Computational Notebook

Clancy Andrews

Autumn 2025

Contents

Introduction

The intention of this notebook is to give a complete overview of the scientific computing methods available for solving ordinary differential equation, partial differential equations, and systems of equations. The idea when solving these equations is to find ways to represent high level mathematics in terms of something that a computer can understand. We will address three specific problems, Van der Pol Oscillator, Quantum Harmonic Oscillator, and Reaction-Diffusion, with observations into the methods employed to solve them.

There are various sources of information on the topic of scientific computing. Textbooks on the topic, including but not limited to those done by Heath (2018) and Kincaid & Cheney (2009), give a deep understanding to mathematical computing topics, with introductory material, to numerical methods for solving differential equations.

While scientific computing has just started taking off with the introduction of the computer, problems and methods for solving complex equations have been identified and approached centuries before. Gustafsson (2018) addressed these topics in the textbook Scientific Computing, A Historical Perspective, where the idea of computation was more or less achieved before the invention of computers, and how it evolved after the fact. We can observe a method that was developed and was computationally complex before the utilization of computers, which was the simplex method for linear optimization. Bertsimas and Tsitsiklis (1997) demonstrate the simplex algorithm using a tableau, which can be established and carried out by hand for smaller linear programming problems. The advent of the computer has since allowed this method to be achieved with extreme precision and efficiency for much larger formulations.

Work in the field has not slowed down, with more complex problems being addressed and solved today. The paper "Parallel matching-based AMG preconditioners for elliptic equations discretized by IgA", written by D'Ambra, Durastante, and Filippone (2025), is one example of improvements and capabilities made possible by high-performance computing, and efficient methods for solving problems in the field. In their work, D'Ambra, Durastante, and Filippone (2025) implement Krylov subspace methods, with parallel-computing frameworks, to efficiently solve discretized elliptic boundary value problems. Another example can be found by the work of Li, Di, and Jiang (2025), for which they use a Two-Step Generalized RBF Finite Difference method to solve partial differential equations on manifolds. With both of these examples, it is quite evident that the field of scientific computing has developed substantially following major improvements in methodological refinement and technological advancement.

Methods

There are various methods we can employ to solve problems. The following sections will be a brief overview of methods for solving initial and boundary value problems, with attention to their properties and stability.

Methods for Inital Value Problems

Initial value problems arise when differential equation is paired with an initial state at a specific starting time. The methods that we will address will work utilize the initial value of the equation at the starting time t_0, which will need to be derived or assumed. There are two main types of methods in this section. Those are single-step and multi-step methods. The idea of single-step methods is to use an adjacent point in time to calculate the next step in the series. Multi-step methods are not restricted to just the immediate, adjacent point in time. They will often use points two or more steps away. The Adams-Bashforth and Adams-Moulton methods are the only two multi-step methods that we will go over, the rest with be single-step.

Forward Euler

The Forward Euler method is defined as follows

$$y_{n+1} = y_n + \Delta t f(t_n, y_n)$$

where $f(t_n, y_n) = \frac{dy_n(t)}{dt_n}$. We can observe that the method is explicit, and once derived with the Taylor Expansion, has local error of $O(\Delta t^2)$ and a global error of $O(\Delta t)$.

Backward Euler

The Backwards Euler method is defined as follows

$$y_n = y_{n+1} - \Delta t f(t_{n+1}, y_{n+1})$$

where $f(t_{n+1}, y_{n+1}) = \frac{dy_{n+1}(t)}{dt_{n+1}}$. We can observe that this method has the same local and global error as the Forward Euler method. Since the Backward Euler method is implicit, the initial y_{n+1} terms can be determined using an different explicit method, i.e., Forward Euler.

Heun (Average)

The Heun method is defined as follows

$$y_{n+1} = y_n + \frac{\Delta t}{2}\Big[f(t_n, y_n) + f(t_{n+1}, y_{n+1})\Big]$$

where $f(t_n, y_n) = \frac{dy_n(t)}{dt_n}$.

Runge-Kutta 2 (Midpoint)

The Runge-Kutta 2 method is defined as follows

$$y_{n+1} = y_n + \Delta t\left(f\left[t_n + \frac{\Delta t}{2}, y\left(t_n + \frac{\Delta t}{2}\right)\right]\right)$$

where $f(t_n, y_n) = \frac{dy_n(t)}{dt_n}$.

