AMATH 586 Assignment 4

Tyler Chen

Problem 1

Consider the following method for solving the heat equation $u_t = u_{xx}$:

$$U_j^{n+2} = U_j^n + \frac{2k}{h^2}(U_{j-1}^{n+1} - 2U_j^{n+1} + U_{j+1}^{n+1}).$$

(a) Determine the formal order of accuracy of this method (in both space and time) based on computing the local truncation error.

(b) Suppose we take $k = \alpha h^2$ for some fixed $\alpha > 0$ and refine the grid. Show that this method fails to be Lax-Richtmyer stable for any choice of α.

Do this in two ways:

- Consider the MOL interpretation and the stability region of the time-discretization being used.
- Use von Neumann analysis and solve a quadratic equation for $g(\xi)$.

(c) What if we take $k = \alpha h^3$ for some fixed $\alpha > 0$ and refine the grid. Would this method be stable? Justify your answer.

Solution

(a) We have,

$$\frac{1}{2k}(u(x,t+2k) - u(x,t)) = \frac{1}{h^2}\left(u(x-h,t+k) - 2u(x,t+k) + u(x+h,t+k)\right) + \tau(x,t)$$

Therefore, by the definition of local truncation error,

$$\tau(x,t) = \frac{1}{2k}\left(u(x,t+2k) - u(x,t)\right) - \frac{1}{h^2}\left(u(x-h,t+k) - 2u(x,t+k) + u(x+h,t+k)\right)$$

Using Mathematica,

```
U[dx_, dt_] := Normal[Series[u[dx h z + x, dt k z + t], {z, 0, 4}]] /.
    {z -> 1}
LTE = FullSimplify
        [1/(2k) (U[0,2]-U[0,0])-1/h^2 (U[-1,1]-2U[0,1]+U[1,1]),
        Assumptions->{
            D[u[x,t],t]==D[u[x,t],{x,2}],
            D[u[x,t],{t,2}]==D[u[x,t],t,{x,2}],
            D[u[x,t],{t,3}]==D[u[x,t],{x,2},{t,2}]
        }
      ]
```

This gives,

$$\tau(x,t) = k^2\left(\frac{1}{6}u_{ttt}\right) - h^2\left(\frac{1}{12}u_{tt}\right) + \mathcal{O}(k^3) + \mathcal{O}(h^3)$$

(b) We can view the above method as the midpoint method applied to the system,

$$U_j'(t) = \frac{1}{h^2}(U_{j-1}(t) - 2U_j(t) + U_{j+1}(t))$$

In matrix notation,

$$U' = AU, \qquad A = \frac{1}{h^2}\begin{bmatrix} -2 & 1 & & \\ 1 & \ddots & \ddots & \\ & \ddots & & \end{bmatrix}$$

Note that A is normal as it is symmetric. Then, for stability of the midpoints method we require that for all eigenvalues λ of A,

$$|\operatorname{Im}(1+2k\lambda)| \le 1 \qquad \operatorname{Re}(1+2k\lambda) = 0$$

The eigenvalues of A are,

$$\lambda_p = \frac{2}{h^2}(\cos(p\pi h) - 1), \qquad p = 1, 2, \ldots, m$$

Then, taking $k = \alpha h^2$ for some $\alpha > 0$,

$$k\lambda_p = 2\alpha(\cos(p\pi h) - 1)$$

Clearly λ_p are all real and distinct. Therefore $1 + 2k\lambda_p$ cannot have zero real part for all λ_p. Therefore this method is not stable for any α. □

Set $U_j^n = e^{ijh\xi}$. Then, $U_j^{n+1} = g(\xi)e^{ijh\xi}$ and $U_j^{n+2} = g(\xi)^2 e^{ijh\xi}$. Inserting these into method we have,

$$g(\xi)^2 e^{ijh\xi} = e^{ijh\xi} + \frac{2k}{h^2}\left(g(\xi)e^{i(j-1)h\xi} - 2g(\xi)e^{ijh\xi} + g(\xi)e^{i(j+1)h\xi}\right)$$

