j)*x**5 + (-5/6 + (89/90)*1j)*x**6 "
                "+ (-409/315 - (40/63)*1j)*x**7 + (2101/2520 - (89/90)*1j)*x
                    **8",
"1 + (1+1j)*x + (-1+1j)*x**2 + (-4/3 - (2/3)*1j)*x**3 + (5/6 - 1j)*x**4 "
                "+ (13/10 + (19/30)*1j)*x**5 + (-5/6 + (89/90)*1j)*x**6 "
                "+ (-409/315 - (40/63)*1j)*x**7 + (2101/2520 - (89/90)*1j)*x
                    **8 "
                "+ (4207/3240 + (14401/22680)*1j)*x**9 "
                "+ (-2101/2520 + (112141/113400)*1j)*x**10 "
                "+ (-202462/155925 - (44003/69300)*1j)*x**11 "
                "+ (6239969/7484400 - (112141/113400)*1j)*x**12"
]



for i in range ( len ( expression_list ) ) :

    sympy_expression = expression_list [ i ]
    ai_expression = sp.sympify ( ai_expressions [ i ] )
```

A.4 Vandermonde system

```
def fd_weights ( offsets , order , show_system = False ) :

global s , A

"""
Name          : fd_weights -> finite difference weights
Purpose       : calculate the set of coefficients for a finite difference
                approximation of the order-th derivative using the given
                offsets. We are essentially solving the linear system
                A c = b, where A is the Vandermonde matrix of the offsets,
                c is the vector of coefficients we want to find, and b is
                a vector of zeros with a factorial(order) in the order-th
                position.
Author          : christlieb, augmented by Joshua Belieu.
Date augmented  : 2025-08-26
Last modified   : 2025-08-26
Inputs :
    - offsets     : list. list of offsets (in units of grid stride h)
    - order       : int. order of derivative to approximate
    - show_system : bool. if True, print the system of equations being
                    solved. default is False.
Outputs:
    - <ret>: <type> ... <meaning/units/shape>
Dependencies:
    - None.
"""

'''
list of offsets (in units of grid stride h), nsimplify allows floats.
'''

s = [ sp.nsimplify ( offset ) for offset in offsets ]
m = len ( s )

'''
the Vandermonde matrix of the offsets. the "A" in our system Ac = b
'''

A = sp.Matrix ([[ s[j] ** k for j in range ( m ) ] for k in range ( m ) ])

'''
a vector of zeros with a factorial(order) in the order-th position. the
"b" in our system Ac = b.
'''

b = sp.Matrix ( [ 0 ] * m )
b [ order ] = sp.factorial ( order )

'''
```

```
    the coefficients we want to find. the "c" in our system Ac = b. LUsolve
        is
    a decomposition method that splits A into a lower and upper triangular
        matrix
    and solves the system (Ux=y and Ly=c) in two steps.
    '''

    c = A.LUsolve ( b )

    if show_system :

        system = sp.Eq( sp.MatMul ( A , c , evaluate = False ) , b )
        sp.pprint(system)

    return sp.simplify ( c )
```

A.5 Programatic moment verification

```
    """
Name: moment printer
Purpose: provided some weights and offsets, compute the order moments from
    an
         expansion
Author: Joshua Belieu | Fletch
Date written: 2025-09-01
Last modified: 2025-09-10
Inputs:
- offsets : array-like. a collection of the offsets in the stencil.
- weights : array-like. a collection of coefficients determined for the
    system.
Outputs:
- moments : printout. this prints to terminal the order of the moment and
    its
            value.
Dependencies:
- fd_weights
"""

offsets_arr = np.array ( offsets )

for i in range ( len(offsets_arr) + 1 ) :

    moment_i = sp.simplify ( np.sum ( np.dot ( weights , offsets_arr ** i )
        ) )

    print ( f'moment {i} : {moment_i}' )
```

A.6 Numerical check

```
    """
Name: error and order tester
Purpose: test test the error and order of a method at different step sizes.
Author: Joshua Belieu | Fletch
Date written: 2025-09-01
```

