MATH 352 - Spring 2021
Homework 2

Dr. Alessandro Veneziani

Kai Chang, Kevin Hong, Rodrigo Gonzalez

---

1 Verify the exact solution

The exact solution is given by

$$u_{ex} = 1 + x^2 + \alpha y^2 + \beta t \tag{1}$$

The partials are

$$\frac{\partial^2 u_{ex}}{\partial x^2} = 2 \tag{2}$$

$$\frac{\partial^2 u_{ex}}{\partial y^2} = 2\alpha \tag{3}$$

$$\frac{\partial u_{ex}}{\partial t} = \beta \tag{4}$$

Then,

$$\frac{\partial u_{ex}}{\partial t} - \left(\frac{\partial^2 u_{ex}}{\partial x^2} + \frac{\partial^2 u_{ex}}{\partial y^2}\right) = \beta - (2 + 2\alpha) = f(x, y) \tag{5}$$

So the PDE is satisfied.

We check the initial condition

$$u_{ex}(x, y, 0) = 1 + x^2 + \alpha y^2 = u_0(x, y) \tag{6}$$

We check the boundary conditions

$$u_{ex}(0, y, t) = 1 + \alpha y^2 + \beta t = u_D(\partial\Omega) \quad (x = 0) \tag{7}$$

$$u_{ex}(1, y, t) = 2 + \alpha y^2 + \beta t = u_D(\partial\Omega) \quad (x = 1) \tag{8}$$

$$u_{ex}(x,0,t) = 1 + x^2 + \beta t = u_D(\partial\Omega) \quad (y=0) \tag{9}$$

$$u_{ex}(x,1,t) = 1 + x^2 + \alpha + \beta t = u_D(\partial\Omega) \quad (y=1) \tag{10}$$

2 Numerical Solution

Let $u^n_{i,j}$ be the numerical approximation for $u(x,y,t)$, where $x = i\Delta x = x_i$, $y = j\Delta y = y_j$, $t = n\Delta t = t_n$, and $i = 1,\ldots,m-1$, $j = 1,\ldots,k-1$. We approximate the spatial derivatives using the following second order formulas

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{x_i,y_j,t_n} = \frac{u^n_{i+1,j} - 2u^n_{i,j} + u^n_{i-1,j}}{\Delta x^2} \tag{11}$$

$$\left.\frac{\partial^2 u}{\partial y^2}\right|_{x_i,y_j,t_n} = \frac{u^n_{i,j+1} - 2u^n_{i,j} + u^n_{i,j-1}}{\Delta y^2} \tag{12}$$

Which means that the time derivative can be approximated as follows

$$\left.\frac{\partial u}{\partial t}\right|_{x_i,y_j,t_n} = \left(\frac{u^n_{i+1,j} - 2u^n_{i,j} + u^n_{i-1,j}}{\Delta x^2} + \frac{u^n_{i,j+1} - 2u^n_{i,j} + u^n_{i,j-1}}{\Delta y^2}\right) + f^n_{i,j} \tag{13}$$

We may simplify this equation by allowing $\Delta x = \Delta y = h$. Note that this implies that $k = m$. Then,

$$\left.\frac{\partial u}{\partial t}\right|_{x_i,y_j,t_n} = \frac{1}{h^2}\left(u^n_{i+1,j} - 2u^n_{i,j} + u^n_{i-1,j} + u^n_{i,j+1} - 2u^n_{i,j} + u^n_{i,j-1}\right) + f^n_{i,j} \tag{14}$$

Notice that if we fix $t = t_n$, and let i and j iterate, this scheme produces a system of $(m-1)^2$ linear equations, which we want to express in a matrix form. For this, define the following vector in $\mathbb{R}^{m-1}$

$$u^n_j = \begin{bmatrix} u^n_{1,j} \\ u^n_{2,j} \\ \vdots \\ u^n_{m-1,j} \end{bmatrix} \tag{15}$$

Note that this vector contains the numerical solution at $t = t_n$, $y = y_j$ and $x = x_i$ with $i = 1,\ldots,m-1$.

