Student's Names: Siro Brotón Gutierrez and Yan Chen Zhou

NIA: 100496757 and 100496683

Lab 1: Solving the Heat Equation – A Demonstration of Stiffness

The Method of Lines (MOL)

Linear systems of ODEs naturally arise when solving PDEs. For example, consider solving the following initial boundary value problem:

$$u_t = u_{xx}, \quad x \in (0,1), \quad u(0) = u(1) = 0, \quad u(x,0) = f(x).$$

To solve this:

1. We first discretize the spatial variable x on a uniform grid:

   $$x_i = ih, \quad i = 0, \ldots, N, \quad \text{with } h = 1/N.$$

   Define $U_i(t) \approx u(x_i, t)$.

2. Approximate the second derivative using the second-order finite difference formula:

   $$\left.\frac{\partial^2 u}{\partial x^2}\right|_{(x_j,t)} \approx \frac{u(x_j - h, t) - 2u(x_j, t) + u(x_j + h, t)}{h^2}.$$

   Substituting into the Heat equation, we obtain the ODE system:

   $$U_i'(t) = \frac{1}{h^2}(U_{i-1}(t) - 2U_i(t) + U_{i+1}(t)), \quad i = 1, \ldots, m.$$

   Or, in matrix-vector form:

   $$\mathbf{U}'(t) = A\mathbf{U},$$

   where $\mathbf{U}(t) = [U_1(t), \ldots, U_m(t)]^T$ and A is the $m \times m$ matrix:

   $$A = \frac{1}{h^2}\begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{bmatrix}.$$

   This ODE system is called the **semidiscrete method**.

3. We then solve this system using a numerical ODE solver, a technique called the **method of lines (MOL)**.

1. Numerical Implementation

Write a script to solve this PDE problem with the following requirements:

- Include an `input` section where all parameters of the PDE problem are defined.
- Implement a function that returns matrix A given h and m. In Python, this can be done using NumPy:

```
import numpy as np

def Amatrix(m, h):
    A = (-2 / h**2) * np.diag(np.ones(m)) \
        + (1 / h**2) * np.diag(np.ones(m - 1), 1) \
        + (1 / h**2) * np.diag(np.ones(m - 1), -1)
    return A
```

- Define the right-hand side function of the semidiscrete method.
- Modify your ODE solvers (FE, RK4, BE and Crank-Nicolson) to allow to solve the linear system of ODEs

$$\mathbf{U}'(t) = A\mathbf{U}.$$

```
# Imports
import numpy as np
import scipy.optimize as opti
import matplotlib.pyplot as plt
import math
```

```
# Parameters
a, b = 0, 1
m = 41
h = 1 / (m + 1)
x = np.linspace(a, b, m+2)
dt = 0.5 * h**2  # CFL condition
tspan = np.arange(0, 0.5 + dt, dt)

# Function definitions
def u_exact(x, t):
    return np.sin(np.pi * x) * np.exp(-np.pi**2 * t)

def Amatrix(m, h):
    A = (-2 / h**2) * np.eye(m) + (1 / h**2) * (np.eye(m, k=1) + np.eye(m, k=-1))
    return A

# Initialize ODE
A = Amatrix(m, h)
f = lambda u: A @ u
```

Systems of differential equations

For systems of differential equations, $u' = Au$ the general solution is of the form $u(t) = e^{At}u(0)$. The behavior of this solution depends largely on the eigenvalues of A. A necessary condition for absolute stability is that $z = \Delta t$ be in the stability region for each eigenvalue λ of A. To see this, suppose that $A \in \mathbb{R}^{m\times m}$ can be diagonalized as $A = R\Lambda R^{-1}$. Then,

$$u' = Au = R\Lambda R^{-1}u.$$

Let $v = R^{-1}u$ so then we obtain m decoupled equations:

$$v' = \Lambda v.$$

Let $\lambda_k, k = 1, \ldots, m$, be the eigenvalues of A. If we use one method to solve the system of equations, then we must choose the time step Δt so that $\Delta t\lambda_k$ lies in the stability region for all $k = 1, \ldots, m$.