```
Last modified: 2025-09-10
Inputs:
- f : array-like. the function to approximate at a point
- f_prime : array-like. the derivative of f to compare against.
- x0 : float. the expansion point.
- idx_arr : array-like. a collection of refinement levels for the step size.
            goes like h^(-idx).
Outputs:
- table : a table printout that shows the refinement level, step size,
          approximation value, error, and error/h^4 (order diagnostic).
- plot : idx_arr vs error/(h^i). this shows what the order of the method is.
         curves with slope ~ 0 are the order. (next cell)
Dependencies:
- fd_weights
"""

'''
function and derivative setup. used for calculation and verification.
'''

f = lambda x : np.sin ( x )
f_prime = lambda x : np.cos ( x )
x0 = 0.

'''
idx represents inverse power of step size. err is the container for each
    error.
h_arr is used later for plotting so we can see the effect of smaller
    stepsizes.
'''

idx_arr = np.arange ( 3 , 8 + 1 )
err_arr = np.zeros ( len ( idx_arr ) )
h_arr = 2. ** ( -idx_arr )

'''
i/o header for calculation.
'''

head_str = f"|{'idx':^4}|{'h=2^-idx':^11}|{'approximation':^18}|{'error
    ':^11}|{'error/h^4':^11}|"
print('='*len(head_str))
print(head_str)
print('-'*len(head_str))

'''
for each refinement level, calclulate the step size and use that to
    calculate
the function and its approximation via an inner product with the weights
divided by the step size. due to the shapes approx.shape is (1,). the error
is the absolute value of the difference.
'''

for i in idx_arr :

    h = 2. ** ( -i )

    f_arr = f(x0+h*offsets_arr)
    approx = np.dot(weights,f_arr)[0] / h
    error = np.abs(approx-f_prime(x0))
    err_arr[i-3] = error

    form_str = f"|{str(i).zfill(2):^4}|{h:^11.3e}|{approx:^18}|{error:^11.3e
       }|{error/h**4:^11.4e}|"
    print(form_str)
print('='*len(head_str))

for i in range ( 6 + 1 ) :

    plt.plot(idx_arr, err_arr/h_arr**(i), 'o-',label=f'error/h^{i}')

# plt.xscale('log')
plt.yscale('log')
plt.grid(ls='--',alpha=0.5)
plt.xlabel('Refinement level')
plt.ylabel('Error/h^i')
plt.tick_params(axis='both',which='both', direction='in')
plt.legend();
```

A.7 Symbolic moment verification

```
    """
Name: symbolic moments tester
Purpose: compute moments symbolically to show order
Author: Joshua Belieu | Fletch
Date written: 2025-09-01
Last modified: 2025-09-10
Inputs:
- offsets : array-like. collection of stencil points
- weights : array-like. collection of weights calculated from offsets.
Outputs:
- moments : sympy matrix. the vector of moments.
Dependencies:
- fd_weights
"""

'''
compound list comprehension to setup 2d list and convert to matrix object in
sympy. just wrap the weights with the Matrix function. matrix multiply to
    get
moments vector (must transpose weights cause of shape shenanigans)
'''

offsets13_mat = sp.Matrix ( [ [ off ** i for off in offsets13 ] for i in
    range(len(offsets13)+1) ] )
weights13_mat = sp.Matrix ( weights13 )

moments = offsets13_mat*weights13_mat.T

"""
```

```
results indicate k* = 6 and so,
p = k* - d = 4
"""

moments
```

B Part 2

B.1 Multi-step size error refinement

```
    """
Name: Error and step size method explorer
Purpose: to investigate the effect that step size has on a method. we
    specify
          500 maximum values of step size (H) and draw from a uniform
          distribution three values on the interval [0,H] and use that to
          generate the coefficients of the ansatz. Then, we test the
              performance
          against a trial function and plot the behaviour
Author: Joshua Belieu | Fletch
Date written: 2025-09-06
Last modified: 2025-09-11
Inputs:
- f : array-like. function we will try to approximate an order d derivative.
- f_2prime : array-like. the 2nd order derivative of f.
- x0 : float/int. expansion point
- H_lower/higher : float. The bounds of the interval for H
Outputs:
- error_list : list. collection of errors for each value in H
- H_arr : numpy array. the values of H.
- plot : plot. log-log plot of H vs |Error|.
Dependencies:
- fd_weights
"""

