Problem 1. (Polynomial interpolation, 10 pts)

a) What is the definition of the (polynomial) interpolation problem? **(2 pts)**

b) Write down the error estimate for the interpolation error (don't prove it!) and discuss briefly why it might be advantageous to use Chebyshev interpolation nodes instead of equidistant interpolation nodes. **(3 pts)**

Next, consider the data points

x_i	-2	0	1
$f(x_i)$	2	2	4

c) Determine the 3 Cardinal (Lagrange basis) functions $\{\ell_i(x)\}_{i=0}^2$ associated with the nodes $x_0 = -2$, $x_1 = 0$ and $x_2 = 1$. **(3 pts)**

d) Use the Cardial functions to determine the interpolation polynomial $p_2(x)$ associated with the given data points $(x_i, f(x_i))$ above. The final interpolation polynomial should be written in the form $\sum_{i=0}^2 a_i x^i$. **(2 pts)**

Solution.

a) Given $n+1$ points $(x_i, y_i)_{i=0}^n$, find a polynomial $p(x)$ of lowest possible degree satisfying the **interpolation condition**

$$p(x_i) = y_i, \qquad i = 0, \dots, n.$$

1 pt for interpolation condition, 1 pt for lowest degree

b) Assume that $\{x_i\}_{i=0}^n \subset [a, b]$, then for $f \in C^{n+1}(a, b)$ the error estimate for any $x \in [a, b]$ is given by

$$f(x) - p_n(x) = \frac{f^{n+1}(\xi)}{(n+1)!}\omega_{n+1}(x)$$

where $\omega_{n+1}(x) = (x - x_0) \cdot (x - x_1) \cdots (x - x_n)$ and $\xi \in (a, b)$. Chebyshev nodes can be advantageous as they usually drastically reduce the interpolation error. This is caused by the fact that for Chebyshev nodes, the resulting polynomial $\omega_{n+1}(x)$ appearing in the error estimate has a min-max property. More precisely, it is the polynomial among all polynomials of order $n+1$ with leading coefficient 1 which minimizes its maximum norm.
1 pt for error estimate, 1 pt for mentioning error reduction when using Chebyshev nodes, 1 pt for min-max property

c) In general the Cardinal functions/Lagrange polynomials for $n+1$ nodes are given

$$Ł_i(x) = \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}, \qquad i = 0, \dots, n$$

. Lagrange polynomial and resulting interpolation polynomial for the given 3 data points are thus

$$L_0 = \frac{x\,(x-1)}{6}$$
$$L_1 = -\frac{(x-1)\,(x+2)}{2}$$
$$L_2 = \frac{x\,(x+2)}{3}$$

1 pt for each Lagrange polynomial

d) The resulting interpolation polynomial can then be computed as

$$\begin{aligned} p_2(x) &= 2 \cdot L_9(x) + 2 \cdot L_1(x) + 4 \cdot L_2(x) \\ &= \frac{x\,(x-1)}{3} - (x-1)\,(x+2) + \frac{4x\,(x+2)}{3} = \frac{2x^2}{3} + \frac{4x}{3} + 2 \end{aligned}$$

1 pt for linear combination of Lagrange polynomials, 1 pt for correct final form

Problem 2. (Quadrature, 10 pts)

a) Explain what the degree of exactness for a general quadrature rule is. **(1 pts)**

b) Given the quadrature points $x_0 = -2$, $x_1 = 0$ and $x_2 = 1$. Determine the corresponding weights $\omega_0, \omega_1, \omega_2$ such that the quadrature rule defined by $\{(x_i, \omega_i)\}_{i=0}^2$ has *at least* degree of exactness 2 on the interval $[-2, 1]$.