2. Eigenvalues of Matrix A

By simply substitution of the eigenfunction $u_j^{(k)} = \sin(k\pi jh)$ into the discretized operator $(u_{j-1}^{(k)} - 2u_j^{(k)} + u_{j+1}^{(k)})/h^2 = \lambda_k u_j^{(k)}$, it is possible to show that

$$\lambda_k = -\frac{2}{h^2}[1 - \cos(k\pi h)], \quad k = 1, \ldots, m.$$

are the eigenvalues of A. Verify this numerically.

Hint: you can do that in Python using:

```
eigenvalues = np.linalg.eigvals(A)  # Numerical eigenvalues
dk = -2 / h**2 * (1 - np.cos(np.arange(1, m + 1) * np.pi * h))
```

```
A = Amatrix(m, h)
eigenvalues = np.linalg.eigvals(A)                              # Numerical eigenvalues
dk = -2 / h**2 * (1 - np.cos(np.arange(1, m + 1) * np.pi * h))  # Analytical eigenvalues

eigenvalues.sort()
dk = np.flip(dk)

print(eigenvalues)
print(dk)

# print(dt)
# print(np.max(np.abs(eigenvalues)) * dt)
```

```
[-7046.13499646 -7016.59515492 -6967.54567418 -6899.26085881
 -6812.12258522 -6706.61816596 -6583.33762455 -6442.97039579
 -6286.30147015 -6114.20700382 -5927.64941886 -5727.67202096
 -5515.39316485 -5292.         -5058.74183161 -4816.92313396
 -4567.89625532 -4313.053855   -4053.82111507 -3791.64777017
 -3528.         -3264.35222983 -3002.17888493 -2742.946145
 -2488.10374468 -2239.07686604 -1997.25816839 -1764.
 -1540.60683515 -1328.32797904 -1128.35058114  -941.79299618
  -769.69852985  -613.02960421  -472.66237545  -349.38183404
  -243.87741478  -156.73914119   -88.45432582   -39.40484508
    -9.86500354]
[-7046.13499646 -7016.59515492 -6967.54567418 -6899.26085881
 -6812.12258522 -6706.61816596 -6583.33762455 -6442.97039579
 -6286.30147015 -6114.20700382 -5927.64941886 -5727.67202096
 -5515.39316485 -5292.         -5058.74183161 -4816.92313396
 -4567.89625532 -4313.053855   -4053.82111507 -3791.64777017
```

```
-3528.         -3264.35222983 -3002.17888493 -2742.946145
-2488.10374468 -2239.07686604 -1997.25816839 -1764.
-1540.60683515 -1328.32797904 -1128.35058114  -941.79299618
 -769.69852985  -613.02960421  -472.66237545  -349.38183404
 -243.87741478  -156.73914119   -88.45432582   -39.40484508
   -9.86500354]
```

3. Forward Euler Method

Consider the forward Euler method applied to the heat equation:

$$U_j^{n+1} = U_j^n + \frac{\Delta t}{h^2}\left(U_{j-1}^n - 2U_j^n + U_{j+1}^n\right).$$

The local truncation error for this method can be shown to be

$$\tau(x_j, t_n) = \frac{u(x_j, t_{n+1}) - u(x_j, t_n)}{\Delta t} - \frac{1}{h^2}[u(x_{j-1}, t_n) - 2u(x_j, t_n) + u(x_{j+1}, t_n)] = \mathcal{O}(\Delta t + h^2).$$

(a) Stability Analysis

Find the condition on $\Delta t \lambda_k$ for absolute stability. This yields the so-called *CFL condition*.

(b) Numerical Experiment

Suppose that the exact solution is $u(x,t) = \sin(\pi x)e^{-\pi^2 t}$, and we want to integrate the equation up to $T = 0.5$. Run your code for different spatial resolutions $m = 10, 20, 40, 80, 160, \ldots$. For each m, make sure that Δt meets the condition for absolute stability in (b). Compute the global error at $t = T$ and determine the order of convergence (similar to what we did in class with an ODE). Measure execution time.