f = lambda x : np.cos ( x )
f_2prime = lambda x : -np.cos ( x )
x2 = 0

H_lower = 1e-3
H_upper = 1e-1
H_arr = np.linspace ( H_lower , H_upper , int ( 5e2 ) )

error_list = []

rng = np.random.default_rng()
for H_val in H_arr :

    h1,h2,h3 = rng.uniform(0.,H_val,size=3)
    h_arr = np.array([ -h1,0,h2,h2+h3 ])
    f_arr = f(x2+h_arr)

    c1, c2, c3, c4 = sp.symbols('c1 c2 c3 c4')

    eq1 = sp.Eq(c1 + c2 + c3 + c4, 0)
    eq2 = sp.Eq(-h1*c1 + h2*c3 + (h2+h3)*c4, 0)
    eq3 = sp.Eq((h1**2*c1 + h2**2*c3 + (h2+h3)**2*c4)/2, 1)
    eq4 = sp.Eq((-h1**3*c1 + h2**3*c3 + (h2+h3)**3*c4)/6, 0)

    c_vec = sp.solve([eq1, eq2, eq3, eq4], [c1, c2, c3, c4])

    '''
    convert sympy floats/array to numpy floats/array because np.log doesnt
    like sympy floats/arrays :(.
    '''

    c_arr = np.array([ float(c_vec[c]) for c in (c1,c2,c3,c4) ])
    approx = np.dot(c_arr,f_arr)
    error = np.abs(approx - f_2prime(x2))
    error_list.append(error)

    plt.scatter(H_arr,error_list)

plt.xlabel('H')
plt.ylabel('|Error|')
plt.grid(ls='--',which='both',alpha=0.3)
plt.xscale('log')
plt.yscale('log')
plt.tick_params(axis='both',which='both', direction='in');
```

B.2 Least squares fit

```
    """
Name: least squares analysis
Purpose: this calculates the vector x from Ax=b where A is a 2xlen(error)
         matrix where the left column is a vector of 1s and the right column
         is the log(H). b is the log(|error_list|)
Author: Joshua Belieu | Fletch
Date written: 2025-09-06
Last modified: 2025-09-11
Inputs:
- A : numpy array. 2xlen(error) where the left column is a vector of 1s and
    the
      right column is the log(H).
- b : numpy array. the log(|error_list|)
Outputs:
- K,p : float. the intercept and slope of the linear least squares fit.
- plot : the scatter plot from Error and step size method explorer with the
         line placed over top. the axis scales are already log-log so in
             stead
         of plotting K+log(H)*p you input e^(K)*H^p
Dependencies:
- Error and step size method explorer
"""

A = np.column_stack ( [np.ones_like(H_arr),np.log(H_arr)] )
b = np.log ( error_list )
x = np.linalg.lstsq ( A , b ,rcond=None)

K,p = x[0]
```

```

plt.scatter(H_arr,error_list)
plt.plot(H_arr,np.exp(K)*H_arr**p,color='red')
plt.xlabel('H')
plt.ylabel('|Error|')
plt.grid(ls='--',which='both',alpha=0.3)
plt.xscale('log')
plt.yscale('log')
plt.tick_params(axis='both',which='both', direction='in');
```

C Part 3

C.1 Multi-lattice point error order test