Dividing through by $e^{ijh\xi}$, using the identity $e^{-ih\xi} + e^{ih\xi} = 2\cos(h\xi)$, and moving everything to the left gives,

$$\begin{aligned} 0 &= g(\xi)^2 - \frac{2k}{h^2}\left(e^{-ih\xi} - 2 + e^{ih\xi}\right)g(\xi) - 1 \\ &= g(\xi)^2 - \frac{4k}{h^2}(\cos(h\xi) - 1)g(\xi) - 1 \end{aligned}$$

This is a quadratic in $g(\xi)$ with solutions,

$$\begin{aligned} g(\xi) &= \frac{2k/h^2(\cos(h\xi) - 1) \pm \sqrt{(4k/h^2(\cos(h\xi) - 1))^2 - 4(1)(-1)}}{2} \\ &= \frac{k}{h^2}(\cos(h\xi) - 1) \pm \sqrt{\frac{2^2k^2}{h^4}(\cos(h\xi) - 1)^2 + 1} \end{aligned}$$

Note that $\sqrt{a^2+1} > 1$ so that if $a < 0$, $a/2 - \sqrt{a^2+1} < -1$.

Suppose $k = \alpha h^2$ for some $\alpha > 0$. Then $a = 2\alpha(\cos(h\xi) - 1)$. Clearly for sufficiently small h and any real ξ we have $\cos(h\xi) \to 1^-$ so $a < 0$. Thus,

$$|g(\xi)| = \left|\alpha(\cos(h\xi) - 1) - \sqrt{2^2\alpha^2(\cos(h\xi) - 1)^2 + 1}\right| > 1$$

Therefore there are values of ξ for which $|g(\xi)| > 1$. This proves the method is not stable when $k = \alpha h^2$ as $h \to 0$. □

(c) If we take $k = \alpha h^3$ then $a = h\alpha(\cos(h\xi) - 1)$. Again as $h \to 0$ we have $a < 0$. Therefore there are values of ξ for which $|g(\xi)| > 1$. This proves the method is not stable for $k = \alpha h^3$ as $h \to 0$. □

Problem 2

Consider the PDE

$$u_t = \kappa u_{xx} - \gamma u,$$

which models diffusion combined with decay provided $\kappa > 0$ and $\gamma > 0$. Consider methods of the form

$$U_j^{n+1} = U_j^n + \frac{k\kappa}{2h^2}[U_{j-1}^n - 2U_j^n + U_{j+1}^n + U_{j-1}^{n+1} - 2U_j^{n+1} + U_{j+1}^{n+1}] - k\gamma[(1-\theta)U_j^n + \theta U_j^{n+1}],$$

where θ is a parameter. In particular, if $\theta = 1/2$ then the decay term is modeled with the same centered-in-time approach as the diffusion term and the method can be obtained by applying the trapezoidal method to the MOL formulation of the PDE. If $\theta = 0$ then the decay term is handled explicitly. For more general reaction-diffusion equations it may be advantageous to handle the reaction terms explicitly since these terms are generally nonlinear, so making them implicit would require solving nonlinear systems at each time step (whereas handling the diffusion term implicitly only gives a linear system to solve at each time step).

(a) By computing the local truncation error, show that this method is $O(k^p + h^2)$ accurate, where $p = 2$ if $\theta = 1/2$ and $p = 1$ otherwise.

(b) Using von Neumann analysis, show that this method is unconditionally stable if $\theta \geq 1/2$.

(c) Show that if $\theta = 0$ then the method is stable provided $k \leq 2/\gamma$, independent of h.

Solution

(a) We have,

$$\frac{1}{k}(u(x,t+k) - u(x,t)) = \frac{\kappa}{2h^2}(u(x-h,t) - 2u(x+h,t) + u(x-h,t) + u(x-h,t+k) - 2u(x,t+k) + u(x+h,t+k)) - \gamma((1-\theta)u(x,t) + \theta u(x,t+k)) + \tau(x,t)$$

Therefore, by the definition of local truncation error,

$$\tau(x,t) = \frac{1}{k}(u(x,t+k) - u(x,t)) - \frac{\kappa}{2h^2}(u(x-h,t) - 2u(x+h,t) + u(x-h,t+k) + u(x-h,t+k) - 2u(x,t+k) + u(x+h,t+k)) + \gamma((1-\theta)u(x,t) + \theta u(x,t+k))$$