Now, we define

$$u^n = \begin{bmatrix} u^n_1 \\ u^n_2 \\ \vdots \\ u^n_{m-1} \end{bmatrix} \tag{16}$$

which is a vector in $\mathbb{R}^{(m-1)^2}$ that stands for the solution at time $t = t_n$.

Then, the matrix representation of the problem becomes

$$\left.\frac{\partial u}{\partial t}\right|_{t_n} = \mathcal{A}u^n + f^n \tag{17}$$

Where $\mathcal{A}$ is the $(m-1)^2$ by $(m-1)^2$ block matrix defined as follows

$$\mathcal{A} = \frac{1}{h^2}\begin{bmatrix} A & I & 0 & 0 & ... & 0 \\ I & A & I & 0 & ... & 0 \\ 0 & I & A & I & ... & 0 \\ ... & ... & ... & ... & ... & ... \\ ... & ... & ... & ... & ... & ... \\ 0 & 0 & ... & ... & I & A \end{bmatrix} \tag{18}$$

Where both I, and A are $m-1$ by $m-1$ matrices, I is the identity, and A is given by

$$A = \begin{bmatrix} -4 & 1 & 0 & 0 & ... & 0 \\ 1 & -4 & 1 & 0 & ... & 0 \\ 0 & 1 & -4 & 1 & ... & 0 \\ ... & ... & ... & ... & ... & ... \\ ... & ... & ... & ... & ... & ... \\ 0 & 0 & ... & ... & 1 & -4 \end{bmatrix} \tag{19}$$

Now we try to determine f^n. We fix n and let

$$u_{i,j} = u_{i,j}^n = u(x_i, y_j, t_n)$$

where $x_i = i \cdot h$, $y_j = j \cdot h$, and $t_n = n \cdot \Delta t$. By the boundary conditions that are given, we have

$$\begin{cases} u_{0,j}^n = 1 + \alpha(jh)^2 + \beta(n\Delta t) \\ u_{i,0}^n = 1 + (ih)^2 + \beta(n\Delta t) \\ u_{m,j}^n = 2 + \alpha(jh)^2 + \beta(n\Delta t) \\ u_{i,m}^n = 1 + \alpha + (ih)^2 + \beta(n\Delta t) \end{cases}$$

Based on the way of how the system of equations and the matrix $\mathcal{A}$ are constructed, it is necessary to deal with the cases of $i = 1$, $j = 1$, $i = m-1$, and $j = m-1$ respectively. We then modify the value of corresponding entries of f^n.

For each row of blocks, j is fixed. Therefore, the cases of $i = 1$ correspond to the first row of each block row. When $i = 1$, we have

$$\left.\frac{\partial u}{\partial t}\right|_{x_1, y_j, t_n} = \frac{1}{h^2}\left(u_{2,j}^n - 2u_{1,j}^n + u_{0,j}^n + u_{i,j+1}^n - 2u_{1,j}^n + u_{1,j-1}^n\right) + f_{1,j}^n$$

However, we do not include the coefficients of $u_{0,j}^n$ in $\mathcal{A}$. Therefore, we need to add $\frac{1}{h^2}u_{0,j}^n$ to $f_{0,j}^n$.

Similarly, when $j = 1$, the equation becomes

$$\left.\frac{\partial u}{\partial t}\right|_{x_i,y_1,t_n} = \frac{1}{h^2}\left(u^n_{i+1,1} - 2u^n_{i,1} + u^n_{i-1,1} + u^n_{i,2} - 2u^n_{i,1} + u^n_{i,0}\right) + f^n_{i,1}$$

for which we need to add $\frac{1}{h^2}u^n_{i,0}$ to $f^n_{i,0}$.
When $i = m-1$, the equation becomes

$$\left.\frac{\partial u}{\partial t}\right|_{x_{m-1},y_j,t_n} = \frac{1}{h^2}\left(u^n_{m,j} - 2u^n_{m-1,j} + u^n_{m-2,j} + u^n_{m-1,j+1} - 2u^n_{m-1,j} + u^n_{m-1,j-1}\right) + f^n_{1,j}$$

for which we need to add $\frac{1}{h^2}u^n_{m,j}$ to $f^n_{m-1,j}$.
When $j = m-1$, the equation becomes