Runge-Kutta 4

The Runge-Kutta 4 method is defined as follows

$$y_{n+1} = y_n + \frac{\Delta t}{6}\left(F_1 + 2F_2 + 2F_3 + F_4\right)$$

where

$$\begin{aligned}
F_1 &= f(t_n, y_n) = \frac{dy_n(t)}{dt_n}, \\
F_2 &= f\left(t_n + \frac{\Delta t}{2}, y_n + \frac{\Delta t}{2}F_1\right), \\
F_3 &= f\left(t_n + \frac{\Delta t}{2}, y_n + \frac{\Delta t}{2}F_2\right), \\
F_4 &= f\left(t_n + \Delta t, y_n + \Delta t F_3\right).
\end{aligned}$$

Test

Adams-Bashforth

The Adams-Bashforth method is defined as follows

$$y_{n+1} = y_n + \frac{\Delta t}{2}\Big(3f_n - f_{n-1}\Big)$$

where $f_n = f(t_n, y_n) = \frac{dy_n(t)}{dt_n}$.

Adams-Moulton

The Adams-Moulton method is defined as follows

$$y_{n+1} = y_n + \frac{\Delta t}{2}\Big(f_{n+1} - f_n\Big)$$

where $f_n = f(t_n, y_n) = \frac{dy_n(t)}{dt_n}$.

Methods for Boundary Value Problems

Another set of problems are boundary value problems. Unlike initial value problems, BVPs require two or more initial conditions, which will need to be derived or assumed. Working with differential equations allows us to implement different boundary conditions, some of which are Neumann boundary conditions, Dirichlet boundary conditions, and Robin (mixed) boundary conditions. Neumann boundary conditions have us use the boundary values of our derivative function $f'(t_0) = \alpha$ and $f'(t_n) = \beta$ where Dirichlet boundary conditions $f(t_0) = \alpha$ and $f(t_n) = \beta$. For Robin boundary conditions, we get to use a linear combination of the Neumann and Dirichlet boundary conditions, and it will be defined as

$$\alpha_1 f'(t_0) + \beta_1 f(t_0) = \gamma_1$$

and

$$\alpha_2 f'(t_n) + \beta_2 f(t_n) = \gamma_2.$$

Finite Difference

Text for finite difference using central, forward, backward difference

Central Difference Text for central difference

Forward Difference Text for forward difference

Backward Difference Text for backward difference

Spectral

Text FFT info

Methods of Lines

Methods of lines like leap frog and cd

Finite Difference (1,2)

Text

Leap Frog (2,2)

Text

Leap Frog (2,4)

Text

Methods for Systems of Equations

While we might know some methods for solving systems of equations, i.e., Gaussian Elimination, we will also address some other methods that perform with better efficiency for problems with higher computational complexity.

Gaussian Elimination

Text

Backward Substitution

Text

LU Decomposition

Text

Jacobi

Text

Gauss-Seidel

Text

Problems

With the methods established above, we can now proceed with solving the three problems mentioned at the beginning: Van der Pol Oscillator, the Quantum Harmonic Oscillator, and the Reaction-Diffusion equation.

Van der Pol Oscillator

The Van der Pol Oscillator is a second-order ordinary differential equation, defined as follows

$$\frac{d^2y(t)}{dt^2} + \epsilon\Big[y^2(t) - 1\Big]\frac{dy(t)}{dt} + y(t) = 0$$

with ϵ being our dampening parameter. By observation, we can see that the equation is nonlinear, and our ϵ parameter works to help amplify or decay our nonlinear portion $y^2(t) - 1$. Since we want ϵ to be small, we can set $\epsilon = 0.1$. We also want to setup our discretization of t, as well as our initial conditions $y_0' = y(0) = 2$ and $y_0' = \frac{dy(t_0=0)}{dt} = \pi^2$.

```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

### Initial Conditions
epsilon = 0.1
tspan = np.arange(0, 32 + 1e-6, 0.1)
y0 = [2, np.pi**2]
```