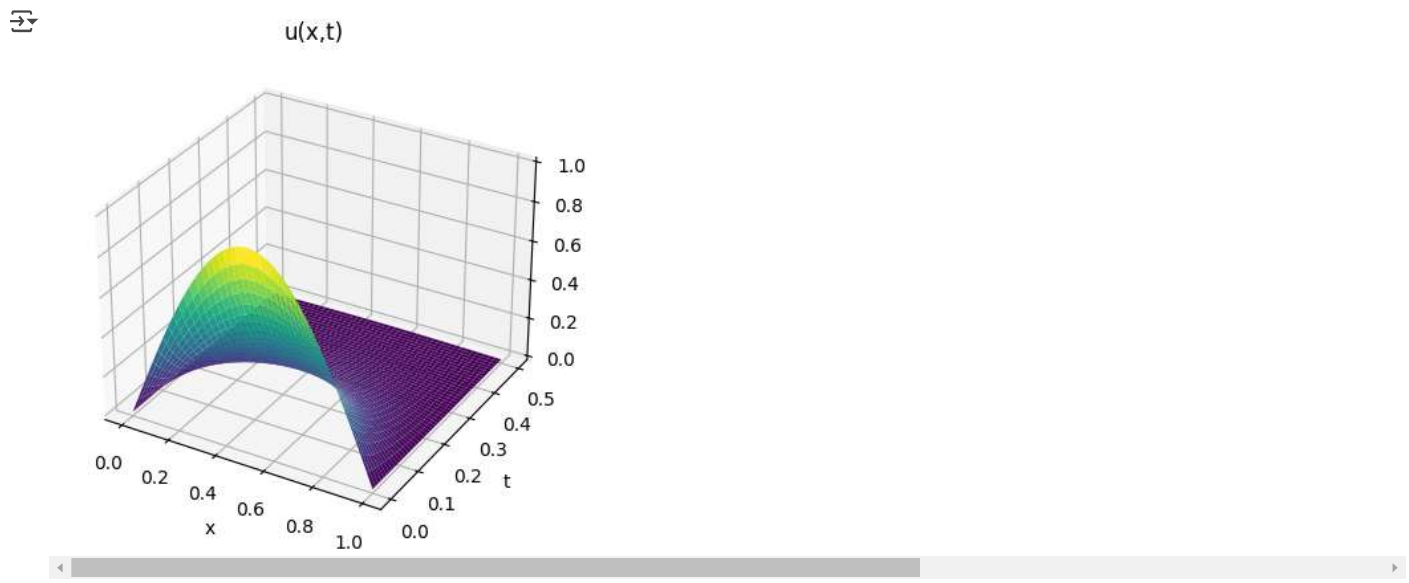
(c) Solution Plot

Plot the evolution of the solution in a (x,t)-coordinate system. What do you observe if the stability condition in (b) (*CFL condition*) is not met?

```
import numpy as np
import matplotlib.pyplot as plt

# Parameters
a, b, m = 0, 1, 41
h = 1 / (m + 1)
x = np.linspace(a, b, m+2)
dt = 0.5 * h**2  # CFL condition
tspan = np.arange(0, 0.5 + dt, dt)

# Function definitions
def u_exact(x, t):
    return np.sin(np.pi * x) * np.exp(-np.pi**2 * t)

def Amatrix(m, h):
    A = (-2 / h**2) * np.eye(m) + (1 / h**2) * (np.eye(m, k=1) + np.eye(m, k=-1))
    return A

def forward_euler(f, tspan, u0, dt):
    u = [u0]
    for _ in tspan[:-1]:
        u.append(u[-1] + dt * f(u[-1]))
    return np.array(u)

# Initialize and solve ODE
A = Amatrix(m, h)
f = lambda u: A @ u
u0 = u_exact(x[1:-1], 0)

u = forward_euler(f, tspan, u0, dt)
u = np.vstack([np.zeros(len(tspan)), u.T, np.zeros(len(tspan))])  # Apply BCs

# Plot results
X, T = np.meshgrid(x, tspan)
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, T, u.T, cmap='viridis')

ax.set_title('u(x,t)')
```

```
ax.set_xlabel('x')
ax.set_ylabel('t')
plt.show()
```

```
def forward_euler(f, tspan, u0, dt):
    u = [u0]
    for _ in tspan[:-1]:
        u.append(u[-1] + dt * f(u[-1]))
    return np.array(u)

# Initialize and solve ODE
A = Amatrix(m, h)
f = lambda u: A @ u
u0 = u_exact(x[1:-1], 0)

u = forward_euler(f, tspan, u0, dt)
u = np.vstack([np.zeros(len(tspan)), u.T, np.zeros(len(tspan))])  # Apply BCs

# Plot results
X, T = np.meshgrid(x, tspan)
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, T, u.T, cmap='viridis')

ax.set_title('u(x,t)')
ax.set_xlabel('x')
ax.set_ylabel('t')
plt.show()
```

