The Galerkin Method for Partial Differential Equations

Elijan Jakob Mastnak
Student ID: 28181157

December 2020

Contents

Assignment

1. Use the Galerkin method to solve for the velocity profile of uniform, laminar fluid flow in a pipe with a semi-circular cross section using homogeneous boundary conditions.

2. Use the Galerkin method to find the Poiseuille coefficient C (see the theory section for more detail) for the above velocity profile.

3. Investigate the dependence of the method's accuracy on the size of the indices M and N used for the Galerkin method basis functions.

4. *Optional*: Use the Galerkin method to solve the first-order wave equation

$$\frac{\partial u}{\partial t} - \frac{\partial u}{\partial \xi} = 0$$

on the interval $\xi \in [0, 2\pi]$ with the initial condition

$$u(\xi, 0) = \sin\left[\pi \cos \xi\right]$$

and the periodic boundary conditions $u(0, t) = u(2\pi, t)$.

---

1 Theory

To jump right to the solution, see Section 2.

1.1 The Problem's Relevant Hydrodynamics

Uniform, laminar flow of an incompressible fluid in a long, straight pipe under the influence of a pressure gradient p' is described by the Poisson equation

$$\nabla^2 v = -\frac{p'}{\eta},$$

where v is the component of fluid velocity along the pipe's longitudinal axis and η is the fluid's viscosity. Note that v depends only on the position in the pipe's cross section, which we will encode with the polar coordinates r and ϕ. The fluid flux in the pipe is governed by the Poiseuille equation

$$\Phi = \int_S v \, \mathrm{d}S = C\frac{p' S^2}{8\pi\eta},$$

where the coefficient C depends on the pipe's cross-sectional geometry. For a circular cross section $C = 1$, and the Poiseuille equation reduces to the familiar form

$$\Phi = \frac{p' S^2}{8\pi\eta}.$$

For more complicated cross-sectional geometries $C \neq 1$—in this report, we determine C for a pipe with a semicircular cross section of radius R.

1.2 Transition to Dimensionless Variables

For computational convenience, we replace the radius r and fluid velocity v with the dimensionless quantities

$$\xi \equiv \frac{r}{R} \qquad \text{and} \qquad u \equiv \frac{v\eta}{p'R^2}.$$

In terms of ξ and u, the problem's Poisson equation $\nabla^2 v = -p'/\eta$ and homogeneous boundary conditions simplify to

$$\nabla^2 u(\xi, \phi) = -1 \qquad \text{and} \qquad u(1, \phi) = u(\xi, 0) = u(\xi, \pi) = 0.$$

Note that for a circular cross section, homogeneous boundary conditions are concisely written $u(1, \phi) = 0$, but for a semi-circular pipe we must explicitly specify the homogeneous behavior at $\phi = 0$ and $\phi = \pi$ (along the pipe's flat edge). In terms of ξ, ϕ, and u, the semicircular pipe's Poiseuille coefficient C is

$$C = \frac{8\pi}{S^2} \int_S u \, \mathrm{d}S = \frac{32}{\pi} \int_{\xi=0}^{1} \int_{\phi=0}^{\pi} u(\xi, \phi) \xi \, \mathrm{d}\xi \, \mathrm{d}\phi,$$

where the pipe's cross-sectional area is $S = \pi/2$.

1.3 The Galerkin Method

We approximate the velocity profile $u(\xi, \phi)$ with the ansatz

$$\tilde{u}(\xi, \phi) = \sum_{i=1}^{I} a_i \Psi_i(\xi, \phi),$$

where the basis functions[1] Ψ_i should satisfy the problem's boundary conditions—in our case the homogeneous conditions $\Psi_i(1, \phi) = 0$. The approximate solution $\tilde{u}$ does not perfectly satisfy the Poisson equation—we're left with an error (or *residual*) ϵ, defined by

$$\nabla^2 \tilde{u}(\xi, \phi) = -1 + \epsilon(\xi, \phi).$$

Our goal is to find weight coefficients a_i such that the basis functions Ψ_i minimize the residual ϵ. There are many possible approaches to minimizing the residual—the *Galerkin method* takes a geometric approach. We assume the basis functions are orthogonal, with a well-defined inner product

$$\langle \Psi_i | \Psi_k \rangle = \int_V \Psi_i \Psi_k \, \mathrm{d}V = \delta_{ik},$$

and require the residual is orthogonal to the basis functions, i.e.

$$\langle \epsilon | \Psi_i \rangle = 0 \qquad \text{for } i = 1, 2, \ldots, I.$$

[1] A more appropriate name might be *trial functions*, since the $\{\Psi_i\}$ comprise only a final set and thus do not form a complete basis of the generally infinite-dimensional solution space.

Thus, at least in the scope of the I-dimensional functional subspace spanned by the $\{\Psi_i\}$, the approximate solution doesn't have any error. Next, we substitute the approximate solution $\tilde{u}$ into the differential equation $\nabla^2\tilde{u} = -1 + \epsilon$ to get

$$\sum_{i=1}^{I} a_i \nabla^2 \Psi_i = -1 + \epsilon.$$

We take the inner product of the equation from the right with Ψ_i and apply the Galerkin condition $\langle \epsilon | \Psi_i \rangle = 0$ to get

$$\sum_{j=1}^{I} a_j \left\langle \nabla^2 \Psi_j \middle| \Psi_i \right\rangle = \langle -1 | \Psi_i \rangle .$$

Finally, we write the above result as a system of equations for the coefficients a_j:

$$\sum_{j=1}^{I} \mathrm{A}_{ij}\, a_j = b_i, \qquad i = 1, 2, \ldots, I,$$

where the matrix elements A_{ij} are given by

$$\mathrm{A}_{ij} = \left\langle \nabla^2 \Psi_j \middle| \Psi_i \right\rangle \qquad \text{and} \qquad b_i = \langle -1 | \Psi_i \rangle .$$

In terms of A_{ij} and b_i, the Poiseuille coefficient C reads

$$C = -\frac{32}{\pi} \sum_i b_{ij}\, \mathrm{A}^{-1}_{ij}\, b_j = -\frac{32}{\pi} \sum_i b_i a_i. \tag{1}$$

1.4 Choosing Basis Functions

All that remains is to choose an appropriate set of basis function $\{\Psi_i\}$ suited to our problem's semicircular cross-sectional geometry. We take our inspiration from the analytic solution for $u(\xi, \phi)$, which—assuming the axis of symmetry dividing the pipe's cross section in half occurs at $\phi = \pi/2$—reads

$$u(\xi, \phi) = \sum_{m=0}^{\infty} \sum_{s=1}^{\infty} c_{mn} J_{2m+1}\big(y_{(2m+1)_s} \xi\big) \sin\big[(2m+1)\phi\big],$$

where J_{2m+1} are Bessel functions of the first kind and $y_{(2m+1)s}$ are the s-th zeros of the $(2m+1)$-th Bessel function J_{2m+1}. We choose our basis functions to be

$$\Psi_{mn}(\xi, \phi) \equiv \xi^{2m+1}(1-\xi)^n \sin\big[(2m+1)\phi\big], \tag{2}$$

where the indices take on the values $m = 0, 1, \ldots, M$ and $n = 1, 2, \ldots, N$. This choice of basis functions preserves exact solution's ϕ dependence and replaces the radial Bessel functions with the simpler expression $\xi^{2m+1}(1-\xi)^n$. The functions also satisfy the problem's homogeneous boundary conditions $\Psi_i(1, \phi) = 0$.