In order to start solving this problem, we will want to put the equation in canonical form. We can start by defining the following:

$$\begin{aligned} w_1 &= y'(t), \\ w_2 &= y''(t) = w_1'(t). \end{aligned}$$

With our new variables, we can rearrange the original equation:

$$\begin{aligned} \frac{d^2y(t)}{dt^2} + \epsilon\Big[y^2(t) - 1\Big]\frac{dy(t)}{dt} + y(t) &= 0 \\ \frac{d^2y(t)}{dt^2} &= -\epsilon\Big[y^2(t) - 1\Big]\frac{dy(t)}{dt} - y(t) \\ \frac{dw_2}{dt} &= -\epsilon\Big[w_1^2 - 1\Big]w_2 - w_1. \end{aligned}$$

With the canonical form defined, we can establish our right-hand side function.

```
def Van_der_Pol(t, y, epsilon):
  """
  Implementation of the Van der Pol Oscillator
  """
  w1, w2 = y
  return w2, -epsilon * ((w1**2)-1) * w2 - w1
```

With our initial conditions established and our right-hand side function setup, we can now go about solving our initial value problem using `scipy.integrate.solve_ivp`.

```
### Solution of the Van der Pol Oscillator over t
sol = solve_ivp(Van_der_Pol, [tspan[0], tspan[-1]], y0, \
                args = [epsilon], t_eval = tspan)

t = sol.t
y1 = sol.y[0,:]       # Grabs the solutions to y'(t)
y2 = sol.y[1,:]       # Grabs the solutions to y''(t)

plt.figure(1)
plt.plot(t, y1, label = r"$y'(t)$")
plt.plot(t, y2, label = r"$y''(t)$")
plt.title(r"Van der Pol Oscillator with $\epsilon = 0.1$")
plt.xlabel(r"$t$", fontsize = 12)
plt.ylabel(r"$y(t)$", fontsize = 12)
plt.legend()
plt.show()
```

We can observe in Figure 1 that between $t = 0$ and $t = 4$, there is a bit of inconsistency in the approximation of $y''(t)$. To counteract this inconsistency, we can try decreasing our value of ϵ to 0.01.

```
### Same solution with different epsilon
epsilon = 0.01
sol = solve_ivp(Van_der_Pol, [tspan[0], tspan[-1]], y0, \
                args = [epsilon], t_eval = tspan)

t = sol.t
y1 = sol.y[0,:]       # Grabs the solutions to y'(t)
y2 = sol.y[1,:]       # Grabs the solutions to y''(t)

plt.figure(2)
plt.plot(t, y1, label = r"$y'(t)$")
plt.plot(t, y2, label = r"$y''(t)$")
```