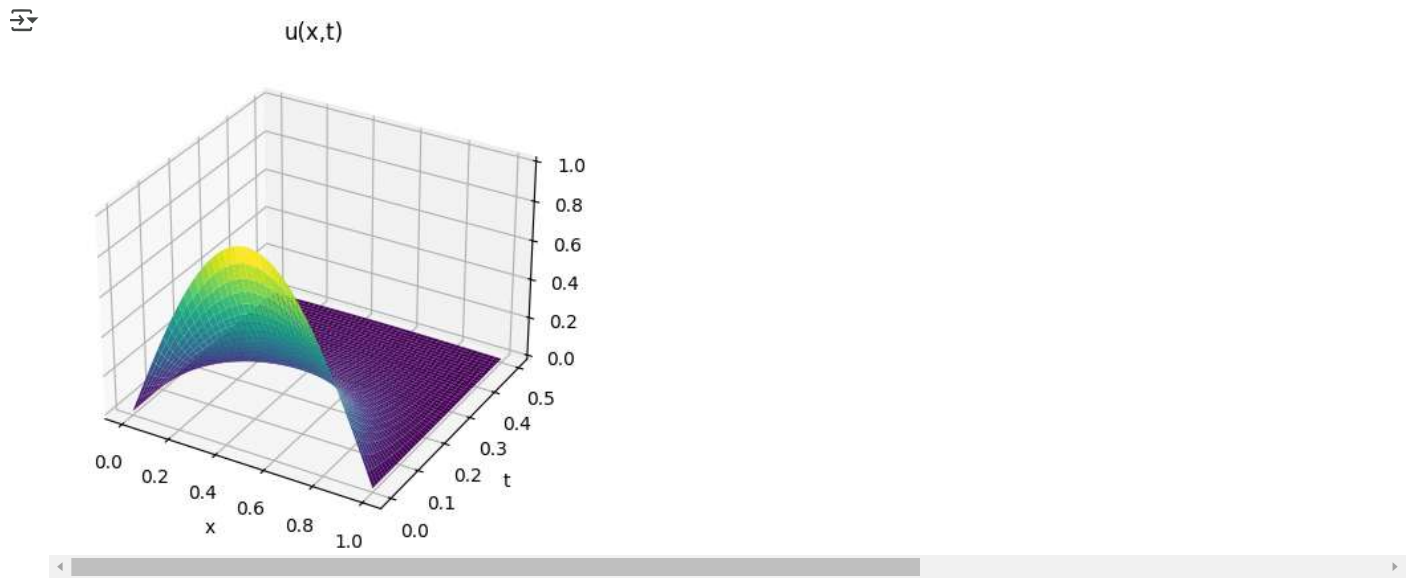


4. RK4 Method

Repeat part (3a) using the Runge-Kutta 4th order (RK4) method.

```
import numpy as np
import matplotlib.pyplot as plt


def rk4(f, tspan, u0, dt):
    u = [u0]
    for _ in tspan[:-1]:
        y1 = u[-1]
        y2 = u[-1] + (dt/2)*f(y1)
        y3 = u[-1] + (dt/2)*f(y2)
        y4 = u[-1] + dt*f(y3)
        f1, f2, f3, f4 = f(y1), f(y2), f(y3), f(y4)
        ui = u[-1] + (dt/6)*(f1+2*f2+2*f3+f4)
        u.append(ui)

    return np.array(u)

# Initialize and solve ODE
A = Amatrix(m, h)
f = lambda u: A @ u
u0 = u_exact(x[1:-1], 0)

u = rk4(f, tspan, u0, dt)
```

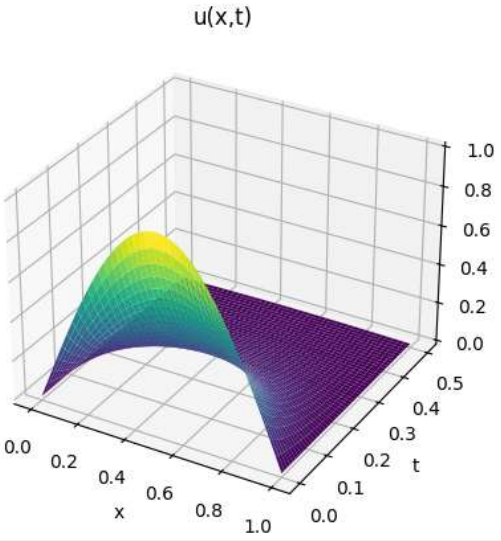
```
u = np.vstack([np.zeros(len(tspan)), u.T, np.zeros(len(tspan))])  # Apply BCs

# Plot results
X, T = np.meshgrid(x, tspan)
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, T, u.T, cmap='viridis')

ax.set_title('u(x,t)')
ax.set_xlabel('x')
ax.set_ylabel('t')
plt.show()
```