Caution: The basis functions depend on two indices—m and n. The index i in the expression $\{\Psi_i\}$ is really a double index—it counts both m and n. Establishing a well-defined conversion scheme between the single and double-index representations of the basis functions is key to solving this problem.

2 Solution

2.1 Implementing The Galerkin Method

The solution for both the velocity profile $u(x,t)$ and the Poiseuille coefficient C rests on solving the system of equations

$$\sum_{j=1}^{I} A_{ij} a_j = b_i, \qquad i = 1, 2, \ldots I$$

for the weight coefficients a_j. To solve the system of equations, we must first implement concrete expressions for the matrix elements and vector components

$$\mathrm{A}_{ij} = \left\langle \nabla^2 \Psi_j \middle| \Psi_i \right\rangle \qquad \text{and} \qquad b_i = \left\langle -1 | \Psi_i \right\rangle .$$

For our choice of basis functions (Eq. 2), the inner product on the domain $\xi \in [0,1]$, $\phi \in [0,\pi]$ can be written in terms of the beta function $\mathrm{B}(x,y)$ as

$$\langle \Psi_{m'n'} | \Psi_{mn} \rangle \equiv \int_0^1 \int_0^\pi \Psi_{m'n'} \Psi_{mn} \xi \, \mathrm{d}\xi \, \mathrm{d}\phi = \frac{\pi}{2} \mathrm{B}(4 + 4m, 1 + n + n') \delta_{m'm}.$$

Note that the functions Ψ_{mn} are orthogonal with respect to the index m. Again using the beta function, the matrix elements A_{ij} (or, with double indices, $\mathrm{A}_{(m'n')(mn)}$) are

$$\mathrm{A}_{(m'n')(mn)} \equiv \left\langle \nabla^2 \Psi_{mn} \middle| \Psi_{m'n'} \right\rangle = -\frac{\pi}{2} \frac{nn'(3+4m)}{2 + 4m_n + n'} \mathrm{B}(n + n' - 1, 3 + 4m) \delta_{m'm}. \quad (3)$$

Finally, the vector components $b_i \equiv b_{mn}$ are

$$b_i = \langle -1 | \Psi_{m'n'} \rangle = -\frac{2}{2m' + 1} \mathrm{B}(2m' + 3, n' + 1).$$

The nonzero matrix elements A_{ij} where $m = m'$ can be found with the Python code

```python
import numpy as np
from scipy.special import beta
def get_Aij(m, n1, n2):
    """ Returns the nonzero (m' = m) matrix element A_{ij} = A_{(mn')(mn)}"""
    return -0.5*np.pi*beta(n1+n2-1, 3+4*m)*(n1*n2*(3+4*m))/(2+4*m+n1+n2)
```

Meanwhile, the vector components b_i are found in Python with the following code:

```python
def get_bi(m, n):
    """ Returns the vector components b_{i} = b_{mn} """
    return -2*beta(2*m + 3, n + 1)/(2*m + 1)
```

2.2 Constructing Matrix and Vector Quantities

Before going further, I rewrote the system of equations

$$\sum_{j=1}^{I} A_{ij} a_j = b_i, \qquad i = 1, 2, \ldots I$$

as the matrix equation $\mathbf{A}\boldsymbol{a} = \boldsymbol{b}$, where, in terms of a single index, the vector $\boldsymbol{a}$ is

$$\boldsymbol{a} = \big[a_1, a_2, \ldots, a_I\big]^T \in \mathbb{R}^I.$$

Because of nontrivial re-indexing process, I discuss the vector $\boldsymbol{b}$ and matrix $\mathbf{A}$ in dedicated subsections below.

2.2.1 The Vector $\boldsymbol{b}$

The vector $\boldsymbol{b}$ has components

$$b_i \equiv b_{mn}, \qquad m = 0, 1, \ldots, M,\ n = 1, 2, \ldots, N.$$

There are $M+1$ values of m and N values of n, so $\boldsymbol{b}$ has $N(M+1)$ elements. In a notation that makes the re-indexing from i to (mn) clear, the vector $\boldsymbol{b}$ reads

$$\boldsymbol{b} = \Big[(b_{01}, b_{02}, \ldots, b_{0N}),\ (b_{11}, b_{12}, \ldots, b_{1N}),\ \ldots, (b_{M1}, b_{M2}, \ldots, b_{MN})\Big]^T \in \mathbb{R}^{N(M+1)}.$$

Using a loop approach, the vector $\boldsymbol{b}$ is found in Python with

```python
def get_b(M, N):
    """ Returns the vector b used to find the weight coefficients a_i """
    b = np.zeros(N*(M+1))  # preallocate
    i = 0  # collective index to count both m and n
    for m in range(0, M+1, 1):
        for n in range(1, N+1, 1):
            b[i] = get_bi(m, n)  # calculate individual vector component
            i += 1               # increment total index
    return b
```

An efficient but perhaps less intuitive vectorized approach follows in Subsection 3.1.

2.2.2 The Matrix A

The matrix $\mathbf{A}$ has elements

$$\mathrm{A}_{ij} \equiv \mathrm{A}_{(m'n')(mn)}\,.$$

There are $M+1$ values of m' and m; and N values of n' and n, so $\mathbf{A}$ has $N(M+1)$ rows and $N(M+1)$ columns. Because of orthogonality with respect to m, $\mathbf{A}$ simplifies to a block-diagonal of the form

$$\mathbf{A} = \begin{pmatrix} \tilde{\mathbf{A}}^{(0)} & & & & \\ & \tilde{\mathbf{A}}^{(1)} & & & \\ & & \ddots & & \\ & & & \tilde{\mathbf{A}}^{(m)} & \\ & & & & \ddots \\ & & & & & \tilde{\mathbf{A}}^{(M)} \end{pmatrix} \in \mathbb{R}^{N(M+1)\times N(M+1)}, \tag{4}$$

where $\tilde{\mathbf{A}}^{(m)}$ is a $N \times N$ matrix of the form

$$\tilde{\mathbf{A}}^{(m)} = \begin{pmatrix} \tilde{\mathrm{A}}^{(m)}_{11} & \tilde{\mathrm{A}}^{(m)}_{12} & \cdots & \tilde{\mathrm{A}}^{(m)}_{1N} \\ \tilde{\mathrm{A}}^{(m)}_{21} & \tilde{\mathrm{A}}^{(m)}_{22} & \cdots & \tilde{\mathrm{A}}^{(m)}_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ \tilde{\mathrm{A}}^{(m)}_{N1} & \tilde{\mathrm{A}}^{(m)}_{N2} & \cdots & \tilde{\mathrm{A}}^{(m)}_{NN} \end{pmatrix} \in \mathbb{R}^{N\times N}. \tag{5}$$

The matrix elements $\tilde{\mathrm{A}}^{(m)}_{n'n}$ are defined in Equation 3. Note that the orthogonality relation $A_{(m'n')(mn)} \propto \delta_{mm'}$ allows the use of a single index $m = m'$ for the nonzero block diagonals. I constructed $\mathbf{A}$ as follows:

1. Initialize a zero-filled $(N \cdot (M+1)) \times (N \cdot (M+1))$ matrix—because of orthogonality with respect to m, many elements will remain zero anyway.