Using Mathematica,

```
U[dx_, dt_] := Normal[Series[u[dx h z + x, dt k z + t], {z, 0, 4}]] /.
    {z -> 1}
LTE = Collect[FullSimplify[
        1/k (U[0,1]-U[0,0])-\[Kappa]/(2h^2) (U[-1,0]-2U[0,0]+U[1,0]+U
            [-1,1]-2U[0,1]+U[1,1])+ \[Gamma]((1-\[Theta])U[0,0]+\[Theta]
             U[0,1]),
        Assumptions->{
            D[u[x,t],t]==\[Kappa] D[u[x,t],{x,2}]-\[Gamma] u[x,t],
            D[u[x,t],{t,2}]==\[Kappa] D[u[x,t],t,{x,2}]-\[Gamma] D[u[x,
                t],t]
        }
        ],{k,h},Simplify]
```

This gives,

$$\tau(x,t) = -k\left(\frac{(1-2\theta)\gamma}{2}u_t\right) + k^2\left(\frac{\gamma\theta}{2}u_{tt} + \frac{1}{6}u_{ttt} - \frac{\kappa}{4}u_{xxtt}\right) - h^2\left(\frac{\kappa}{12}u_{xxxx}\right) + \mathcal{O}(k^3) + \mathcal{O}(h^3)$$

Clearly the $\mathcal{O}(k)$ term vanishes if and only if $\theta = 1/2$. Therefore the local truncation error is $\mathcal{O}(k+h^2)$ if $\theta \neq 1/2$ and $\mathcal{O}(k^2+h^2)$ if $\theta = 1/2$. □

(b) Set $U_j^n = e^{ijh\xi}$. Then, $U_j^{n+1} = g(\xi)e^{ijh\xi}$. Inserting these into method we have,

$$\begin{aligned} g(\xi)e^{ijh\xi} = e^{ijh\xi} + \frac{k\kappa}{2h^2}\Big(&e^{i(j-1)h\xi} - 2e^{ijh\xi} + e^{i(j+1)h\xi} + g(\xi)e^{i(j-1)h\xi} \\ &- 2g(\xi)e^{ijh\xi} + g(\xi)e^{i(j+1)h\xi}\Big) - k\gamma\left((1-\theta)e^{ijh\xi} + \theta g(\xi)e^{ijh\xi}\right) \end{aligned}$$

Dividing through by $e^{ijh\xi}$ and grouping terms of $g(\xi)$,

$$g(\xi)\left(1 - \frac{k\kappa}{2h^2}\left(e^{-ih\xi} - 2 + e^{ih\xi}\right) + k\gamma\theta\right) = 1 + \frac{k\kappa}{2h^2}\left(e^{-ih\xi} - 2 + e^{ih\xi}\right) - k\gamma(1-\theta)$$

Using the fact that $e^{-ih\xi} + e^{ih\xi} = 2\cos(h\xi)$,

$$g(\xi) = \frac{1 + \frac{k\kappa}{h^2}\left(\cos(h\xi) - 1\right) - k\gamma(1-\theta)}{1 - \frac{k\kappa}{h^2}\left(\cos(h\xi) - 1\right) + k\gamma\theta}$$

Define $z = -k\kappa/h^2(\cos(h\xi) - 1)$. Note that $z < 0$.
We have we have $(z + k\gamma(1-\theta)), (z + k\gamma\theta) < 0$. Therefore, if $\theta \in [1/2, 1]$,

$$|g(\xi)| = \left|\frac{1 - (z + k\gamma(1-\theta))}{1 + (z + k\gamma\theta)}\right| \leq \left|\frac{1 - (z + k\gamma/2)}{1 + (z + k\gamma/2)}\right| \leq 1$$

This proves the method is unconditionally stable if $\theta \in [1/2, 1]$. □

(c) When $\theta = 0$ we have,

$$g(\xi) = \frac{1 + \frac{k\kappa}{h^2}\left(\cos(h\xi) - 1\right) - k\gamma(1-\theta)}{1 - \frac{k\kappa}{h^2}\left(\cos(h\xi) - 1\right) + k\gamma\theta} = \frac{1 - z - k\gamma}{1 + z}$$