$$\left.\frac{\partial u}{\partial t}\right|_{x_i,y_{m-1},t_n} = \frac{1}{h^2}\left(u^n_{i+1,m-1} - 2u^n_{i,m-1} + u^n_{i-1,m-1} + u^n_{i,m} - 2u^n_{i,m-1} + u^n_{i,m-2}\right) + f^n_{i,m-1}$$

for which we need to add $\frac{1}{h^2}u^n_{i,m}$ to $f^n_{i,m-1}$.
We now use a θ-method to approximate the time derivative.
Using explicit Euler,

$$\frac{u^{n+1} - u^n}{\Delta t} = \mathcal{A}u^n + f^n \tag{20}$$

Using implicit Euler,

$$\frac{u^{n+1} - u^n}{\Delta t} = \mathcal{A}u^{n+1} + f^{n+1} \tag{21}$$

Now, let $\theta \in [0,1]$, multiplying EE by θ, IE by $(1-\theta)$, and adding the results, we get

$$B_\theta u^{n+1} = C_\theta u^n + d \tag{22}$$

where

$$B_\theta = I - \Delta t\theta\mathcal{A} \tag{23}$$

$$C_\theta = I + \Delta t(1-\theta)\mathcal{A} \tag{24}$$

and

$$d = \Delta t(1-\theta)f^n + \Delta t\theta f^{n+1} \tag{25}$$

Note that I refers to the $(m-1)^2$ by $(m-1)^2$ identity matrix in this case.

3 Verify Stability Condition

Let $\theta = 0$, $h = 0.1$, and $\Delta t = 0.1$. The following table shows the error as a function of the time step n.

	Error
0	0.000000e+00
1	2.930989e-14
2	1.072475e-12
3	6.221335e-11
4	4.049924e-09
5	3.033481e-07
6	2.252957e-05
7	1.662654e-03
8	1.232297e-01
9	9.195642e+00
10	6.854483e+02
11	5.106921e+04
12	3.804740e+06
13	2.835348e+08
14	2.113959e+10
15	1.577098e+12
16	1.177429e+14
17	8.797315e+15
18	6.578397e+17
19	4.923248e+19
20	3.687624e+21

Notice that the error starts with a small value, but it steadily increases to high values near the end, showing that the method is not stable.

In order to meet the stability criteria, we fix $h = 0.1$, but we reduce Δt, and compute the maximum error. The following table contains the results.

	dt	Error
0	0.001000	1.065814e-14
1	0.001375	1.687539e-14
2	0.001750	8.437695e-15
3	0.002125	3.552714e-15
4	0.002500	6.217249e-15
5	0.002875	2.787975e+49
6	0.003250	2.816893e+98
7	0.003625	4.886032e+127
8	0.004000	3.099361e+146
9	0.004375	5.043272e+158
10	0.004750	9.543952e+165
11	0.005125	5.959521e+169
12	0.005500	1.184210e+171
13	0.005875	7.166366e+171
14	0.006250	1.245202e+171
15	0.006625	7.496792e+169
16	0.007000	2.054251e+168
17	0.007375	1.310545e+166
18	0.007750	9.772262e+164
19	0.008125	2.062404e+162
20	0.008500	1.117959e+160
21	0.008875	1.456370e+157
22	0.009250	9.836860e+154
23	0.009625	9.406813e+151
24	0.010000	9.193268e+149

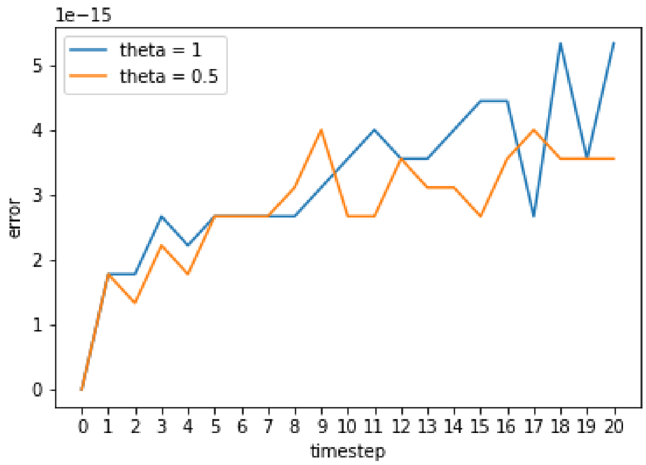
We notice that the method becomes stable for any Δt such that $\Delta t \leq 0.0025$. This implies that

$$\Delta t \leq Ch^2$$

, with a value of $C = 1/4$.