```
    """
Name: Sparse matrix solver
Purpose: establish a second order accuracy for this method/system at
    different
        grid spacings.
Author: Joshua Belieu | Fletch
Date written: 2025-09-13
Last modified: 2025-09-13
Inputs:
- x_low/high : float/int. the system domain limits.
- lattice_points : array-like. the dimensions of the discretized system, nxn
    .
- u_true : array-like. the analytic solution of the system.
- A[i,j] : float/int. these adjustments are due to the boundary values given
           in the problem statement. make sure they're accurate for your
           system!
Outputs:
- plot : plot. a plot of the correlation between the lattic points and the
    max
         value in the absolute value of the difference in the approximation
         and analytic expression divided by some order of the lattice
             spacing,
         h. for reference, h=x_upp/(lattice_point-1)
Dependencies:
- None.
"""

'''
bounds of the system
'''

x_low = 0.; x_upp = 10.

'''
the dimensions of the array, nxn. the spacings are h and are used in the
    plot.
max_err_ls is max error list and is used to store the max error for each
lattice size.
'''

lattice_points = np.array([40,60,80,100])
```

```
lattice_spacing = x_upp/(lattice_points-1)
max_err_ls = []

'''
for each lattice size do; calculate step size, model the true function,
generate a sparse matrix and adjust the first and last rows according to
boundary values, discretize the RHS of the ODE scaled by h^2, and solve for
    the
eigenfunction using spsolve. take the |difference| of the approximation and
    the
true function, extract the max value and add it to max_err_ls.
'''

for n in lattice_points :

    h = x_upp / (n-1)
    x_arr = np.linspace(x_low,x_upp,n)
    u_true = lambda x: np.exp(x_upp)/(2*np.cos(x_upp))*(np.cos(x)+np.sin(x))
        -0.5*np.exp(x)


    e = np.ones(n)
    A = diags([e, (-2+h**2)*e, e], offsets=[-1,0,1], shape=(n,n)).tolil()
    A[0,0] = -3/(2*h) - 1
    A[0,1] = 4/(2*h)
    A[0,2] = -1/(2*h)

    A[-1, -3] = 1/(2*h)
    A[-1, -2] = -2/h
    A[-1, -1] = 3/(2*h) + 1

    f = -np.exp(x_arr)*h**2
    u = spsolve(A.tocsr(),f)

    diff = u-u_true(x_arr)

    error = np.abs(diff)
    max_err = np.max(error)
    max_err_ls.append(max_err)

'''
for different orders, plot lattice points against max_err_ls/h^order to test
what order our method is.
'''

for i in range (1,6) :

    plt.plot(lattice_points,
             np.array(max_err_ls)/lattice_spacing**i,
             marker='o',
             label=i);

plt.xlabel('Lattice points',fontsize=14)
plt.ylabel(r'||Error||$_{\infty}$/h^i',fontsize=14)
plt.yscale('log')
plt.grid(ls='--',alpha=0.5)
plt.tick_params(axis='both',which='both', direction='in')
plt.legend(title='i',loc='upper left');
```

C.2 Solving for A,D using sympy

```
    A,D,x = sp.symbols('A D x')
g = (-sp.cos(10)+sp.sin(10))/(sp.cos(10)+sp.sin(10))

eq1 = sp.Eq((sp.sin(x)+sp.cos(x))*A-(g*sp.sin(x)+sp.cos(x))*D,0)
eq2 = sp.Eq((-sp.sin(x)+sp.cos(x))*A-(sp.sin(x)-g*sp.cos(x))*D,1)

solution = sp.solve([eq1, eq2], [A, D])
print(solution)
```

D Part 4

D.1 Laplacian code

```
    """
Name: 2d laplacian plotter
Purpose: plot the laplacian at each development step to amake sure i dont
    guff
         it up.
Author: Joshua Belieu | Fletch
Date written: 2025-09-17
Last modified: 2025-09-20
Inputs:
- n : int. number of lattice points
- weights : array-like. container of stencil weights.
- offsets : array-like. container of stencil offsets. elements represent
            distance from main diagonal.
- no name : floats/ints. inside the for loop are the weights of the neumann
            stencil which are placed according to offsets. example :