Suppose $k \leq 2/\gamma$. Then, since $k \geq 0$,

$$-1 = \frac{-1-z}{1+z} = \frac{1-z-2}{1+z} \leq g(\xi) \leq \frac{1-z}{1+z} \leq 1$$

Then clearly $|g(\xi)| \leq 1$. This proves the method is stable if $\theta = 0$ and $k \leq 2/\gamma$ □

Problem 3

Download the code `heat_CN.m` from the textbook repository:

`http://faculty.washington.edu/rjl/fdmbook/matlab/heat_CN.m` .

This code solves the heat equation $u_t = \kappa u_{xx}$ (with $\kappa = 0.02$) using the Crank-Nicolson method. It is currently set up to solve a problem whose solution is known. You tell it the number of interior grid points m and it sets the time step $k = 4h$. It then runs the Crank-Nicolson scheme, generating an approximate solution and comparing it to the true solution.

(a) Run this code and, by changing the number of grid points, confirm that it is second-order accurate. (Observe how the error at some fixed time such as $T = 1$ behaves as k and h go to zero with a fixed relation between k and h such as $k = 4h$, as currently set in the code.) [Note that the parameter m in the code is the number of *interior* grid points, so, for example, set $m = 19$ to get $h = 1/20$ and $k = 1/5$.] Produce a log-log plot of the error (at the final time T) versus h.

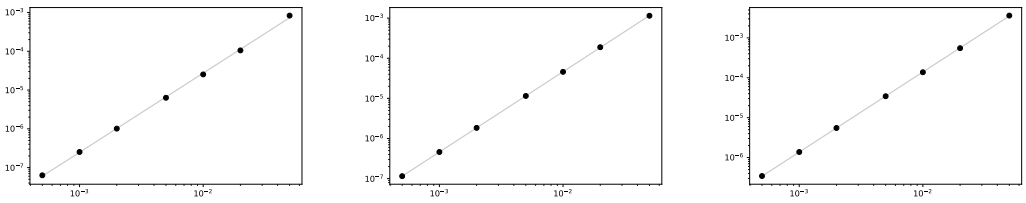
(b) Modify this code to produce a new version that implements the TR-BDF2 method on the same problem. Test it to confirm that it is also second order accurate. Explain how you determined the proper boundary conditions in each stage of this method.

(c) Modify the code to produce a new m-file `heat_FE` that implements the forward Euler method on the same problem. Test it to confirm that it is $O(h^2)$ accurate as $h \to 0$ when $k = 24h^2$ is used as the time step. Verify that this is within the stability limit for $\kappa = 0.02$. [Note how many more time steps are required compared to Crank-Nicolson or TR-BDF2, especially on finer grids.]

(d) Test `heat_FE` with $k = 26h^2$, for which it should be unstable. Note that the instability does not become apparent until about time $t = 4.5$ for the parameter values $\kappa = 0.02$, $m = 39$, $\beta = 150$. Explain why the instability takes several hundred time steps to appear and why it appears as a sawtooth oscillation. [Hint: What wave numbers ξ are growing exponentially for these parameter values? What is the initial magnitude of the most unstable eigenmode in the given initial data? The expression (E.30) for the Fourier transform of a Gaussian may be useful.]

Solution

(a) Figure 1a shows a plot the infinity norm of the error at ($T = 1$). The slope of the log-log plot is 2.0430 suggesting $\mathcal{O}(h^2)$ convergence when $k = 4h$.



(a) Trapezoid Rule (b) TR-BDF2 (c) Forward Euler

Figure 1: Infinity norm of error at $T = 1$ vs. mesh size $h = 1/(m + 1)$ when $k = 4h$. Linear fit to log-log plot shown in grey

(b) Our system is of the form,

$$U'(t) = f(U(t), t) = AU(t) + g(t)$$

In particular we take A to be κD_2 and $g = (\kappa/h^2)[u(0,t), \ldots, u(1,t)]^T$.