Hint: You might want to solve Problem 1c) first to save you some time. **(4 pts)**

c) Let $BQ[f](a,b)$ be the so-called Boole's quadrature rule which computes an approximation of the integral $I[f](a,b) := \int_a^b f(x)\ \mathrm{d}x$ and satisfies for any $f \in C^6[a,b]$ an error estimate of the form

$$|BQ[f](a,b) - I[f](a,b)| = \frac{8}{945}\left(\frac{b-a}{4}\right)^7 f^{(6)}(\xi)$$

with some $\xi \in (a,b)$. Which degree of exactness has Boole's quadrature rule and why? **(2 pts)**

d) Assume that you now use Boole's quadrature rule to create a composite Boole's quadrature rule $CBQ[f]$(a,b) on the interval $[a,b]$ using m equally spaced subintervals $I_i = [x_{i-1}, x_i]$ where $x_i = a + hi$, $i = 0, \ldots, m$ and $h = (b-a)/m$. Prove then the following estimate for the quadrature error holds:

$$|CBQ[f](a,b) - I[f](a,b)| \leqslant \frac{8}{945 \cdot 4^7}(b-a)h^6 \max_{\xi\in[a,b]} |f^{(6)}(\xi)|$$

(3 pts)

Solution.

a) A numerical quadrature has degree of exactness d if $Q[p](a,b) = I[p](a,b)$ for all $p \in \mathbb{P}_d$ and there is at least one $p \in \mathbb{P}_{d+1}$ such that $Q[p](a,b) \neq I[p](a,b)$. 1 pt for definition

b) We need to compute the Lagrange polynomials L_0, L_1 and L_2 associated with the quadrature points x_0, x_1, x_2. Then ω_i are determined by

$$\omega_i = \int_{-2}^{1} L_i(x)dx$$

Note that we had the same points in the Problem 1, so we do not need to recompute L_i,

we only need to integrate the computed Lagrange polynomials

$$\omega_0 = \int_{-2}^{1} \frac{x\,(x-1)}{6} dx = 3/4$$
$$\omega_1 = -\int_{-2}^{1} \frac{(x-1)\,(x+2)}{2} dx = 9/4$$
$$\omega_2 = \int_{-2}^{1} \frac{x\,(x+2)}{3} dx = 0$$

1 pt for general weight formula, 1 pt for each weight

c) Boole's quadrature rule has degree of exactness equal to 5, because the right-hand side of its error estimate involves the 6th derivative of f which vanishes for all polynomials of degree less or equal to 5, showing that in that case the quadrature error is 0. 1 pt for correct degree of exactness, 1 pt for reasoning

d) Use the additivity of the usual integral w $I[f](a,b) = \sum_{i=1}^{m} I[f](x_{i-1}, x_i)$, the definition of the composite quadrature rule and the error estimate for the Boole quadrature above on a single interval to deduce that

$$|CBQ[f](a,b) - I[f](a,b)| = |\sum_{i=1}^{m} BQ[f](x_{i-1},x_i) - I[f](x_{i-1},x_i)| \quad (1)$$
$$\leqslant \sum_{i=1}^{m} |\frac{8}{945}\left(\frac{x_i - x_{i-1}}{4}\right)^7 f^{(6)}(\xi)| \quad (2)$$
$$\leqslant \frac{8}{945 \cdot 4^7} \sum_{i=1}^{m} h^7 M_i \leqslant \frac{8M}{945 \cdot 4^7} m h^7 = \frac{8M}{945 \cdot 4^7}(b-a)h^6 \quad (3)$$

where we used the notation $\xi_i \in (x_{i-1}, x_i)$, $M_i = \max_{\xi \in [x_{i-1}, x_i]} |f(\xi)|$, $M = \max_{i=1,\ldots,m} M_i$. 1 pt for additivity of integral, 1 pt for using BQ on each subinterval, 1 pt for correct final estimate

Problem 3. (Numerical methods for ODEs, 15 points)

For an ordinary differential equation of the form

$$\frac{dy}{dt} = f(t, y), \quad y(0) = y_0$$

the Crank-Nicolson method to approximate the solution defined by the following scheme

$$y_{n+1} = y_n + \frac{\tau}{2}\left(f(t_n, y_n) + f(t_{n+1}, y_{n+1})\right)$$

a) Explain the connection between the Crank-Nicolson method and the trapezoidal rule. **(1 pts)**

b) Rewrite the Crank-Nicolson method into the form of a Runge-Kutta method and derive the Butcher tableau. **(4 pts)**

c) Determine the stability function $r(z)$ of the Crank-Nicolson method. **(3 pts)**

d) Determine the stability region of the Crank-Nicolson method. **(4 pts)**

e) Use the order conditions for Runge-Kutta methods as summarized in the table below,