2. For each $m = 0, 1, \ldots, M$ construct the sub-matrix $\tilde{\mathbf{A}}^{(m)}$ given in Equation 5 by iterating over $n' = 1, 2, \ldots, N$ in an outer loop and $n = 1, 2, \ldots, N$ in an inner loop, and calculating the matrix element $A_{(mn')(mn)}$ according to Equation 3.

   Place the sub-matrix $\tilde{\mathbf{A}}^{(m)}$ in the appropriate block-diagonal position of $\mathbf{A}$, as shown schematically in Equation 4.

This procedure is implemented in the following Python code:

```python
def get_A(M, N):
    """ Returns the matrix A used to find the weight coefficients a_i """
    A = np.zeros((N*(M+1), N*(M+1)))  # preallocate main matrix
    for m in range(0, M+1, 1):
        A_m = np.zeros((N, N))        # preallocate mth submatrix
        for n1 in range(1, N+1, 1):
            for n2 in range(1, N+1, 1):
                A_m[n1-1, n2-1] = get_Aij(m, n1, n2)
        A[m*N:(m+1)*N, m*N:(m+1)*N] = A_m  # place A_m in A's block diagonal
    return A
```

A more efficient vectorized implementation follows in Subsection 3.1.

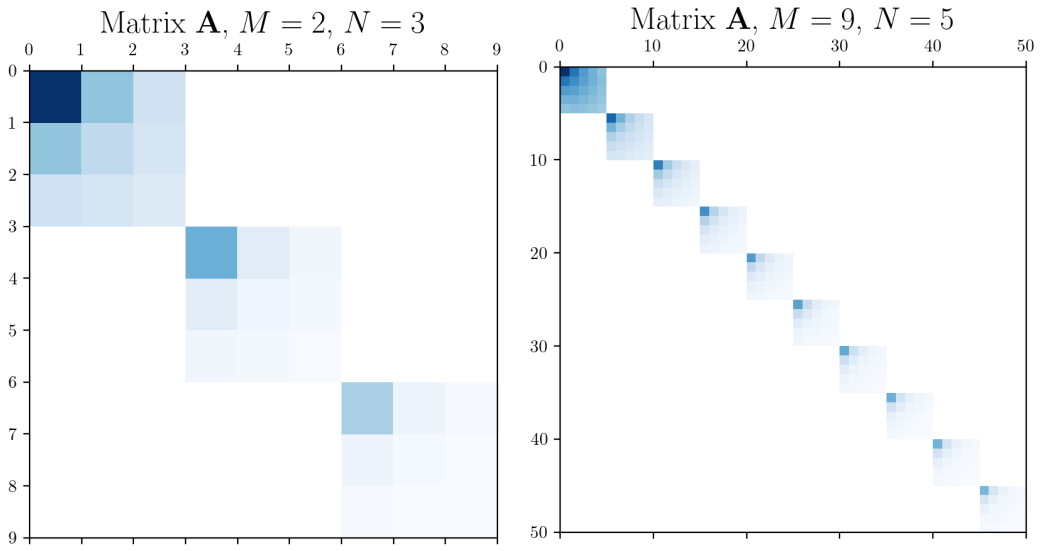

Figure 1: Visualizing the block-diagonal matrix $\mathbf{A}$ for two values of M and N—dark squares are larger in magnitude and white squares are zero. Note that the matrix elements A_{ij} in the right image are raised to the 1/3 power for stronger color.

2.3 Solutions

2.3.1 Finding the Poiseuille Coefficient C

To find the Poiseuille coefficient C, I solved the matrix equation $\mathbf{A}\boldsymbol{a} = \boldsymbol{b}$ for $\boldsymbol{a}$,[2] then calculated the coefficient C according to Equation 1, i.e.

$$C = -\frac{32}{\pi}\sum_i b_i a_i = -\frac{32}{\pi}\boldsymbol{b}\cdot\boldsymbol{a}.$$

In Python, using SciPy's optimized sparse matrix equation function `spsolve` from the package `scipy.sparse.linalg`, the coefficient C is found as follows:

```
from scipy.sparse.linalg import spsolve
from scipy.sparse import csr_matrix
# define values of M and N...
 b = get_b(M, N)
 A = get_A(M, N)
 a = spsolve(csr_matrix(A), b)  # convert A to CSR format to use spsolve
 C = - (32/np.pi)*np.dot(b, a)
```

Table 1 gives representative values of the semicircular pipe's coefficient C for various M and N, while Figure 2 shows the effect of M and N on the solution's accuracy.

M	N	C
1	1	0.7461241
10	10	0.7576178
100	100	0.7577218
150	150	0.7577220

M	N	C
1	10	0.7493260
1	150	0.7493264
10	1	0.7518211
150	1	0.7518413

Table 1: A few representative value of C for various M and N. The value at $(M, N) = (150, 150)$ is used as the reference value for estimating error in Figure 2.

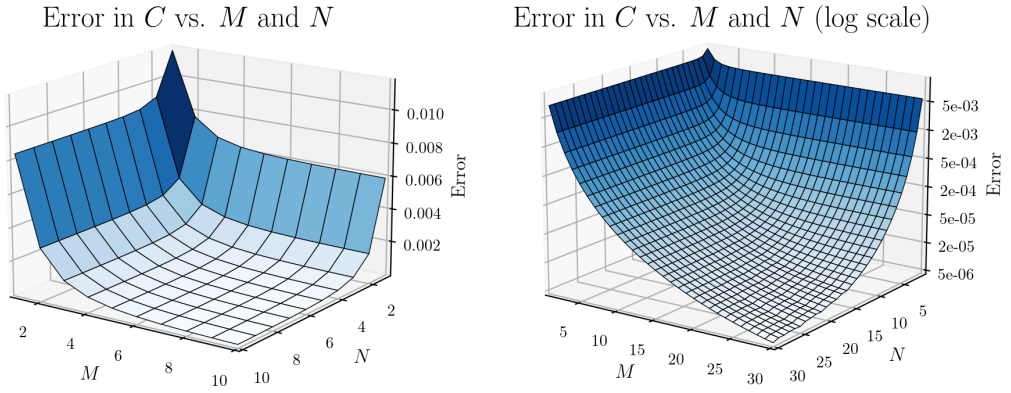

Figure 2: Error in C with respect to the reference value at $(M, N) = (150, 150)$ shown in Table 1 for various M and N in a linear (left) and log scale (right).