The TR-BDF2 method is defined as,

$$U^* = U^n + \frac{k}{4}(f(U^n, t_n) + f(U^*, t_{n+1/2}))$$
$$U^{n+1} = \frac{1}{3}\left(4U^* - U^n + kf(U^{n+1}, t_{n+1})\right)$$

For our equations we have,

$$U^* = U^n + \frac{k}{4}\left(AU^n + g_n + AU^* + g_*\right) \quad \Longrightarrow \quad \left(I - \frac{k}{4}A\right)U^* = \left(I + \frac{k}{4}A\right)U^n + \frac{k}{4}\left(g_n + g_*\right)$$
$$U^{n+1} = \frac{1}{3}\left(4U^* - U^n + k(AU^{n+1} + g_{n+1})\right) \quad \Longrightarrow \quad \left(I - \frac{k}{3}A\right)U^{n+1} = \frac{1}{3}(4U^* - U^n) + \frac{k}{3}g_{n+1}$$

We know that U^* represents a solution at time $t_{n+1/2} = t_n + k/2$. We implement this in Python and iterate over various mesh sizes to generate Figure 1b.

Figure 1 shows a plot the infinity norm of the error at $(T = 1)$. The slope of the log-log plot is 2.0012 suggesting $\mathcal{O}(h^2)$ convergence when $k = 4h$.

(c) We have the same system as above. Using forward Euler we have,

$$U^{n+1} = U^n + kf(U^n) = U^n + kAU^n + kg_n$$

We implement this in Python and iterate over various mesh sizes to generate Figure 1c.

Note that the eigenvalues of A are,

$$\lambda_p = \frac{2\kappa}{h^2}(\cos(ph\pi) - 1)$$

Recall that forward Euler applied to the wave equation is stable provided,

$$\frac{k\kappa}{h^2} \leq \frac{1}{2}$$

Clearly if $\kappa = 1/50$ and $k = 24h^2$ this equation is satisfied.

Figure 1 shows a plot the infinity norm of the error at $(T = 1)$. The slope of the log-log plot is 2.0073 suggesting $\mathcal{O}(h^2)$ convergence when $k = 24h^2$.

(d) If $\kappa = 1/50$ and $k = 26h^2$ the condition for stability is not satisfied. We set $k = 26h^2$ and run to time $T = 5$ using $m = 39$ interior mesh points. The solution is shown in Figure 2. Clearly the solution is no longer converging as $k\kappa/h^2 > 1/2$.

In particular,

$$|g(\xi)| = \left|1 + 2\frac{k\kappa}{h^2}\cos(\xi h) - 1)\right| = \left|1 + \frac{52}{50}(\cos(\xi h) - 1)\right|$$

will not be less than one in magnitude for $\xi \notin (-\pi/h, \pi/h)$. This means that the method is unstable and we expect it to "blow up" at some point. Moreover, the ξ for which the method is unstable are the highest frequency modes of the DFT. This mode which alternates between 1 and -1 at each point explaining the alternating behavior in space seen in our solution.

This behavior is not seen until a later time because the weight of eigenmodes corresponding to unstable ξ are small initially. More specifically, the modulus of the entries of the DFT of the Gaussian initial condition corresponding to high frequency modes are small initially . This means that it will take a while for these modes to blow up to a level which can be seen over the other stable modes.