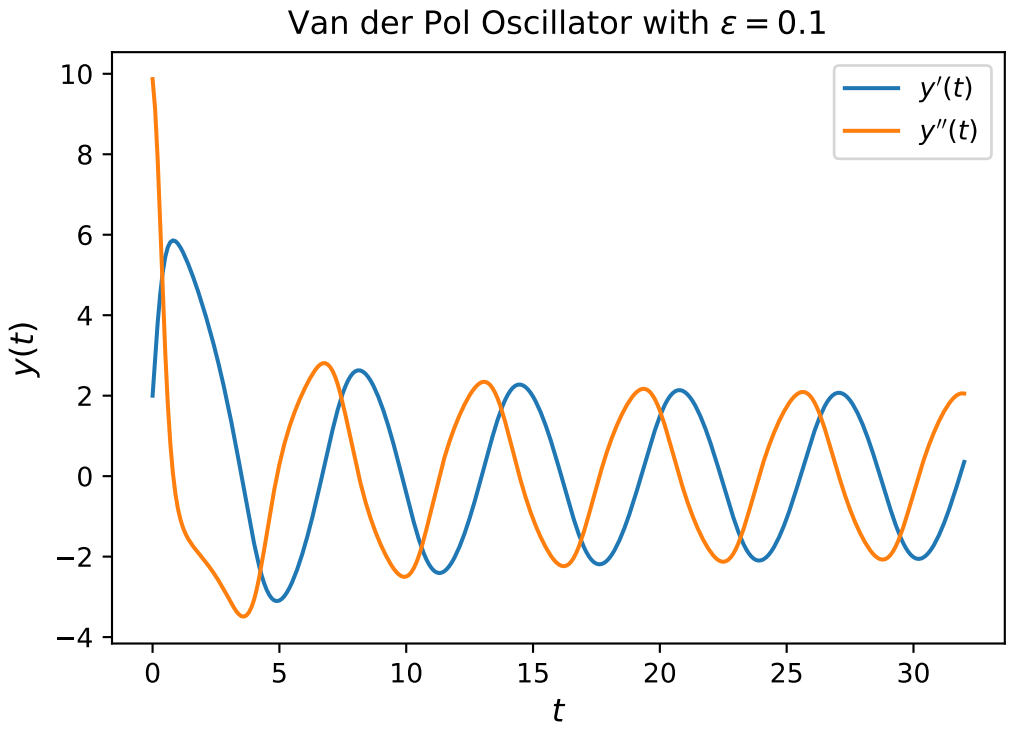

Figure 1: Solution of the Van der Pol Oscillator with $t \in [0, 32]$ and $\epsilon = 0.1$. Notice the instability for t on the interval $[0, 4]$.

```
plt.title(r"Van der Pol Oscillator with $\epsilon = 0.01$")
plt.xlabel(r"$t$", fontsize = 12)
plt.ylabel(r"$y(t)$", fontsize = 12)
plt.legend()
plt.show()
```

It is important to note that the `solve_ivp` function uses the Runge-Kutta 4 method by default. Changing the `method` parameter will allow us to utilize different methods, some of which may be better to use, depending on the problem.

Quantum Harmonic Oscillator

Text

Reaction-Diffusion Equations

The Reaction-Diffusion equation helps to solve for and find phenomena when one or more substances change over space and time. Those equations are defined as follows

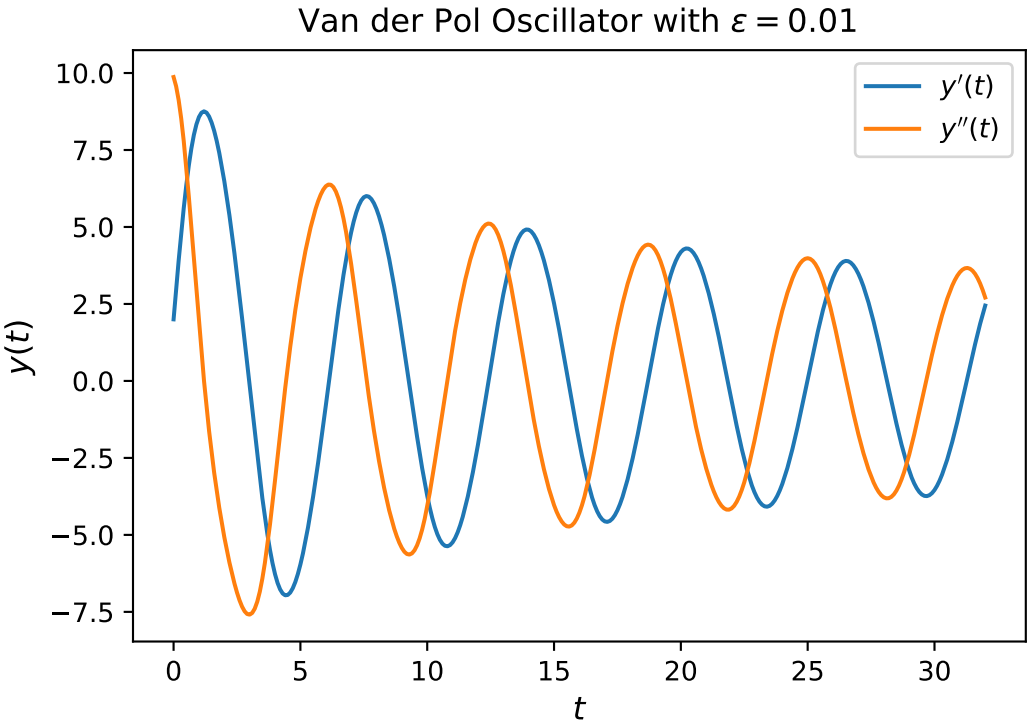

Figure 2: Solution of the Van der Pol Oscillator with $t \in [0, 32]$ and $\epsilon = 0.01$. We can now notice a more stable solution for t on the interval $[0, 4]$.

$$U_t = \lambda(A)U - \omega(A)V + D_1\nabla^2 U$$
$$V_t = \omega(A)U - \lambda(A)V + D_2\nabla^2 V$$

where $A^2 = U^2 + V^2$ and $\nabla^2 = \partial_x^2 + \partial_y^2$. In this formulation, we will look at the system where

$$\lambda(A) = 1 - A^2,$$
$$\omega(A) = -\beta A^2.$$

We will now investigate the solutions to this system formulation using two different methods, those being the spectral fast fourier transform method with periodic boundary conditions and the Chebyshev polynomial method with Dirichlet boundary conditions.