[2] In principle, one could also find C with $C = -\frac{32}{\pi}\boldsymbol{b}\mathbf{A}^{-1}\boldsymbol{b}$, but solving for $\boldsymbol{a}$—especially with the sparse-optimized `spsolve`—is more efficient than calculating the inverse matrix $\mathbf{A}^{-1}$.

2.3.2 Velocity Profile Solution

First, we define a position grid spanning the pipe's semicircular cross section:

$$\begin{aligned} \xi_k &= \xi_0 + k\Delta\xi, \qquad k = 0, 1, \ldots, K, \qquad [\xi_0, \xi_K] = [0, 1] \\ \phi_l &= \phi_0 + l\Delta\phi, \qquad l = 0, 1, \ldots, L, \qquad [\phi_0, \phi_L] = [0, \pi]. \end{aligned}$$

Next, for each coordinate pair (ξ_k, ϕ_l), define the 1D vector $\boldsymbol{\Psi}(\xi_k, \phi_l)$ according to

$$\begin{aligned} \boldsymbol{\Psi}(\xi_k, \phi_l) &= \big[\Psi_1(\xi_k, \phi_l), \ldots, \Psi_I(\xi_k, \phi_l)\big]^T \in \mathbb{R}^I \\ &\equiv \big[(\Psi_{01}, \Psi_{02}, \ldots, \Psi_{0N}), \ldots (\Psi_{M1}, \Psi_{M2}, \ldots, \Psi_{MN})\big]^T \in \mathbb{R}^{N\cdot(M+1)}, \end{aligned}$$

where in the second line Ψ_{mn} is implicitly evaluated at (ξ_k, ϕ_l) for conciseness. In terms of $\boldsymbol{\Psi}$, the approximate solution for velocity at the point (ξ_k, ϕ_l) is found as

$$\tilde{u}(\xi_k, \phi_l) = \sum_i^I a_i \Psi_i(\xi, \phi) = \boldsymbol{a} \cdot \boldsymbol{\Psi}(\xi_k, \phi_l)$$

I'm leaving out the Python code for brevity—see the function `get_u_loop(K, L, M, N)` in the attached `galerkin.py` file for the exact implementation. Figures 3 and 4 show the semicircular pipe's velocity profile $u(\xi, \phi)$ in a Cartesian and polar coordinate system, respectively, for $K = L = 200$ and $M = N = 25$.

The attached animation `anim-velocity.mp4` shows how the velocity profile changes with increasing M and N. I was surprised to find that even the "rough" approximation $M = N = 1$, using only two basis functions, produced a fairly accurate rendition of the result found for higher (M, N), and the effects of increasing (M, N) are essentially invisible beyond $(M, N) = (10, 10)$. Meanwhile, for comparison, using a higher-order approximation in the previous Crank-Nicolson report hugely improved the basic $r = M = 1$ approximation. (Of course, the differential Crank-Nicolson method is not exactly comparable to the finite-element-style Galerkin method.)

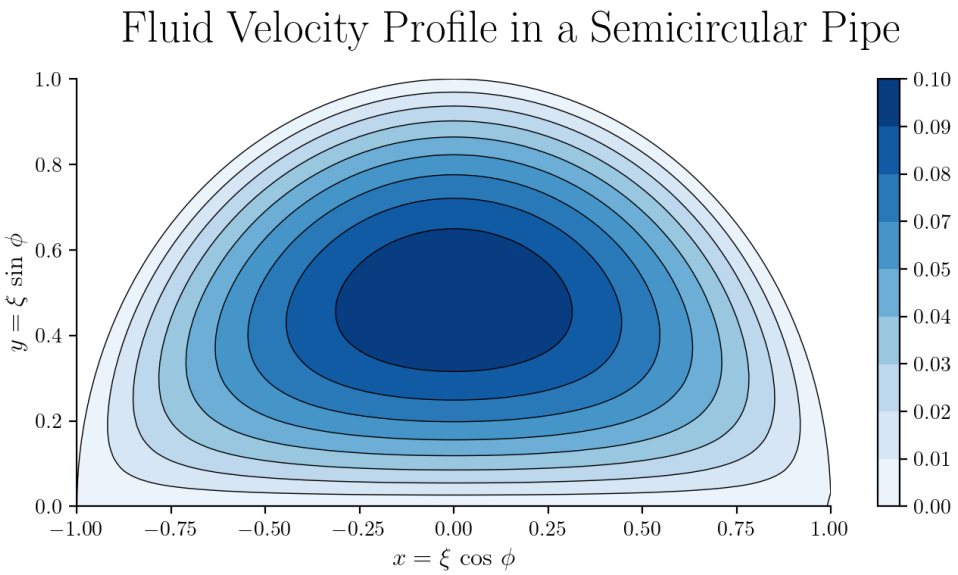

Figure 3: The fluid's velocity profile $u(\xi, \phi)$ in a Cartesian coordinate system.

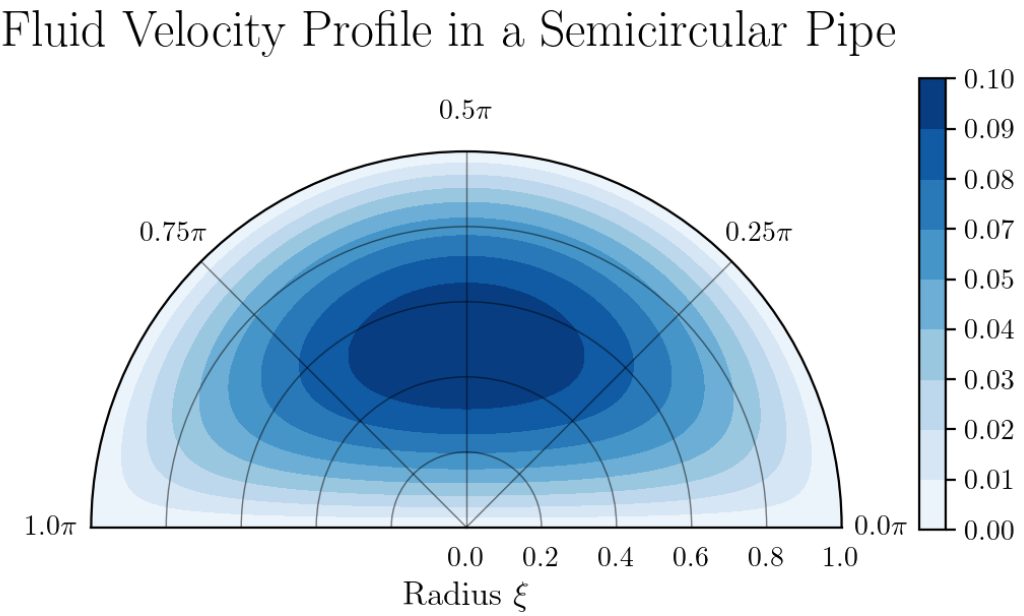

Figure 4: The fluid's velocity profile $u(\xi, \phi)$ in a polar coordinate system.