p	conditions
1	$\sum_{i=1}^{s} b_i = 1$
2	$\sum_{i=1}^{s} b_i c_i = 1/2$
3	$\sum_{i=1}^{s} b_i c_i^2 = 1/3$ $\sum_{i,j=1}^{s} b_i a_{ij} c_j = 1/6$
4	$\sum_{i=1}^{s} b_i c_i^3 = 1/4$ $\sum_{i,j=1}^{s} b_i c_i a_{ij} c_j = 1/8$ $\sum_{i,j=1}^{s} b_i a_{ij} c_j^2 = 1/12$ $\sum_{i,j,k=1}^{s} b_i a_{ij} a_{jk} c_k = 1/24$

to determine the consistency order of the Crank-Nicolson method. **(3 pts)**

Solution.

a) The Crank-Nicolson method can be derived by rewriting the ODE into an integral equation and applying the trapezoidal rule to the integral:

$$\begin{aligned} y(t+\tau) - y(t) &= \int_t^{t+\tau} y'(s)\, \mathrm{d}s \\ &= \int_t^{t+\tau} f(s, y(s))\, \mathrm{d}s \approx \frac{\tau}{2}(f(t, y(t)) + f(t+\tau, y(t+\tau))). \end{aligned}$$

1 pt for integral equation and application of trapezoidal rule

b) To rewrite the Crank-Nicolson method as a general Runge-Kutta method, define,

$$k_1 := f(t_k, y_k) = f(t_k + \underbrace{0}_{c_1} \cdot \tau_k, y_k + \tau_k \underbrace{0}_{a_{11}} \cdot k_1 + \tau_k \underbrace{0}_{a_{21}} \cdot k_2) \Rightarrow c_1 = a_{11} = a_{21} = 0$$

and

$$k_2 := f(t_k + \tau_k, y_k + \tau_k \tfrac{1}{2} k_1 + \tau_k \tfrac{1}{2} k_2)$$

so that next step can be computed by

$$y_{k+1} := y_k + \tau_k(\frac{1}{2}k_1 + \frac{1}{2}k_2) = y_k + \tau_k \underbrace{\frac{1}{2}}_{b_1} k_1 + \tau_k \underbrace{\frac{1}{2}}_{b_2} k_2$$

Thus, the Butcher table is

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & \frac{1}{2} & \frac{1}{2} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

1 pt for k_1, 1 pt for k_2, 1 pt for rewriting y_{n+1} 1 pt for Butcher tableau

c) To compute the stability function for the Crank-Nicolson method, simply apply the scheme to the test equation $\frac{dy}{dt} = \lambda y$. Then

$$\begin{aligned} y_{n+1} &= y_n + \frac{\tau}{2}\lambda y_n + \frac{\tau}{2}\lambda y_{n+1} \\ \Leftrightarrow y_{n+1} &= \frac{1 + \frac{\tau\lambda}{2}}{1 - \frac{\tau\lambda}{2}} y_n \end{aligned}$$

Setting $z = \tau\lambda$, we see that the stability function is given by

$$r(z) = \frac{1 + \frac{z}{2}}{1 - \frac{z}{2}}$$

d) To determine the stability region $S = \{z \in \mathbb{C} \mid |r(z)| \leqslant 1\}$, we first determine its boundary

$$\partial S = \{z \in \mathbb{C} \mid |r(z)| = 1\}$$

Now $|r(z)| = 1 \Leftrightarrow |r(z)|^2 = 1$. Writing $r(z) = a + bi$

$$\begin{aligned} & |r(z)|^2 = 1 \Leftrightarrow |1 - z/2|^2 = |1 + z/2|^2 \\ \Leftrightarrow \quad & (1 - a/2)^2 + (b/2)^2 = (1 + a/2)^2 + (b/2)^2 \\ \Leftrightarrow \quad & (1 - a/2)^2 = (1 + a/2)^2 \\ \Leftrightarrow \quad & 1 - 2a + a^2 = 1 + 2a + a^2 \Leftrightarrow 4a = 0 \end{aligned}$$