For the first solution, we will being by implementing the initial conditions for the spectral fast fourier transform. We will assume that $x, y \in [-10, 10]$, $n = 64$, $\beta = 1$, $D_1 = D_2 = 0.1$, $m = 1$, and time spanning 0 to 4 with time steps of 0.5.

```
import numpy as np
from scipy.integrate import solve_ivp
from numpy.fft import fft2, ifft2
from math import pi
```

```
n = 64
beta = 1
D1 = D2 = 0.1
m = 1            # Number of spirals
L = 10

tspan = np.arange(0, 4 + 1e-6, 0.5)
xspan = np.linspace(-L, L, n+1)
x = xspan[:n]
y = x

X,Y = np.meshgrid(x,y, indexing='xy')

u = np.tanh(np.sqrt(X**2 + Y**2)) \
    * np.cos(m * np.angle(X + 1j*Y) - (np.sqrt(X**2 + Y**2)))
v = np.tanh(np.sqrt(X**2 + Y**2)) \
    * np.sin(m * np.angle(X + 1j*Y) - (np.sqrt(X**2 + Y**2)))

ut = fft2(u)
vt = fft2(v)

n2 = n*n

ut_flat = ut.reshape(n2, order='F')
vt_flat = vt.reshape(n2, order='F')

uv = np.concatenate([
    np.real(ut_flat), np.imag(ut_flat),
    np.real(vt_flat), np.imag(vt_flat)
])

# Spectral domain initialization
kx = (2*np.pi/(2*L)) \
      * np.concatenate((np.arange(0, n//2),
                        np.arange(-n//2, 0)))
kx[0] = 1e-6
ky = kx

KX, KY = np.meshgrid(kx, ky, indexing='xy')

K = KX**2 + KY**2
```

```
def spectral_rhs(t, uv_vec, n, beta, D1, D2, K):
    """RHS for FFT spectral version

    Accepts a real vector y containing
    real/imag parts:
      y = [Re(ut); Im(ut); Re(vt); Im(vt)] of length 4*n^2

    Returns derivatives stacked the same way
    """
    n2 = n * n

    Re_ut = uv_vec[0:n2]
    Im_ut = uv_vec[n2:2 * n2]
    Re_vt = uv_vec[2 * n2:3 * n2]
    Im_vt = uv_vec[3 * n2:4 * n2]

    ut = (Re_ut + 1j * Im_ut).reshape((n, n), order='F')
    vt = (Re_vt + 1j * Im_vt).reshape((n, n), order='F')

    # Physical fields (take real part; should remain real if symmetry holds)
    u = np.real(ifft2(ut))
    v = np.real(ifft2(vt))

    A2 = u ** 2 + v ** 2
    lam = 1 - A2
    omega = -beta * A2

    dudt_hat = fft2(lam * u - omega * v) - (K * D1) * ut
    dvdt_hat = fft2(omega * u + lam * v) - (K * D2) * vt

    dudt_flat = dudt_hat.reshape(n * n, order='F')
    dvdt_flat = dvdt_hat.reshape(n * n, order='F')

    # stack into real-imag parts
    dRe_ut = np.real(dudt_flat)
    dIm_ut = np.imag(dudt_flat)
    dRe_vt = np.real(dvdt_flat)
    dIm_vt = np.imag(dvdt_flat)

    return np.concatenate([dRe_ut, dIm_ut, dRe_vt, dIm_vt])
```

```
spectral_sol = solve_ivp(
    spectral_rhs,
    [tspan[0], tspan[-1]],
```

```
    uv,
    args=[n, beta, D1, D2, K],
    t_eval=tspan
)
```

For our second solution, we will implement the initial conditions for the Chebyshev polynomial method. We again assume that $x, y \in [-10, 10]$, $\beta = 1$, $D_1 = D_2 = 0.1$, $m = 1$, and time spanning 0 to 4 with time steps of 0.5. However, we will now let $n = N = 30$.

```
import numpy as np
from scipy.integrate import solve_ivp
from math import pi
import matplotlib.pyplot as plt
import imageio

N = 30
L = 10
m = 1      #Number of spirals
beta = 1
D_1 = D_2 = 0.1
tspan = np.arange(0, 4 + 1e-6, 0.5)
```