3 Vectorized Solutions

I took this assignment as an opportunity to explore vectorization using Python's Numpy library, with which I had no previous experience. Vectorization is the process of rewriting algorithms involving element-by-element calculations inside Python loops into equivalent procedures acting on entire Numpy `ndarray` objects with Numpy functions, where operations are delegated to efficient, pre-compiled `C` code.[3] The following sections describe the vectorization process and the resulting performance improvement.

3.1 The Vector b and Matrix A, Vectorized

Vectorization heavily involves using Numpy's `meshgrid` function to create multidimensional coordinate grids on which to evaluate Numpy functions. Using `meshgrid` to define a grid of m and n values, I found the vector $\boldsymbol{b}$ with the following code:

```
def get_b_vectorized(M, N):
    """ Vectorized implementation for finding the vector b """
    m = np.arange(0, M+1, 1)  # 0, 1, ..., M
    n = np.arange(1, N+1, 1)  # 1, 2, ..., N
    mm, nn = np.meshgrid(m, n, indexing="ij")  # 2D grids
    B = get_bi(mm, nn)              # returns a 2D (M+1) by N grid
    return np.reshape(B, N*(M+1))  # reshape into a 1D N*(M+1)-element vector
```

To find the matrix $\mathbf{A}$, I calculated only the non-zero block diagonal sub-matrices $\mathbf{A}^{(m)}$ (Eq. 5) and assembled $\mathbf{A}$ from the $\mathbf{A}^{(m)}$ using Scipy's `scipy.linalg.block_diag`

[3] Looping directly in Python is undesirable because Python is a dynamically-typed language, meaning that, "under the hood", the Python interpreter must check variable type in each loop iteration, significantly slowing down code execution.

function, as shown in the code block below.

```python
from scipy.linalg import block_diag  # to construct block diagonal matrix A
def get_A_vectorized(M, N):
    """ Vectorized implementation for finding the matrix A """
    n = np.arange(1, N+1, 1)
    m = np.arange(0, M+1, 1)
    mm, nn1, nn2 = np.meshgrid(m, n, n, indexing="ij")  # 3D grids
    A_vec = get_Aij(mm, nn1, nn2)  # returns a 3D grid of M+1 (N by N)
    ↪ block-diagonal sub-matrices (A^{(m)} in the report's notation)
    return block_diag(*A_vec)      # unpack submatrices and construct A
```

Figure 5 shows the computation time required to assemble the matrix $\mathbf{A}$ as a function of the single index $I = N(M+1)$, where I used $M = N$ for simplicity. Note the 40-fold improvement in performance thanks to vectorization.

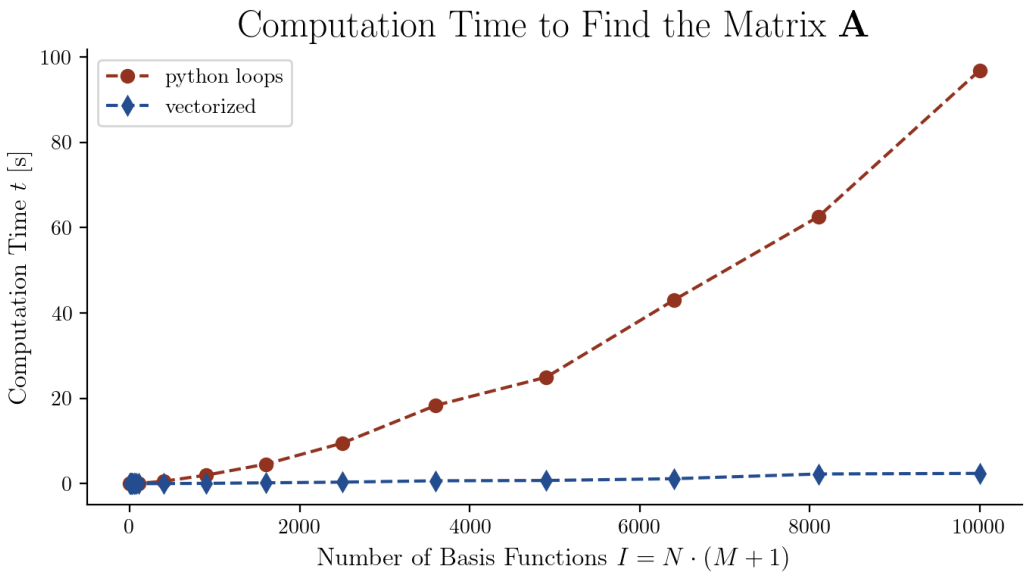

Figure 5: Computation time, averaged over 10 runs, required to calculate the matrix $\mathbf{A}$ using "vanilla" Python loops compared to an optimized vectorized approach.

3.2 Defining Higher-Dimensional Grids for Vectorization

This section, besides taking advantage of optimized Numpy's functions, also serves as a good exercise in manipulating two and three-dimensional grids. As in Section 2, we partition the pipe's cross section into the two-dimensional grid

$$\begin{aligned} \xi_k &= \xi_0 + k\Delta\xi, \qquad k = 0, 1, \ldots, K, \qquad [\xi_0, \xi_K] = [0, 1] \\ \phi_l &= \phi_0 + l\Delta\phi, \qquad l = 0, 1, \ldots, L, \qquad [\phi_0, \phi_L] = [0, \pi]. \end{aligned}$$

As a reference for the code that will follow, we create the position grid in Python via

```
# define K, L...
xi0, xiK = 0.0, 1.0                  # max and min scaled radius
xi = np.linspace(xi0, xiK, K+1)
phi0, phiL = 0.0, np.pi              # max and min polar angle
phi = np.linspace(phi0, phiL, L+1)
```

For review, recall that for each (ξ_k, ϕ_l), we defined the 1D vector $\boldsymbol{\Psi}(\xi_k, \phi_l)$ as

$$\boldsymbol{\Psi}(\xi_k, \phi_l) = \left[\Psi_1(\xi_k, \phi_l), \ldots, \Psi_I(\xi_k, \phi_l)\right]^T \in \mathbb{R}^I \equiv \mathbb{R}^{N\cdot(M+1)}.$$

In the vectorized approach, we generalize the vector $\boldsymbol{\Psi}$ to a three-index object

$$\tilde{\boldsymbol{\Psi}} \in \mathbb{R}^{(K+1)\times(L+1)\times N\cdot(M+1)},$$

which holds the one-dimensional vector $\boldsymbol{\Psi}(\xi_k, \phi_l) \in \mathbb{R}^{N\cdot(M+1)}$ at each two-dimensional coordinate (ξ_k, ϕ_l). I found $\tilde{\boldsymbol{\Psi}}$ with the following code:

```
""" Purely vectorized approach for finding the 3D grid Psi (slower) """
# define M, N...
m = np.arange(0, M+1, 1)
n = np.arange(1, N+1, 1)
XI, PHI, mm, nn = np.meshgrid(xi, phi, m, n, indexing="ij")
Psi = get_Psi_mn(XI, PHI, mm, nn)  # 4D grid
Psi = np.reshape(Psi_vec, (K+1, L+1, N*(M+1)))  # reshape to 3D
```