So the boundary is given by the y-axis, $\partial S = \{z \in \mathbb{C} | \mathrm{Re}(z) = 0\}$. Moreover, we for *real* $z < 0$, we see immediately that that $|r(z)| < 1$, so it is the left half-plane which is contained in the stability region. 1 pt for stating the right test ODE, 1 pt for applying it to C-N, 1 pt for final stability function

e) Checking the consistency order using the order conditions we see that

$p = 1$	$b_1 + b_2 = \frac{1}{2} + \frac{1}{2} = 1$	OK
$p = 2$	$b_1 c_1 + b_2 c_2 = \frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 1 = \frac{1}{2}$	OK
$p = 3$	$b_1 c_1^2 + b_2 c_2^2 = \frac{1}{2} \cdot 0^2 + \frac{1}{2} \cdot 1^2 = \frac{1}{2} \neq \frac{1}{3}$	Not satisfied

We don't need to check the second condition for order 3, since the first condition for order 3 is already not satisfied. Thus, the method is of order 2. 1 pt for each check of the order conditions

TMA4320_2025v_code_solution

May 28, 2025

1 Problem 4 A Fourier spectral solver for the heat equation (15 pts)

Consider the **homogenenous** heat equation

$$\partial_t u - \Delta u = 0.$$

(proj:time-dep-biharmonic)

to be solved on a given rectangular domain $\Omega = [0, 2\pi) \times [0, 2\pi) \subset \mathbb{R}^2$ with periodic boundary conditions. Moreover, the problem is supplemented with **initial conditions** $u(0, x, y) = u_0(x, y)$, which we also assume to satisfy periodic boundary conditions.

1.1 Task 1

Provide a mathematical description of the numerical solver for the time-dependent heat equation which combines a Fourier spectral method/discrete Fourier transform in space with Crank-Nicolson (given in the previous problem) method in time. In particular, * explain how the (discrete) Fourier transform is used to transform the heat equation into an ODE in time * write down the equation for how a new solution in Fourier space is computed from the previous solution for each time step when the Crank-Nicolson method is a

(3 pts)

Solution:

a) Applying the Fourier transform to the $-\Delta u(x, y, t)$, we have in general

$$(\Delta u)^\wedge(k_x, k_y, t) = -\tilde{|\mathbf{k}|}^2 \hat{u}(\mathbf{k}, t).$$

where $\tilde{\mathbf{k}} = (k_x, k_y)$ and $\tilde{\mathbf{k}} = 2\pi(k_x/L_x, k_y/L_y)$ and (L_x, L_y) are the lengths of the domain in the x and y directions, respectively. (Note that in the given case, we have $L_x = L_y = 2\pi$, so the wave vector $\tilde{\mathbf{k}}$ is simply given by (k_x, k_y).) Thus

$$(\partial_t u - \Delta u)^\wedge(k_x, k_y, t) = \partial_t \hat{u}(k_x, k_y, t) + \tilde{|\mathbf{k}|}^2 \hat{u}(k_x, k_y, t) = 0.$$

This is now an ordinary differential equation (ODE) in time for each Fourier mode (k, l), which can be solved independently for each mode.

To discretize in space, we simply replace the continuous Fourier transform with a discrete Fourier transform (DFT).

1 pt for Fourier transform of Laplacian, 1 pt for replacement with DFT and final ODE

b) Applying the Crank-Nicolson method to ODE (assuming a constant time step τ), we compute the solution at t^{n+1} from the solution at t^n as follows

$$\hat{u}(k_x, k_y, t^{n+1}) + \frac{\tau}{2}|\tilde{\mathbf{k}}|^2 \hat{u}(k_x, k_y, t^{n+1}) = \hat{u}(k_x, k_y, t^n) - \frac{\tau}{2}|\tilde{\mathbf{k}}|^2 \hat{u}(k_x, k_y, t^n)$$

$$\hat{u}(k_x, k_y, t^{n+1}) = \frac{1 - \frac{\tau}{2}|\tilde{\mathbf{k}}|^2}{1 + \frac{\tau}{2}|\tilde{\mathbf{k}}|^2} \hat{u}(k_x, k_y, t^n)$$

1 pt for applying Crank-Nicolson and rearranging

1.2 Task 2

Implement the solver from the previous task by **completing the code snippet** provided below. As in Project 3, the solver is implemented as a **generator function** using the `yield` statement to return the **discrete Fourier transform** of the solution at each time step together with the current time.