4 Ploting the Error

Using $h = 0.1$, $\Delta t = 0.1$, we graph the error for the two values of θ as a function of the time step n.



Notice that both errors remain small for any time value. This corresponds to the fact that any θ-method is unconditionally stable as long as $\theta \geq 1$ (even when the $\Delta t \leq Ch^2$ is not met).

Appendix

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.sparse as sp
import scipy
import scipy.sparse.linalg as spla
from google.colab import files
from numpy import linalg as LA
from mpl_toolkits.mplot3d import Axes3D
import pandas as pd

########## Matrix Builder ##########
# return a block diagonal matrix of size N**2 by N**2
# mu: constant
# h: size of space step
###########################################################################################
NOTE: the matrix returned is not the block matrix in the problem
      to get the matrix we need to multiply by mu/h^2
###########################################################################################
def matrix_builder(N):

  # construct matrix A
  A = sp.diags([1., -4., 1.], [-1, 0, 1], shape=[N, N], format = 'csr')

  # construct a zero matrix
  I0 = sp.diags([0.], shape=[N, N], format = 'csr')

  # construct an identity matrix
  I1 = sp.identity(N)

  # build the first and the last block row
  row1 = [A, I1] + [I0 for i in range(N-2)]
  row1 = sp.hstack(row1)
  rowN = [I0 for i in range(N-2)] + [I1, A]
  rowN = sp.hstack(rowN)

  # build the big matrix
  rows = [row1]
  for i in range(N - 2):
    temp_row = [I0 for i in range(i)] + [I1, A, I1] + [I0 for i in range(N - 3 - i)]
    temp_row = sp.hstack(temp_row)
    rows.append(temp_row)
  rows.append(rowN)

  return sp.vstack(rows)


########## Boundary Conditions ##########
# n: index of time step
# alpha, beta, mu: constants
# h: size of space step
# dt: size of time step
# i: index for x
```

```
# j: index for y

# return boundary case u(0, y_j, t_n)
def u_0j(alpha, beta, mu, j, h, n, dt):
  return mu/h**2 * (1 + alpha*(j*h)**2 + beta*(n*dt))

# return boundary case u(x_i, 0, t_n)
def u_i0(beta, mu, i, h, n, dt):
  return mu/h**2 * (1 + (i*h)**2 + beta*(n*dt))

# return boundary case u(m, y_j, t_n)
def u_mj(alpha, beta, mu, j, h, n, dt):
  return mu/h**2 * (2 + alpha*(j*h)**2 +beta*(n*dt))

# return boundary case u(x_i, m, t_n)
def u_im(alpha, beta, mu, i, h, n, dt):
  return mu/h**2 * (1 + alpha + (i*h)**2 + beta*(n*dt))


########## Constructor of fn ##########
# return a vector fn of length N^2
# n: index of time step
# alpha, beta, mu: constants
# h: size of space step
# dt: size of time step
##############################################################################################
def fn_constructor(alpha, beta, mu, h, dt, n, N):

  # value of function f
  fval = beta - 2 - 2*alpha

  # initialize fn as a column vector of N^2 with all of the entries being fval
  fn = np.full((N**2,1), fval)

  # modify the coefficients of u_0j
  for j in range(1, N+1):
    fn[(j-1)*N] = fn[(j-1)*N] + u_0j(alpha, beta, mu, j, h, n, dt)   # f[(j-1)*N] for 1 <= j <= N

  # modify the coefficients of u_i0
  for i in range(1, N+1):
    fn[i-1] = fn[i-1] + u_i0(beta, mu, i, h, n, dt)

  # modify the coefficients of u_mj
  for j in range(1, N+1):
    fn[j*N-1] = fn[j*N-1] + u_mj(alpha, beta, mu, j, h, n, dt)