The function `get_Psi_mn` is a direction implementation of Equation 2. I assume creating four 4D coordinate grids in the above code (understandably) outweighed vectorization benefits, because I found the hybrid approach below, which combines vectorization and Python loops, performed faster by a factor of roughly 2.

```
""" Hybrid (and faster) vectorized approach for finding the 3D grid Psi """
mm, nn = np.meshgrid(m, n, indexing="ij")      # 2D grid
Psi = np.zeros((K+1, L+1, N*(M+1)))            # preallocate
for k, xi_k in enumerate(xi):                  # loop over xi
    for l, phi_l in enumerate(phi):            # loop over phi
        psi = get_Psi_mn(mm, nn, xi_k, phi_l)  # vectorized
        psi = psi_vec.flatten()                # convert 2D to 1D
        Psi[k][l] = psi_vec  # place 1D psi in 3D Psi
```

Finally, we generalize the scalar solution $\tilde{u}(\xi, \phi)$ to a two-dimensional solution matrix $\mathbf{U} \in \mathbb{R}^{(K+1)\times(L+1)}$, which takes the form

$$\mathbf{U} = \begin{pmatrix} \tilde{u}(\xi_0, \phi_0) & \tilde{u}(\xi_0, \phi_1) & \cdots & \tilde{u}(\xi_0, \phi_L) \\ \tilde{u}(\xi_1, \phi_0) & \tilde{u}(\xi_1, \phi_1) & \cdots & \tilde{u}(\xi_1, \phi_L) \\ \vdots & \vdots & \ddots & \vdots \\ \tilde{u}(\xi_K, \phi_0) & \tilde{u}(\xi_K, \phi_1) & \cdots & \tilde{u}(\xi_K, \phi_L) \end{pmatrix} \in \mathbb{R}^{(K+1)\times(L+1)},$$

where $\tilde{u}$ is the approximate solution for fluid velocity u at a point (ξ_k, ϕ_l).

3.3 Vectorized Velocity Profile Solution

In terms of $\tilde{\mathbf{\Psi}}$ and the weight coefficient vector $\boldsymbol{a}$, we can concisely find $\mathbf{U}$ with

$$\mathbf{U} = \tilde{\mathbf{\Psi}} \bullet \boldsymbol{a}$$

where $\bullet$ denotes the multiplication of the 3D object $\tilde{\mathbf{\Psi}} \in \mathbb{R}^{(K+1)\times(L+1)\times N\cdot(M+1)}$ with the 1D vector $\boldsymbol{a} \in \mathbb{R}^{N\cdot(M+1)}$ to yield the two-dimensional solution matrix $\mathbf{U} \in \mathbb{R}^{(K+1)\times(L+1)}$. Having found $\boldsymbol{b}$, $\mathbf{A}$, and $\boldsymbol{a}$, we can use Numpy's `np.dot` function to calculate the solution matrix $\mathbf{U}$ in a single line of code:

```
b = get_b_vectorized(M, N)
A = get_A_vectorized_full(M, N)
a = spsolve(csr_matrix(A), b)    # convert A to CSR for use with spsolve
U = np.dot(Psi, a)               # nice "one-liner"!
```

Figure 6 show the performance improvement resulting from a vectorized computation of $u(\xi, \phi)$, again using the single index $I = N(M+1)$ with $M = N$. Table 2 shows the same results in tabular form, to better show the concrete numerical values.

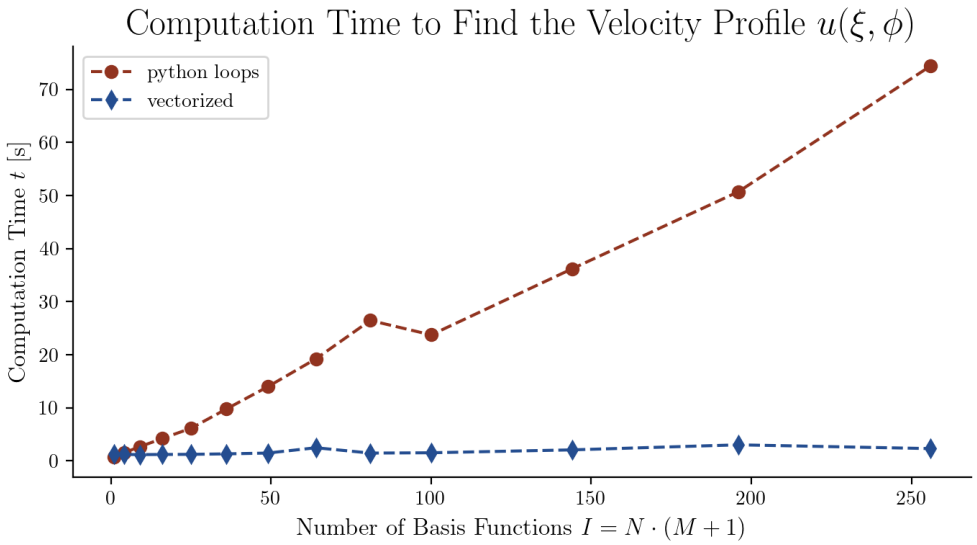

Figure 6: Computation time, averaged over 10 runs, required to calculate the velocity profile $u(\xi, \phi)$ using "vanilla" Python loops compared to a vectorized approach.

M	N	t_{vec}	t_{loop}
1	1	3.12 ms	0.30 ms
10	10	5.57 ms	77.1 ms
50	50	0.36 s	9.50 s
100	100	2.41 s	96.9 s

(a) Times when finding $\mathbf{A}$

M	N	t_{vec}	t_{loop}
1	1	1.22 s	0.66 s
5	5	1.25 s	6.13 s
10	10	1.52 s	23.7 s
15	15	2.31 s	74.4 s

(b) Times when finding u

Table 2: A few representative values of the computation time t required to calculate the matrix $\mathbf{A}$ (left) and velocity profile u (right) with a vectorized and loop approach. Note that the loop approach is actually faster for small (M, N).

4 First-Order Wave Equation

4.1 Implementing the Galerkin Method

For the first-order wave equation, we use the Galerkin ansatz

$$\tilde{u}(x,t) = \sum_{k=-N/2}^{N/2} a_k(t)\Psi_k(x) = \frac{1}{\sqrt{2\pi}} \sum_{k=-N/2}^{N/2} , a_k(t)e^{ikx}$$

where the basis functions are plane waves of the form

$$\Psi_k(x) = \frac{e^{ikx}}{\sqrt{2\pi}}. \tag{6}$$

The inner product between basis functions is

$$\langle \Psi_k | \Psi_j \rangle = \int_0^{2\pi} \Psi_k^*(x)\Psi_j(x)\,\mathrm{d}x = \frac{1}{2\pi}\int_0^{2\pi} e^{-ikx}e^{ijx}\,\mathrm{d}x = \delta_{jk}.$$

Substituting the Galerkin ansatz into the differential equation and taking the inner product of the equation leads to the Galerkin condition

$$\int_0^{2\pi} \left[\frac{\partial \tilde{u}}{\partial t} - \frac{\partial \tilde{u}}{\partial x}\right] \Psi_k^*(x)\,\mathrm{d}x = 0, \qquad \text{for } k = -\frac{N}{2}, \ldots, \frac{N}{2}.$$

We then use the Galerkin condition and orthonormality of the basis functions to get a differential equation for the coefficients $a_k(t)$:

$$\frac{\mathrm{d}a_k}{\mathrm{d}t} - ika_k = 0, \qquad \text{for } k = -\frac{N}{2}, \ldots, \frac{N}{2}. \tag{7}$$

Here we have two options:

1. Find the coefficients $a_k(t)$ with the analytic solution

$$a_k(t) = \sin\left(\frac{k\pi}{2}\right) J_k(\pi)e^{ikt}, \tag{8}$$

   where J_k is the Bessel function of the first kind.