            A_2D[loop index, neumann_offset] = neumann_weight

Outputs:
- plots : plot. 2 plots. a 1x3 plot of the 1d laplacian, a 2d laplacian
    derived
          using the kronecker product, and a 2d laplacian with neumann
          conditions baked in at one boundary. a second plot which is the
              third
          plot of the 1x3 is in the next cell with the values painted at
              each
          lattice point.
Dependencies:
- None
"""

n = 5 # 1D lattice points cheese
h = 1/(n-1) # step size
# h = np.sqrt(1/12) # test value

```

```
weights = [-1/12, 4/3, -5/2, 4/3, -1/12] # 1d laplacian weights
offsets = [-2, -1, 0, 1, 2] # 1d laplacian offsets

'''
the 1d stencil of a second derivative operator
'''

a_1d = diags(np.array(weights)/(h**2), offsets, shape=(n,n))

'''
identity is a matrix diag(1). the 2d laplacian is the kronecker product of
the 1d laplacian and the identity in swapped order summed together. the
dimensions are now n^2 x n^2.
'''

identity = eye(n)
a_2d = kron(identity, a_1d) + kron(a_1d,identity)
a_2d_nobc = a_2d.copy()
a_2d = a_2d.tolil()

'''
b is the Dirichlet boundary conditions adjustment vector. this will store
    the
points in the stencil that are known and will deduct from the f vector to
    form
a rhs vector for our pde.
'''

b = np.zeros(n*n)

'''
dimensions of domain.
'''

L = 1 ; H = 1
x = np.linspace ( 0 , L , n ); y = np.linspace ( 0 , H , n );

'''
functions along the boundaries.
'''

f1 = x / L ; f2 = (L-x)/L ; g = y/H

'''
a big loop to set the boundary conditions. in essence, we clear the rows out
that correspond to the Dirichlet boundaries and set the diagonal to 1. we
    then
store those values in b to subtract from f later. the neumann condition is
    set
using a 4th order one-sided stencil (under construction).
'''

for idx in range ( n ) :

    left_bound_idx = idx * n

    a_2d [ left_bound_idx ,: ] = 0
    a_2d [ left_bound_idx , left_bound_idx ] = 1
    b [ left_bound_idx ] = g [ idx ]

    bottom_bound_idx = idx

    a_2d [ bottom_bound_idx ,: ] = 0
    a_2d [ bottom_bound_idx , bottom_bound_idx ] = 1
    b [ bottom_bound_idx ] = f1 [ idx ]

    upper_bound_idx = n * ( n - 1 ) + idx

    a_2d [ upper_bound_idx ,: ] = 0
    a_2d [ upper_bound_idx , upper_bound_idx ] = 1
    b [ upper_bound_idx ] = f2 [ idx ]

a_2d_dirichlet = a_2d.copy()

for idx in range ( n ) :

    neumann_row_idx = idx * n + n - 1

    if idx > 0 and idx < n-1 :

        a_2d [ neumann_row_idx ,:  ] = 0
        a_2d [ neumann_row_idx , neumann_row_idx] = 1
    # a_2d [ neumann_row_idx , neumann_row_idx] = 25/(12*h)
    # a_2d [ neumann_row_idx , neumann_row_idx -1 ] = -48/(12*h)
    # a_2d [ neumann_row_idx , neumann_row_idx -2 ] = 36/(12*h)
    # a_2d [ neumann_row_idx , neumann_row_idx -3 ] = -16/(12*h)
    # a_2d [ neumann_row_idx , neumann_row_idx -4 ] = -3/(12*h)