We now want to go about implementing the right-hand side function and Chebyshev function that we can use to solve the equations. Like the spectral solution, we will be stacking our `u` and `v` vectors, so the dimension of the new vector will be 1922×1.

```
def cheb(N):
    if N == 0:
        return np.array([[0.]]), np.array([1.])
    x = np.cos(pi * np.arange(0, N + 1) / N).reshape(-1, 1)# column
    c = np.hstack(([2.],
                   np.ones(N - 1), [2.])) \
                   * ((-1) ** np.arange(0, N + 1))
    X = np.tile(x, (1, N + 1))
    dX = X - X.T
    C = c.reshape(-1, 1) @ (1.0 / c).reshape(1, -1)
    D = C / (dX + np.eye(N + 1))
    D = D - np.diag(np.sum(D, axis=1))
    return D, x.flatten()

def chebyshev_rhs(t, uv, beta, D_1, D_2, Lap):
    """
    Implementation of the right hand side of the
    reaction-diffusion equation
```

```
    uv is the horizontally stacked U and V flattened matrices
    Lap is the Laplacian d^2_x + d^2_y

    Returns the Ut and Vt as n**2 horizontal stacked vector
    """
    U,V = np.hsplit(uv, 2)  # Splits the uv vec into u and v vectors

    A2 = U**2 + V**2        # A^2 = U^2 + V^2

    lam = 1.0 - A2          # lambda(A) = 1 - A^2
    w = -beta * A2          # w(A) = -beta A^2

    Ut = np.array(lam * U - w * V + D_1 * (Lap @ U))# Solution to Ut
    Vt = np.array(w * U + lam * V + D_2 * (Lap @ V))# Solution to Vt

    return np.hstack((Ut, Vt))
```

Now that the Chebyshev and right-hand side functions have been implemented, we can setup the rest of the initial conditions before solving. For our initial U and V conditions, we will define them the same as we did with the spectral method.

```
D,x = cheb(N)
D = D / L           # Scale since x,y in [-L,L] -> [-1,1]
D2 = D @ D
D2[0,:] = 0         # Dirichlet boundary conditions (zero first row)
D2[-1,:] = 0        # Dirichlet boundary conditions (zero last row)

x = L*x             # Scale since x in [-1,1] -> [-L,L]
y = x
X,Y = np.meshgrid(x, y, indexing='xy')

u = np.tanh(np.sqrt(X**2 + Y**2)) \
    * np.cos(m * np.angle(X + 1j*Y) - (np.sqrt(X**2 + Y**2)))
u = u.reshape((N+1)*(N+1), order="F")

v = np.tanh(np.sqrt(X**2 + Y**2)) \
    * np.sin(m * np.angle(X + 1j*Y) - (np.sqrt(X**2 + Y**2)))
v = v.reshape((N+1)*(N+1), order="F")

uv = np.hstack((u,v)).T

I = np.eye(D.shape[0])
Lap = np.kron(I, D2) + np.kron(D2, I)
```

With all of the initial conditions and function established, we can now solve the reaction diffusion equations via Chebyshev.

```
cheb_sol = solve_ivp(
    chebyshev_rhs,
    [tspan[0], tspan[-1]],
    uv,
    args=[beta,D_1,D_2,Lap],
    t_eval=tspan
)

frames = []

fig = plt.figure()
ax = fig.add_subplot(111,projection = "3d")

for j in range(len(cheb_sol.t)):
  curU = cheb_sol.y[: (N+1)*(N+1), j].reshape((N+1, N+1), order="F")

  ax.clear()
  ax.plot_surface(X, Y, curU, rstride=1, cstride=1, cmap="viridis")
  ax.set_xlim(-L, L)
  ax.set_ylim(-L, L)
  ax.set_zlim(curU.min(), curU.max())
  ax.set_xlabel("x")
  ax.set_ylabel("y")
  ax.set_zlabel("U(x,y)")
  ax.set_title(f"U(x,y) at t = {cheb_sol.t[j]:.2f}")
  ax.view_init(elev=45, azim=-45)

  plt.tight_layout()
  plt.savefig("frame.png", dpi=80)
  frames.append(imageio.imread("frame.png"))

plt.close(fig)
imageio.mimsave("chebyshev_reaction_diffusion.gif",
                frames,
                fps = 1.85
)
```