2. Find the initial coefficients $a_k(0)$ with the Fourier transform

$$a_k(0) = \int_0^{2\pi} u(x,0)\Psi_k^*(x)\,\mathrm{d}x = \frac{1}{\sqrt{2\pi}}\int_0^{2\pi} u(x,0)e^{-ikx}\,\mathrm{d}x,$$

   where $u(x,0) = \sin\left[\pi\cos x\right]$ is the known initial condition (shown in Figure 7), and propagate the initial coefficients through time using either the analytic solution $a_k(t) = a_k(0)e^{ikt}$ or numerically using Equation 7.

Out of laziness, I opted for the analytic solution in Equation 8. With the $a_k(t)$ known, we find the solution $u(x,t)$ with

$$\tilde{u}(x,t) = \frac{1}{\sqrt{2\pi}} \sum_{k=-N/2}^{N/2} a_k(t)e^{ikx}.$$

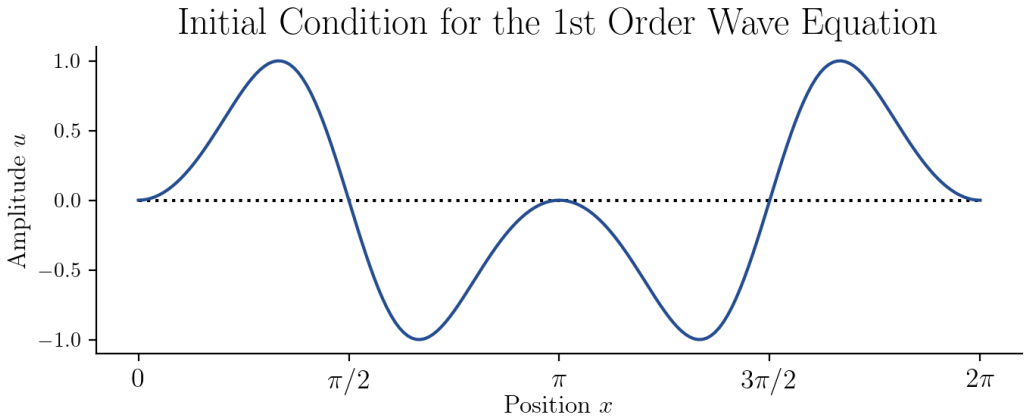

Figure 7: The initial condition used when solving the first-order wave equation.

4.2 Position, Time, and "Frequency" Grids

First, we define a position grid of $N+1$ points spanning $x \in [0, 2\pi]$ and a time grid of $M+1$ points spanning t from 0 to some arbitrary final simulation time, e.g. an integer multiple of π.

$$\begin{aligned} x_n &= x_0 + n\Delta x, \qquad n = 0, 1, \ldots, N, \qquad [x_0, x_N] = [0, 2\pi] \\ t_m &= t_0 + m\Delta t, \qquad m = 0, 1, \ldots, M, \qquad [t_0, t_M] = [0, \mathcal{N}\pi], \quad \mathcal{N} \in \mathbb{N}. \end{aligned}$$

Although this report does not use a purely spectral approach, it is nonetheless worth noting that the term k is analogous to a "frequency" for sampling position x. The corresponding sample rate f_s and Nyquist frequency f_c are

$$f_s = \frac{N}{x_N - x_0} = \frac{1}{\Delta x} \qquad \text{and} \qquad f_c = \frac{f_s}{2} = \frac{N}{2(x_N - x_0)}.$$

With the Nyquist frequency in mind, I generate a grid of $N+1$ values of k spanning

$$k \in \left\{-\tfrac{N}{2}, -\tfrac{N}{2}+1, \ldots, \tfrac{N}{2}\right\}.$$

4.3 Vector and Matrix Quantities

I used a vectorized approach for this problem. First, define the vector $\boldsymbol{a}(t)$ as

$$\boldsymbol{a}(x) = \left[a_{-N/2}(t), \ldots, a_{N/2}(t)\right]^T \in \mathbb{C}^{N+1}.$$

We then generalize $\boldsymbol{a}$ to the coefficient matrix $\mathbf{A} \in \mathbb{C}^{M\times N}$, which holds $\boldsymbol{a}^T(t)$ as a row at each time t. The matrix $\mathbf{A}$ is defined as

$$\mathbf{A} = \begin{pmatrix} a_{-N/2}(t_0) & \cdots & a_{N/2}(t_0) \\ a_{-N/2}(t_1) & \cdots & a_{N/2}(t_1) \\ \vdots & \ddots & \vdots \\ a_{-N/2}(t_M) & \cdots & a_{N/2}(t_M) \end{pmatrix} \in \mathbb{C}^{(M+1)\times(N+1)},$$

where the matrix elements are found with Equation 8. Figure 8 shows the matrix $\mathbf{A}$ for $M = 30$ and $N = 50$. Next, define the vector Ψ_k, given by

$$\boldsymbol{\psi}(x) = \left[\Psi_{-N/2}(x), \ldots, \Psi_{N/2}(x)\right]^T \in \mathbb{C}^{N+1},$$

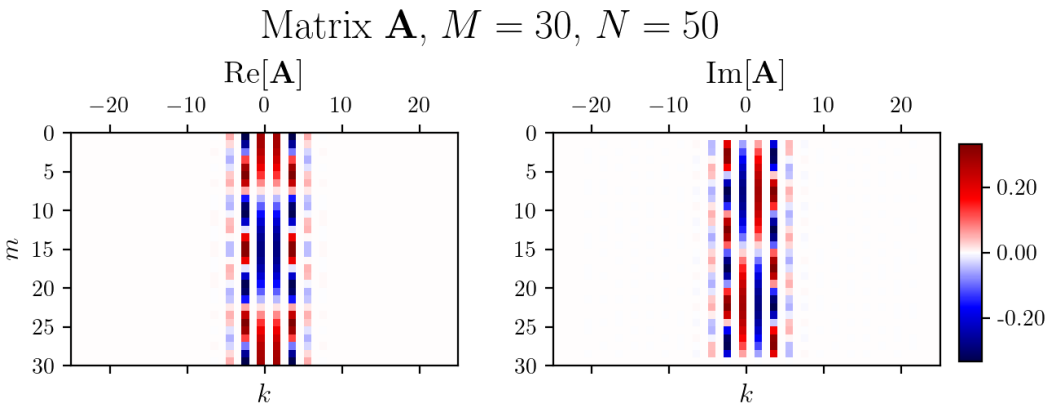

Figure 8: Visualizing the complex matrix $\mathbf{A}$ used with the first-order wave equation. Only central k values are significant, an effect of the $J_k(\pi)$ term in Equation 8.