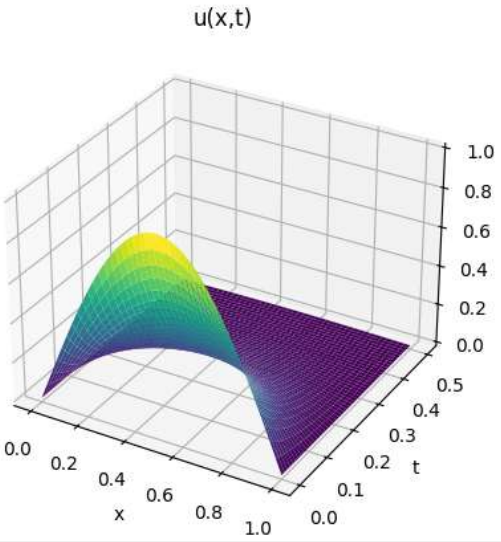
5. Implicit Method

Repeat part (3a) using the BE and Trapezoidal methods for this problem.

```
def backward_euler(f, tspan, u0, dt):
    u = [u0]
    for _ in tspan[:-1]:
        be = lambda x: x-u[-1]-dt*f(x)
        u.append(opti.fsolve(be, u[-1]))
    return np.array(u)

# Initialize and solve ODE
A = Amatrix(m, h)
f = lambda u: A @ u
u0 = u_exact(x[1:-1], 0)

u = backward_euler(f, tspan, u0, dt)
u = np.vstack([np.zeros(len(tspan)), u.T, np.zeros(len(tspan))])  # Apply BCs

# Plot results
X, T = np.meshgrid(x, tspan)
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, T, u.T, cmap='viridis')

ax.set_title('u(x,t)')
ax.set_xlabel('x')
ax.set_ylabel('t')
plt.show()
```




```
def trapezoidal(f, tspan, u0, dt):
    u = [u0]
    for _ in tspan[:-1]:
        trpz = lambda x: x-u[-1]-0.5*dt*(f(u[-1])+f(x))
        u.append(opti.fsolve(trpz, u[-1]))
    return np.array(u)


u = trapezoidal(f, tspan, u0, dt)
u = np.vstack([np.zeros(len(tspan)), u.T, np.zeros(len(tspan))])  # Apply BCs

# Plot results
X, T = np.meshgrid(x, tspan)
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, T, u.T, cmap='viridis')

ax.set_title('u(x,t)')
ax.set_xlabel('x')
ax.set_ylabel('t')
plt.show()
```