Hint: Write the code as simply as possible, and in particular *taylor-made* for the specific problem at hand. There is not need to handle general non-zero right-hand side, or different domain sizes or spacing for each direction! That's why the signature of the generator function `heat_equation_solver` below is rather simple and short.

Hint: The numpy fuctions `linspace` and `meshgrid` might come in handy, as well as the function `fft2, ifft2, fftfreq` from the module `scipy.fft`.

Hint: Below, a plotting function is provided to visualize the solution at a given time step. You will use it in the 3rd task, but it might be useful to visually test your implementation in this task as well.

(6 pts) No detail breakdown for point distribution is given to allow for flexibility in grading when assessing individual implementations.

[78]:
```
import numpy as np
from scipy.fft import fft2, ifft2, fftfreq
```

Code outline:

[ ]:
```
def heat_equation_solver(N, d, U0, t0, T, Nt):
    """
    Solve the heat equation using the Crank-Nicolson method in Fourier space.

    Parameters:
    -----------
    N : int
        Number of grid points in each spatial direction (same for each␣
 ↪dimension)
    d : float (same for each dimension)
        Grid spacing.
    U0 : numpy.ndarray
        Initial condition evaluated on meshgrid.
```

```
    t0 : float
        Initial time.
    T : float
        Final time.
    Nt : int
        Number of time steps.

    Yields:
    -------
    tuple
        A tuple containing the discrete Fourier transform of the solution at␣
↪the current time step (U_hat)
        and the current time (t).
    """

    # Compute wave number grid
    # ...

    # Compute Fourier transform of initial value and yield initial solution for␣
↪convenience
    # ...
    yield U_hat, t

    # Time-stepping
    while t < T-dt/2:
        # Update U_hat and time step
        # ...

        yield U_hat, t
```

[75]:
```
def heat_equation_solver(N, d, U0, t0, T, Nt):
    """
    Solve the heat equation using the Crank-Nicolson method in Fourier space.

    Parameters:
    -----------
    N : int
        Number of grid points in each spatial direction (same for each␣
↪dimension)
    d : float (same for each dimension)
        Grid spacing.
    U0 : numpy.ndarray
        Initial condition evaluated on meshgrid.
    t0 : float
        Initial time.
    T : float
        Final time.
```

```
    Nt : int
        Number of time steps.

    Yields:
    -------
    tuple
        A tuple containing the discrete Fourier transform of the solution at␣
↪the current time step (U_hat)
        and the current time (t).
    """

    # Compute wave number grids
    kx = fftfreq(N,d=d)*2*np.pi
    ky = fftfreq(N,d=d)*2*np.pi
    KX, KY = np.meshgrid(kx, ky, sparse=True)
    K2 = KX**2 + KY**2

    # Define multiplier operator
    t = t0
    dt = (T-t0)/Nt
    mo = (1 - dt/2*K2)/(1+dt/2*K2)

    # Compute Fourier transform of initial value
    U_hat = fft2(U0)
    yield U_hat, t

    # Time-stepping
    while t < T-dt/2:
        t += dt
        U_hat = mo*U_hat
        yield U_hat, t
```

1.3 Task 3

Study the convergence order in time of your solver implementation using the manufactured solution function

$$u_{\text{ex}}(x, y, t) = \sin(x)\cos(y)\exp(-2t)$$

with satisfies the homogenenous heat equation.

Set * $N = 20$ sampling points/subintervals in each space direction * $t_0 = 0$, $T = 1$.

Now solve the problem successively for $N_t = 10, 20, 40, 80$ time steps with equidistant time steps $\tau = T/N_t$. For each run calculate the error in the so-called $L^\infty L^\infty$ norm defined by

$$\|E_k\|_{L^\infty L^\infty} = \max_{k\in\{0,N_t\}} \max_{i,j\in\{1,\dots,N\}} |u_{\text{ex}}(x_i, y_j, t_k) - U^k(x_j, y_j)|,$$

Print the errors and compute (or estimate) the experimentally observed convergence rate (EOC)

with respect to the time step size τ (Just do a simple print, no fancy formatting is needed!).