and the associated matrix quantity

$$\mathbf{\Psi} = \begin{pmatrix} \Psi_{-N/2}(x_0) & \cdots & \Psi_{-N/2}(x_N) \\ \vdots & \ddots & \vdots \\ \Psi_{N/2}(x_0) & \cdots & \Psi_{N/2}(x_N) \end{pmatrix} \in \mathbb{C}^{(N+1)\times(N+1)}.$$

The matrix elements are found with the known basis functions in Equation 6. Note that within $\mathbf{\Psi}$, the index k changes across rows, while k changes across columns in the matrix $\mathbf{A}$.

4.4 Solution

First, we define the solution vector $\boldsymbol{u}(t)$, which holds $u(x)$ at a fixed time t, as

$$\boldsymbol{u}(t) = \big[u(x_0,t), u(x_1,t), \ldots, u(x_N,t)\big]^T \in \mathbb{C}^{N+1}.$$

The associated solution matrix $\mathbf{U} \in \mathbb{C}^{(M+1)\times(N+1)}$ is

$$\mathbf{U} = \begin{pmatrix} u(x_0,t_0) & \cdots & u(x_N,t_0) \\ u(x_0,t_1) & \cdots & u(x_N,t_1) \\ \vdots & \ddots & \vdots \\ u(x_0,t_M) & \cdots & u(x_N,t_M) \end{pmatrix} \in \mathbb{C}^{(M+1)\times(N+1)}.$$

The matrix $\mathbf{U}$ holds the solution $\boldsymbol{u}^T(t)$ as a row at each time t. In terms of $\mathbf{A}$ and Ψ, the solution matrix is found concisely with the matrix product

$$\mathbf{U} = \mathbf{A}\mathbf{\Psi}.$$

Figures 9 and 10 show the time-dependent solution $u(x,t)$ in two and three dimensions, respectively. In my opinion, neither does a satisfactory job of conveying the actual behavior, which is best visualized with the attached animation `anim-wave.mp4`.

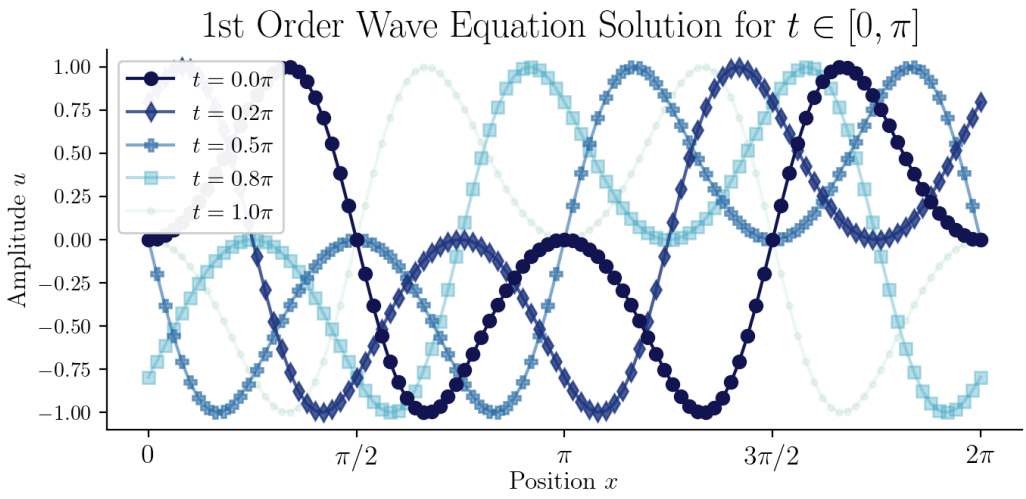

Figure 9: The solution to the first-order wave equation over a half-period. The wave travels left along the x axis and preserves the shape of the initial condition in Figure 7. This is best visualized in the attached animation `anim-wave.mp4`.

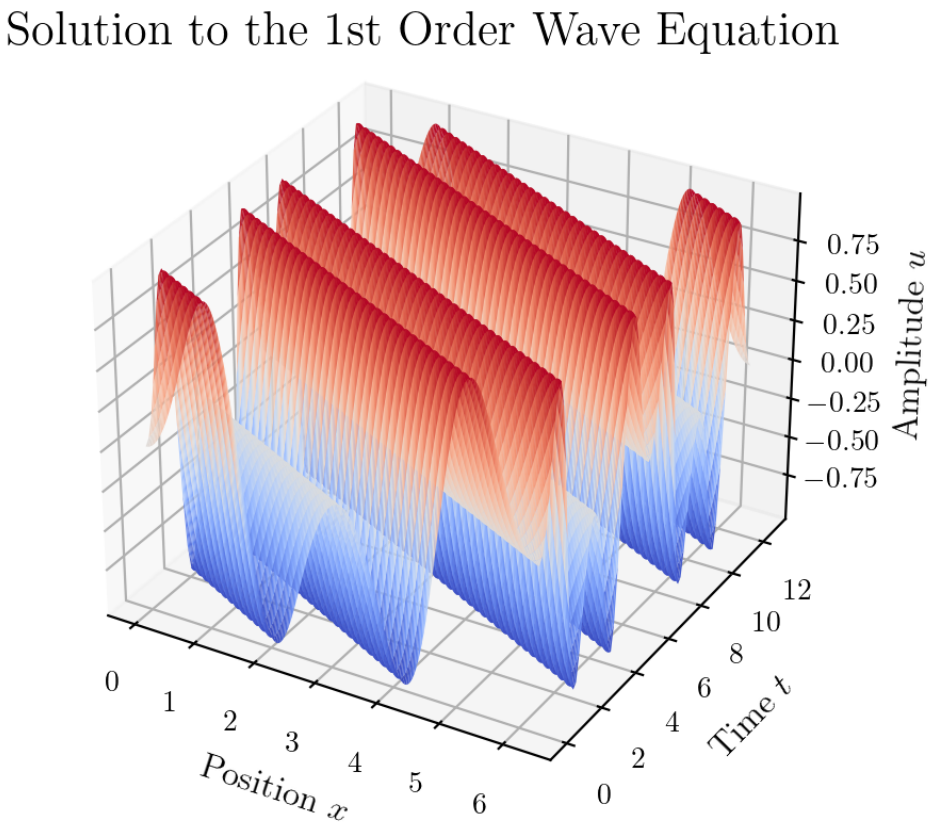

Figure 10: A questionably successful attempt at visualizing the solution to the first-order wave equation in 3D—see the attached `anim-wave.mp4` for a better result.