6. Efficiency Comparison

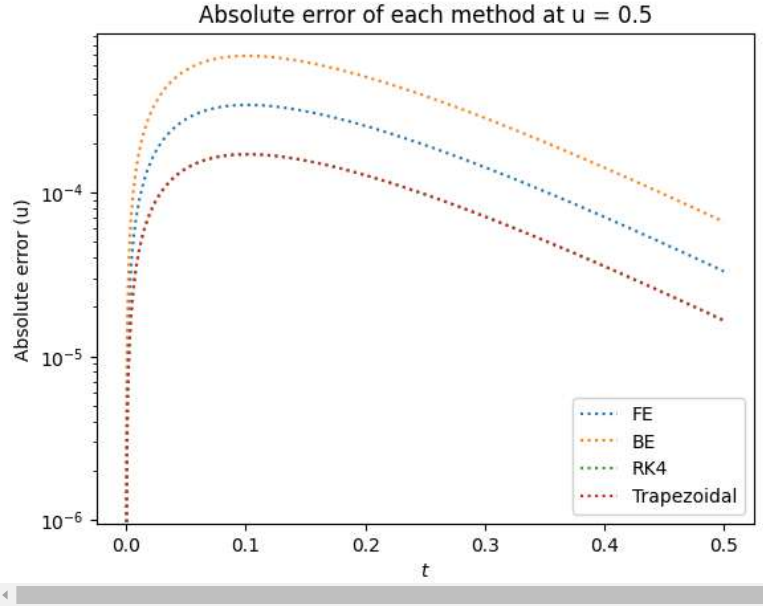
Compare the efficiency of all methods and discuss the best choice for solving this PDE.

```
# Parameters
a, b, m = 0, 1, 41
h = 1 / (m + 1)
dt = 0.5 * h**2  # CFL condition

# Initialize and solve ODE
```

```
A = Amatrix(m, h)
f = lambda u: A @ u
x = np.linspace(a, b, m+2)
u0 = u_exact(x[1:-1], 0)
tspan = np.arange(0, 0.5 + dt, dt)

sol = []
for t in tspan:
    sol.append(u_exact(x[1:-1], t))
sol = np.array(sol)
sol = np.vstack([np.zeros(len(tspan)), sol.T, np.zeros(len(tspan))])

u = forward_euler(f, tspan, u0, dt)
u_fe = np.vstack([np.zeros(len(tspan)), u.T, np.zeros(len(tspan))])  # Apply BCs
u = backward_euler(f, tspan, u0, dt)
u_be = np.vstack([np.zeros(len(tspan)), u.T, np.zeros(len(tspan))])  # Apply BCs
u = rk4(f, tspan, u0, dt)
u_rk4 = np.vstack([np.zeros(len(tspan)), u.T, np.zeros(len(tspan))])  # Apply BCs
u = trapezoidal(f, tspan, u0, dt)
u_trpz = np.vstack([np.zeros(len(tspan)), u.T, np.zeros(len(tspan))])  # Apply BCs
```

To better visualize and compare the errors, the plot below takes the point x which is closest to 0.5. We can see that the Implicit Trapezoidal Method and RK4 perform the best.

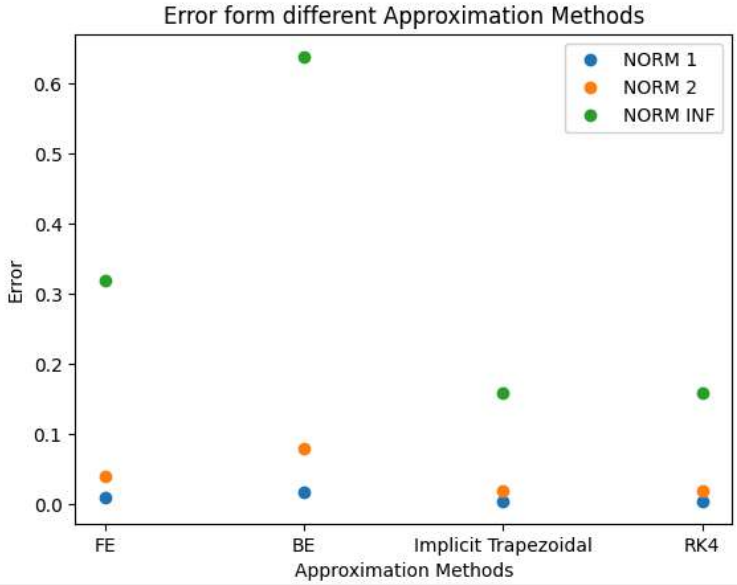
```
err_fe = abs(u_fe - sol)
err_be = abs(u_be - sol)
err_rk4 = abs(u_rk4 - sol)
err_trpz = abs(u_trpz - sol)

plt.semilogy(tspan, err_fe[math.ceil(m/2), :], ':', label='FE')
plt.semilogy(tspan, err_be[math.ceil(m/2), :], ':', label='BE')
plt.semilogy(tspan, err_rk4[math.ceil(m/2), :], ':', label='RK4')
plt.semilogy(tspan, err_trpz[math.ceil(m/2), :], ':', label='Trapezoidal')

plt.legend()
plt.title("Absolute error of each method at u = 0.5")
plt.xlabel("$t$")
plt.ylabel("Absolute error (u)")
plt.show()
```