Finally, for $N_t = 80$, plot the exact solution u_{ex}, the discrete solution U and the error $E = U - U_{ex}$ at $t = 1$ using the `imshow_plot_u` defined below.

(6 pts) No detail breakdown for point distribution is given to allow for flexibility in grading when assessing individual implementations.

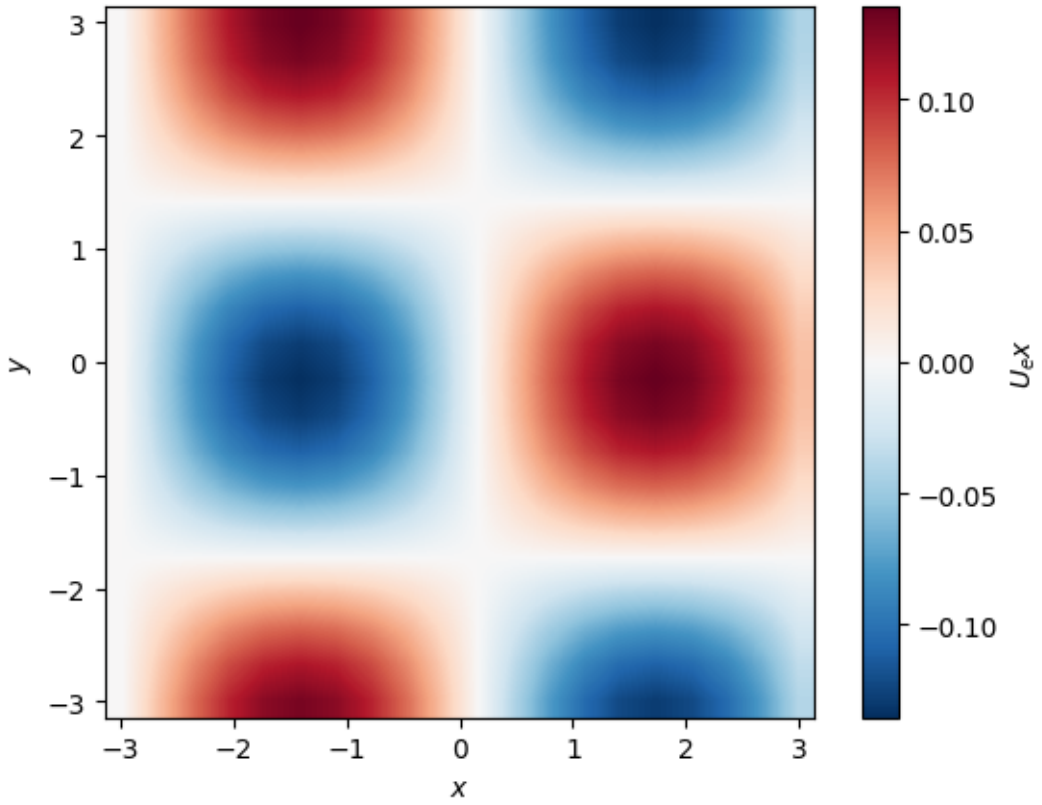
[64]:
```
import matplotlib.pyplot as plt

def imshow_plot_u(U, Lx, Ly, cblabel=r'$U$'):
    """
    Visualizes a 2D array `U` as a heatmap using matplotlib's imshow function.
    Parameters:
    -----------
    U : numpy.ndarray
        A 2D array representing the data to be visualized.
    Lx : float
        The length of the domain in the x-direction.
    Ly : float
        The length of the domain in the y-direction.
    cblabel : str, optional
        Label for the colorbar. Default is r'$U$'.
    Returns:
    --------
    fig : matplotlib.figure.Figure
        The figure object containing the plot.
    ax : matplotlib.axes._subplots.AxesSubplot
        The axes object containing the plot.
    Notes:
    ------
    - The colormap used is 'RdBu_r', which is a diverging colormap.
    - The color limits are set to the minimum and maximum values of `U`.
    - The x and y axes are labeled as r'$x$' and r'$y$', respectively.
    - The colorbar is added to the plot with the specified label.
    """

    fig = plt.figure()
    ax = fig.add_subplot(111)
    img = ax.imshow(U, cmap='RdBu_r', interpolation='bilinear', extent=[-Lx/2,␣
↪Lx/2, -Ly/2, Ly/2])
    ax.set_xlabel(r'$x$')
    ax.set_ylabel(r'$y$')
    cbar = plt.colorbar(img, ax=ax)
    cbar.set_label(cblabel)
    img.set_clim(vmin=U.min(), vmax=U.max())
    return fig, ax
```