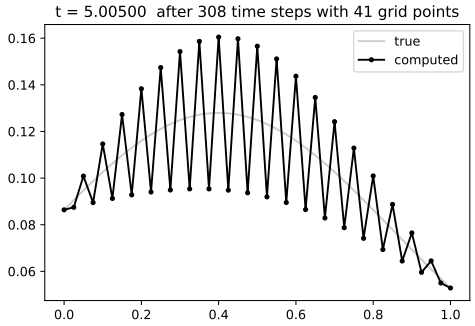


Figure 2: Solution using forward Euler with $k = 26h^2$

```
def heat_CN_TRBDF2(m):
#
# heat_CN.py
#
# Solve u_t = kappa * u_{xx} on [ax,bx] with Dirichlet boundary
    conditions,
# using the Crank-Nicolson method with m interior points.
#
# Returns k, h, and the max-norm of the error.
# This routine can be embedded in a loop on m to test the accuracy,
# perhaps with calls to error_table and/or error_loglog.
#
# Original MATLAB code from  http://www.amath.washington.edu/~rjl/
    fdmbook/  (2007)
# Ported to Python by Tyler Chen (2018)

    plt.figure()              # clear graphics
                              # Put all plots on the same graph (
                                  comment out if desired)

    ax = 0;
    bx = 1;
    kappa = .02;                 # heat conduction coefficient:
    tfinal = 1;                  # final time

    h = (bx-ax)/(m+1);           # h = delta x
    x = np.linspace(ax,bx,m+2);  # note x(1)=0 and x(m+2)=1
```

```
                                # u(1)=g0 and u(m+2)=g1 are known from
                                    BC's
k = 4*h;                        # time step

nsteps = round(tfinal / k);  # number of time steps

#nplot = 1;      # plot solution every nplot time steps
                 # (set nplot=2 to plot every 2 time steps, etc.)
nplot = nsteps;  # only plot at final time

if abs(k*nsteps - tfinal) > 1e-5:
    # The last step won't go exactly to tfinal.
    print(' ')
    print('WARNING *** k does not divide tfinal, k = %1.5f' % k)
    print(' ')

# true solution for comparison:
# For Gaussian initial conditions u(x,0) = exp(-beta * (x-0.4)^2)
beta = 150;
utrue = lambda x,t: np.exp(-(x-0.4)**2 / (4*kappa*t + 1/beta)) / np
    .sqrt(4*beta*kappa*t+1);

# initial conditions:
u0 = utrue(x,0);

# Each time step we solve MOL system U' = AU + g using the TRBDF2

# set up matrices:
r = kappa * k/(h**2);
e = np.ones(m);
A = sparse.spdiags([e,-2*e,e],[-1,0,1],m,m)
A1_ = sparse.eye(m) + (r / 4) * A;
A2_ = sparse.eye(m) - (r / 4) * A;
A2 = sparse.eye(m) - (r / 3) * A;

# initial data on fine grid for plotting:
xfine = np.linspace(ax,bx,1001);
ufine = utrue(xfine,0);

# initialize u and plot:
tn = 0;
u = u0;

plt.plot(x,u,'b.-', xfine,ufine,'r')
plt.legend(['computed','true'])
plt.title('Initial data at time = 0')


# main time-stepping loop:
for n in range(nsteps):
    tnp = tn + k;   # = t_{n+1}
    # boundary values u(0,t) and u(1,t) at times tn and tnp:

    g0n = u[0];
    g1n = u[m+1];
    g0n_ = utrue(ax,tn+k/2);
```

```
        g1n_ = utrue(bx,tn+k/2);
        g0np = utrue(ax,tnp);
        g1np = utrue(bx,tnp);

        # compute right hand side for intermediate linear system:
        uint = u[1:-1];   # interior points (unknowns)
        rhs_ = A1_ @ uint;
        # fix-up right hand side using BC's (i.e. add vector g to A2*
            uint)
        rhs_[0] += (r / 4) * (g0n + g0n_);
        rhs_[m-1] += (r / 4) * (g1n + g1n_);

        # solve intermediate linear system:
        uint_ = sparse.linalg.spsolve(A2_,rhs_);

        # compute right hand side for linear system:
        rhs = (4 * uint_ - uint) / 3
        rhs[0] += (r / 3) * g0np;
        rhs[m-1] += (r / 3) * g1np;

        # solve linear system:
        uint = sparse.linalg.spsolve(A2,rhs)

        # augment with boundary values:
        u = np.concatenate([[g0np], uint, [g1np]]);
        # plot results at desired times:
        if (n+1)%nplot==0 or (n+1)==nsteps:
            print(n)
            ufine = utrue(xfine,tnp);
            plt.plot(x,u,'b.-', xfine,ufine,'r')
            plt.title('t = %1.5f  after %i time steps with %i grid
                points' % (tnp,n+1,m+2))
            error = max(abs(u-utrue(x,tnp)));
            print('at time t = %.5f  max error =  %.5f'%(tnp,error))