# a_2d [ neumann_row_idx , neumann_row_idx ] = 0
        a_2d [ neumann_row_idx , neumann_row_idx -1 ] = a_2d [
            neumann_row_idx , neumann_row_idx -1 ] -48/(25*h)
        a_2d [ neumann_row_idx , neumann_row_idx -2 ] = a_2d [
            neumann_row_idx , neumann_row_idx -2 ] + 36/(25*h)
        a_2d [ neumann_row_idx , neumann_row_idx -3 ] = a_2d [
            neumann_row_idx , neumann_row_idx -3 ] -16/(25*h)
        a_2d [ neumann_row_idx , neumann_row_idx -4 ] = a_2d [
            neumann_row_idx , neumann_row_idx -4 ] +3/(25*h)
        # a_2d [ neumann_row_idx , neumann_row_idx -1 ] = a_2d [
            neumann_row_idx , neumann_row_idx -1 ] -48/(25*12*h**2)
        # a_2d [ neumann_row_idx , neumann_row_idx -2 ] = a_2d [
            neumann_row_idx , neumann_row_idx -2 ] + 36/(25*12*h**2)
        # a_2d [ neumann_row_idx , neumann_row_idx -3 ] = a_2d [
            neumann_row_idx , neumann_row_idx -3 ] -16/(25*12*h**2)
        # a_2d [ neumann_row_idx , neumann_row_idx -4 ] = a_2d [
            neumann_row_idx , neumann_row_idx -4 ] +3/(25*12*h**2)

    b [ neumann_row_idx ] = 0.

'''
setting up the f vector as consistent with the problem statement.
```

```
'''

f = np.zeros((n,n))

for i in range ( n ) :
    for j in range ( n ) :
        if 1/3 <= x[i] <= 1/2 and 1/2 <= y[j] <= 2/3 :
            f[i,j] = -1

f = f.flatten()

rhs = f - b

'''
plot the derivative coefficient matrix at each development step.
''';

fig, axs = plt.subplots(2, 2, figsize=(14,10), constrained_layout=True)

im0 = axs[0][0].imshow(a_1d.toarray(), cmap="viridis", interpolation="none")
axs[0][0].set_title("1D Laplacian",fontsize=14)
plt.colorbar(im0, ax=axs[0][0])

im1 = axs[0][1].imshow(a_2d_nobc.toarray(), cmap="viridis", interpolation="none")
axs[0][1].set_title("2D Laplacian (no BCs)",fontsize=14)
plt.colorbar(im1, ax=axs[0][1])

im2 = axs[1][0].imshow(a_2d_dirichlet.toarray(), cmap="viridis", interpolation="none")
axs[1][0].set_title("2D Laplacian (Dirichlet BCs)",fontsize=14)
plt.colorbar(im2, ax=axs[1][0]);

im3 = axs[1][1].imshow(a_2d.toarray(), cmap="viridis", interpolation="none")
axs[1][1].set_title("2D Laplacian (with BCs)",fontsize=14)
plt.colorbar(im3, ax=axs[1][1]);

plt.imshow ( b.reshape((n,n)), cmap="viridis", interpolation="none",origin='lower' )
plt.colorbar()
plt.title("b(x,y)",fontsize=14);

plt.imshow ( f.reshape((n,n)), cmap="viridis", interpolation="none")#,origin='lower' )
plt.colorbar()
plt.title(r"f(x,y)",fontsize=14);

'''
solve the system we setup and plot the results as a heatmap and a 3d surface.
'''

u = spsolve(a_2d.tocsr(), rhs) # cheese
U = u.reshape((n, n))

```

```
X, Y = np.meshgrid(x, y)

fig = plt.figure ( figsize = ( 12 , 10 ) , constrained_layout=True )
ax1 = fig.add_subplot ( 121 )
ax2 = fig.add_subplot ( 122 , projection = '3d' )

im1 = ax1.imshow(u.reshape((n,n)), origin="lower", extent=[0,1,0,1])
fig.colorbar(im1 , label="u(x,y)",ax=ax1 , shrink = 0.5 )
ax1.set_title("u(x,y) Heat Map")

im2 = ax2.plot_surface(X, Y, U, cmap='viridis', edgecolor="none")
fig.colorbar(im2 , shrink = 0.5 , aspect=20, label="u(x,y)")
ax2.set_xlabel("x")
ax2.set_ylabel("y")
ax2.set_zlabel("u(x,y)")
ax2.set_title("u(x,y) in 3D");
```