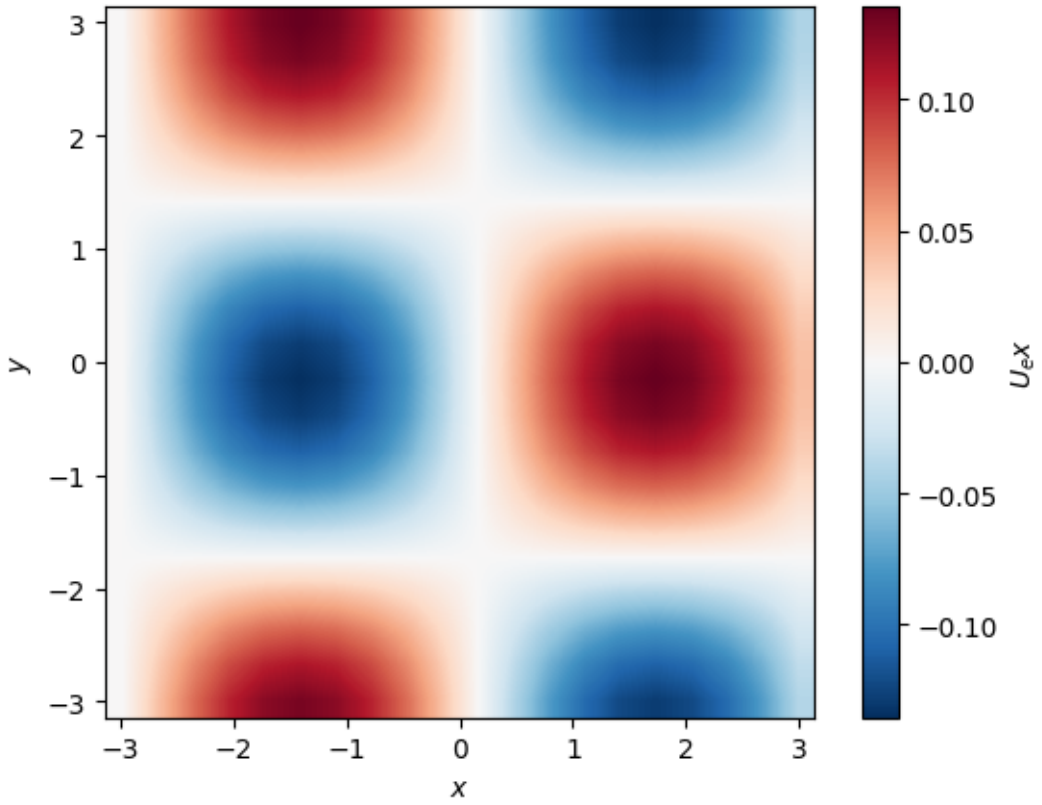
Solution

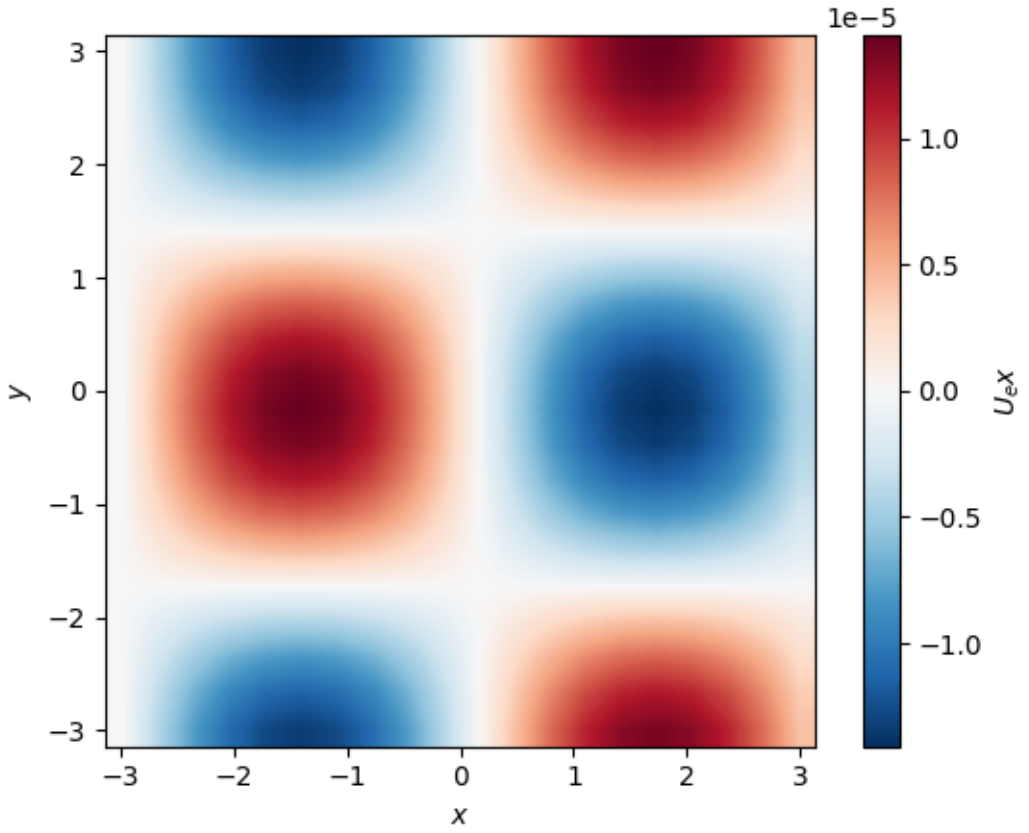
[ ]:
```python
def u_ex(x,y,t):
    return np.sin(x)*np.cos(y)*np.exp(-2*t)

N = 20
d = 2*np.pi/N

t0, T = 0, 1

# Prepare grid and initial data
x = np.linspace(0, 2*np.pi, N, endpoint=False)
y = np.linspace(0, 2*np.pi, N, endpoint=False)
X, Y = np.meshgrid(x,y, sparse=True)
U0 = u_ex(X, Y, t0)

err = []
for Nt in [10, 20, 40, 80]:
# for Nt in [10]:
    solver = heat_equation_solver(N, d, U0, t0, T, Nt)

    err_t = []
    for U_hat, t in solver:
        U = ifft2(U_hat).real
        U_ex = u_ex(X, Y, t)
        err_t.append(np.abs(U-U_ex).max())

    err_t = np.array(err_t)
    err.append(err_t.max())
    print(f"Err = {err[-1]}")