            if (n+1)<nsteps: input('Hit <return> to continue  ')

        tn = tnp;   # for next time step
    plt.show()

    return k,h,error
```

```
def heat_CN_FWE(m):
#
# heat_CN.py
#
# Solve u_t = kappa * u_{xx} on [ax,bx] with Dirichlet boundary
    conditions,
# using the Crank-Nicolson method with m interior points.
#
# Returns k, h, and the max-norm of the error.
# This routine can be embedded in a loop on m to test the accuracy,
# perhaps with calls to error_table and/or error_loglog.
#
# Original MATLAB code from  http://www.amath.washington.edu/~rjl/
    fdmbook/  (2007)
```

```
# Ported to Python by Tyler Chen (2018)

    plt.figure()              # clear graphics
                              # Put all plots on the same graph (
                                  comment out if desired)

    ax = 0;
    bx = 1;
    kappa = .02;                 # heat conduction coefficient:
    tfinal = 1;                  # final time

    h = (bx-ax)/(m+1);           # h = delta x
    x = np.linspace(ax,bx,m+2);  # note x(1)=0 and x(m+2)=1
                                 # u(1)=g0 and u(m+2)=g1 are known from
                                     BC's
    k = 24*h**2;                 # time step

    nsteps = round(tfinal / k);  # number of time steps

    #nplot = 1;      # plot solution every nplot time steps
                     # (set nplot=2 to plot every 2 time steps, etc.)
    nplot = nsteps;  # only plot at final time

    if abs(k*nsteps - tfinal) > 1e-5:
        # The last step won't go exactly to tfinal.
        print(' ')
        print('WARNING *** k does not divide tfinal, k = %1.5f' % k)
        print(' ')

    # true solution for comparison:
    # For Gaussian initial conditions u(x,0) = exp(-beta * (x-0.4)^2)
    beta = 150;
    utrue = lambda x,t: np.exp(-(x-0.4)**2 / (4*kappa*t + 1/beta)) / np
        .sqrt(4*beta*kappa*t+1);

    # initial conditions:
    u0 = utrue(x,0);

    # Each time step we solve MOL system U' = AU + g using the TRBDF2

    # set up matrices:
    r = kappa * k/(h**2);
    e = np.ones(m);
    A = sparse.spdiags([e,-2*e,e],[-1,0,1],m,m)
    A1_ = sparse.eye(m) + (r / 4) * A;
    A2_ = sparse.eye(m) - (r / 4) * A;
    A2 = sparse.eye(m) - (r / 3) * A;

    # initial data on fine grid for plotting:
    xfine = np.linspace(ax,bx,1001);
    ufine = utrue(xfine,0);

    # initialize u and plot:
    tn = 0;
    u = u0;
```

```
    plt.plot(x,u,'b.-', xfine,ufine,'r')
    plt.legend(['computed','true'])
    plt.title('Initial data at time = 0')

    # main time-stepping loop:
    for n in range(nsteps):
        tnp = tn + k;   # = t_{n+1}
        # boundary values u(0,t) and u(1,t) at times tn and tnp:

        g0n = u[0];
        g1n = u[m+1];
        g0np = utrue(ax,tnp);
        g1np = utrue(bx,tnp);

        # compute right hand side for intermediate linear system:
        uint = u[1:-1];   # interior points (unknowns)
        rhs = r*A @ uint;
        # fix-up right hand side using BC's (i.e. add vector g to A2*
            uint)
        rhs[0] += r * g0n;
        rhs[m-1] += r * g1n;

        uint += rhs

        # augment with boundary values:
        u = np.concatenate([[g0np], uint, [g1np]]);
        # plot results at desired times:
        if (n+1)%nplot==0 or (n+1)==nsteps:
            print(n)
            ufine = utrue(xfine,tnp);
            plt.plot(x,u,'b.-', xfine,ufine,'r')
            plt.title('t = %1.5f  after %i time steps with %i grid
                points' % (tnp,n+1,m+2))
            error = max(abs(u-utrue(x,tnp)));
            print('at time t = %.5f  max error =  %.5f'%(tnp,error))
            if (n+1)<nsteps: input('Hit <return> to continue  ')

        tn = tnp;   # for next time step
    plt.show()

    return k,h,error
```