We can observe the spiral of $m = 1$ in the plot below (one frame from the visualization that was output):

```
## <matplotlib.colorbar.Colorbar object at 0x000002707A3952B0>
```

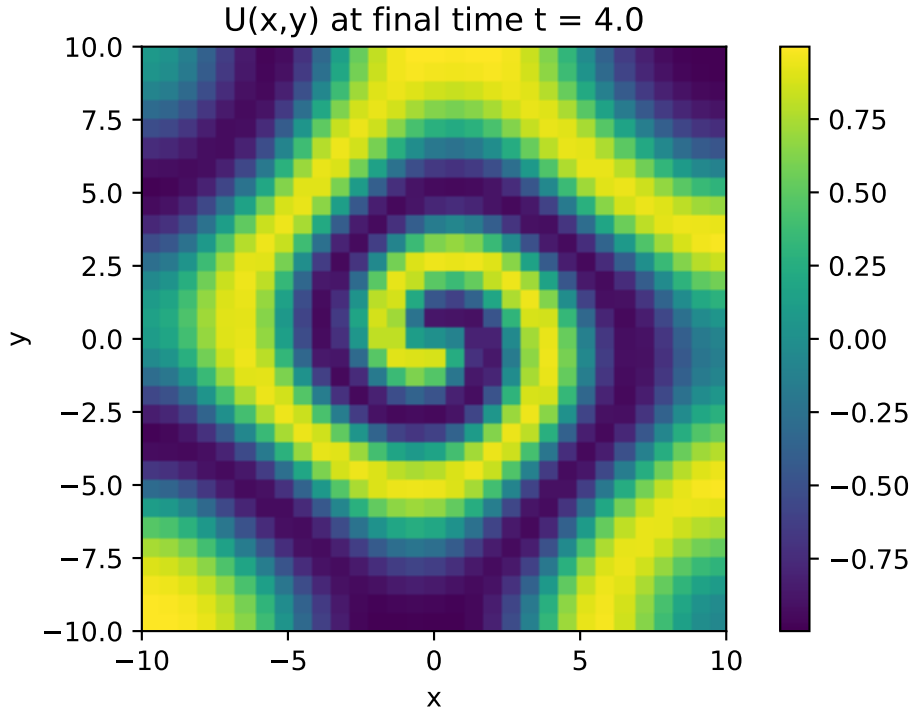

Figure 3: Solution of the Reaction-Diffusion equations with $t \in [0, 4]$ and $m = 1$. It is worth noting that we can increase m and observe the changes to the visualization.

It is important to note that we can increase m to be how ever many spirals we want in the solution. That can be a task for the reader to try and change parameters as they wish, and observe the changes in the visualization.

Conclusion

Throughout this notebook, we were able to discuss the various methods that can be employed to solve differential equations, and were able to look at three different problems that could be solved utilizing those methods. While the topics discussed just touched on the essentials needed to solve differential equations, additional resources can be found exploring the methods in more depth.

References

Bertsimas, D. & Tsitsiklis, J.N. (1997). *Introduction to Linear Optimization (6th Edition).* Athena Scientific.

D'Ambra, P. & Durastante, F. & Filippone, S. (2025). *Parallel mathcing-based AMG preconditioners for elliptic equations discretized by IgA [Preprint].* arXiv. https://arxiv.org/pdf/2511.21268

Gustafsson, B. (2018). *Scientific Computing: A Historical Perspective.* Springer. https://doi.org/10.1007/978-3-319-69847-2.

Heath, M.T. (2018). *Scientific Computing: An Introductory Survey (2nd Edition).* SIAM. https://epubs.siam.org/doi/book/10.1137/1.9781611975581.

Kincaid, D.R. & Cheney, E.W. (2009). *Numerical Analysis: Mathematics of Scientific Computing (3rd Edition).* American Mathematical Society.

Li, R. & Di, H. & Jiang, S.W. (2025). *Two-Step Generalized RBF-Generated Finite Difference Method on Manifolds [Preprint].* arXiv. https://arxiv.org/pdf/2511.18049