err = np.array(err)
eoc = np.log(err[:-1]/err[1:])/np.log(2)
print(eoc)

imshow_plot_u(U_ex, 2*np.pi, 2*np.pi, cblabel=r'$U_{ex}$')
imshow_plot_u(U, 2*np.pi, 2*np.pi, cblabel=r'$U$')
imshow_plot_u(U-U_ex, 2*np.pi, 2*np.pi, cblabel=r'$U_{ex}-U$')
```

```
Err = 0.0012316091182420497
Err = 0.0003068987885736507
Err = 7.666231473058005e-05
Err = 1.9161684992885508e-05
[2.0047096  2.00117349 2.00029313]
```

[ ]:
```
(<Figure size 640x480 with 2 Axes>, <Axes: xlabel='$x$', ylabel='$y$'>)
```


3
2
1
0
−1
−2
−3
y
−3
−2
−1
0
1
2
3
x
0.10
0.05
0.00
−0.05
−0.10
$U_e x$


3
2
1
0
−1
−2
−3
y
−3
−2
−1
0
1
2
3
x
0.10
0.05
0.00
−0.05
−0.10
U_eX


1e−5
3
2
1
0
−1
−2
−3
y
−3
−2
−1
0
1
2
3
x
1.0
0.5
0.0
−0.5
−1.0
U_{ex}