If we evaluate and compare the performance of each approximation method using norms, we find that Implicit Trapezoidal performs the best.

```
labels = [
    "NORM 1",
    "NORM 2",
    "NORM INF"
]
for i, ord in enumerate([1, 2, np.inf]):
    err_fe = np.linalg.norm(u_fe - sol, ord=ord)
    err_be = np.linalg.norm(u_be - sol, ord=ord)
    err_rk4 = np.linalg.norm(u_rk4 - sol, ord=ord)
    err_trpz = np.linalg.norm(u_trpz - sol, ord=ord)
    errs = [err_fe, err_be, err_rk4, err_trpz]
    plt.plot(['FE', 'BE', 'Implicit Trapezoidal', 'RK4'], errs, 'o', label=labels[i])
```

```
plt.title("Error form different Approximation Methods")
plt.legend()
plt.xlabel("Approximation Methods")
plt.ylabel("Error")
plt.show()
```



Forward Euler is obviously inside the stabliity region since it is inside $-2 < k\max(\lambda) < 0$ for the chosen k.

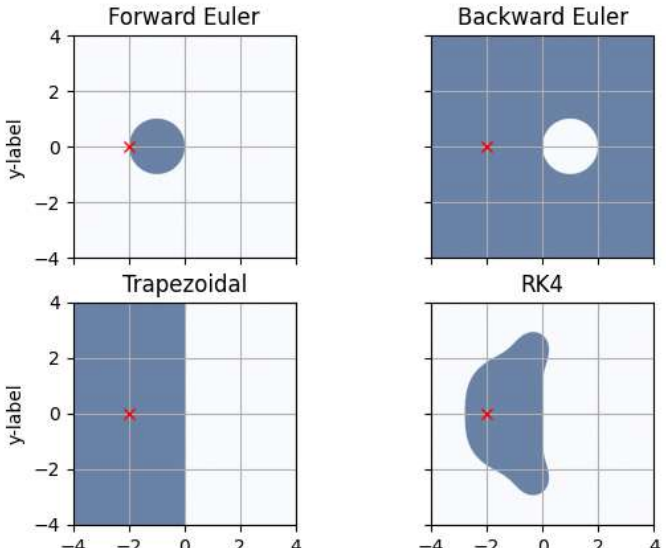
For the implicit methods we need $d(1, k\max(\lambda)) > 1$ in Backward Euler and $k\max(\lambda) < 0$ in Trapezoid method, which can be satisfied with any positive k since $\max(\lambda) < 0$.

The stability region for rk4 (in the real line it's [-2.785,0] approximately) contains that of Forward euler, giving us that because k satisfies stability conditions of the previous methods, rk4 will also be stable

To better picture this, we can see it plotted below:

```
x = np.arange(-4, 4, 0.01)
y = np.arange(-4, 4, 0.01)
x, y = np.meshgrid(x, y)
z = x + 1j * y


R1 = 1 + z
R1 = 1 - z
R4 = 1 + z + 0.5 * z**2 + (1/6) * z**3 + (1/24) * z**4

fig, axs = plt.subplots(2, 2)
axs[0, 0].imshow(np.abs(z+1) < 1, extent=(-4, 4, -4, 4), origin='lower', cmap='Blues', alpha=0.6)
axs[0, 0].set_title('Forward Euler')

axs[0, 1].imshow(np.abs(z-1) > 1, extent=(-4, 4, -4, 4), origin='lower', cmap='Blues', alpha=0.6)
axs[0, 1].set_title('Backward Euler')

axs[1, 0].imshow(x < 0, extent=(-4, 4, -4, 4), origin='lower', cmap='Blues', alpha=0.6)
axs[1, 0].set_title('Trapezoidal')

R4 = 1 + z + 0.5 * z**2 + (1/6) * z**3 + (1/24) * z**4
axs[1, 1].imshow(np.abs(R4) < 1, extent=(-4, 4, -4, 4), origin='lower', cmap='Blues', alpha=0.6)
axs[1, 1].set_title('RK4')

kmaxlambda = -np.max(np.abs(eigenvalues)) * dt
K = 'k'
for ax in axs.flat:
    ax.grid(True)
    ax.set(xlabel='x-label', ylabel='y-label')
    ax.plot(kmaxlambda, 0, 'rx')


# Hide x labels and tick labels for top plots and y ticks for right plots.
for ax in axs.flat:
    ax.label_outer()
```

[4]


Forward Euler
Backward Euler
Trapezoidal
RK4
y-label
y-label
4
2
0
−2
−4