  # modify the coefficients of u_im
  for i in range(1, N+1):
    fn[(N-1)*N-1+i] = fn[(N-1)*N+i-1] + u_im(alpha, beta, mu, i, h, n, dt)

  return fn

########## Initial Condition and Exact Solution ##########
# return u0(x_i, y_j)
```

```
def u0(h, i, j, alpha): return 1 + (i*h)**2 + alpha*(j*h)**2

# return u_ex(x_i, y_j, t_n)
def u_ex(h, i, j, alpha, beta, n, dt): return 1 + (i*h)**2 + alpha*(j*h)**2 + beta*(n*dt)

########## Solver for The Equation Induced by Theta-Method ##########
# T: final time
# Note: assume the initial time is 0
# return a matrix of size N^2 by TN+1 where TN is the number of time steps
###########################################################################################
def solver(alpha, beta, mu, h, dt, T, theta):
  m = int(1/h)
  N = m-1
  TN = int(T/dt) + 1 # number of timesteps (added the plus one to contain the initial condition)

  # initialize u_0
  u_0 = np.zeros((N**2,1))
  for j in range(1,N+1):
    for i in range(1,N+1):
      u_0[(j-1)*N-1+i] = u0(h, i, j, alpha)

  I = sp.identity(N**2)
  A = matrix_builder(N)

  # update matrix A
  A = mu/h**2 * A

  # define left-hand-side matrix B
  B = I - dt*theta*A

  # define right-hand-side matrix C
  C = I + dt*(1-theta)*A

  # build matrix with solution
  sol = np.zeros((N**2,TN))

  # initial condition
  sol[:,0] = u_0.ravel()

  #compute the solution (iterate over time)
  for n in range(1,TN):
    # update f_n & f_{n+1}
    fn_1= fn_constructor(alpha, beta, mu, h, dt, n-1, N)
    fn = fn_constructor(alpha, beta, mu, h, dt, n, N)

    # update vector d
    d = (dt*(1-theta)*fn_1 + dt*theta*fn).ravel()

    # compute solution
    sol[:,n] = spla.spsolve(B, C*sol[:,n-1]+d)

  return sol

########## Solver for The Exact Solution at All The Time Steps ##########
```

```
# return a matrix, each column of which corresponds to the solution at the time step
##########################################################################################
def solver_ex(alpha, beta, h, dt, T):
  m = int(1/h)
  N = m-1
  TN = int(T/dt) + 1 # number of timesteps (added the plus one to contain the initial condition)

  # initialize u_0
  u_0 = np.zeros((N**2,1))
  for j in range(1,N+1):
    for i in range(1,N+1):
      u_0[(j-1)*N-1+i] = u0(h, i, j, alpha)

  # build matrix with solution
  sol = np.zeros((N**2,TN))

  # initial condition
  sol[:,0] = u_0.ravel()

  # loop over the number of time steps to compute all the solutions
  for n in range(1,TN):
    temp_col = np.zeros((N**2,1))
    for j in range(1,N+1):
      for i in range(1,N+1):
        temp_col[(j-1)*N-1+i] = u_ex(h, i, j, alpha, beta, n, dt)
    sol[:,n] = temp_col.ravel()

  return sol

########## Error Computer ##########
# return a list containing the infinity-norm of the error
# between the numerical method and the exact solution at each time step
##########################################################################################
def errors(alpha, beta, mu, h, dt, T, theta):

  # number of time steps (including initial condition)
  TN = int(T/dt) + 1

  # compute the solution given by the theta-method
  disc = solver(alpha, beta, mu, h, dt, T, theta)

  # compute the exact solutions
  ex = solver_ex(alpha, beta, h, dt, T)

  return [max(np.abs(disc[:,i] - ex[:,i])) for i in range(TN)]

########## Define Problem ##########
alpha = 3
beta = 1.2
mu = 1
h = 0.1
dt = 0.1
m = int(1/h) # number of space steps
N = m-1 # side length of each block matrix
```

```
TN = int(1/dt) # number of time steps
theta = 0 # parameter theta in the theta-method

# calling errors(*args) gives the error
# calling solver(*args) gives the numerical solution
# calling solver_ex(*args) gives the exact solution
```