Solving Nonlinear Parabolic Equations

Rong Jiang

October 16, 2003

Contents

List of Tables

List of Figures

1 Problem

In this project[1], we first consider a nonlinear parabolic problem of the form

$$u_t = (u^2)_{xx}, \quad t \in [\,0,1\,], \quad x \in (\,0,1\,) \tag{1}$$

where the initial conditions and Dirichlet boundary conditions correspond to the exact solution given implicitly by $(2u-3)+\log(u-1/2) = 2(2t-x)$. We solve this *1-dim* problem numerically using *Newton's Linearization Method* with *Crank-Nicolson Method*. In the meanwhile, we vary both the spatial stepsize h and the temporal stepsize k, and derive the dependencies of accuracy vs.h and k, as well as the running time.

Secondly, we generalize to the *2-dim* parabolic problem of the form

$$u_t = \Delta(u^2), \quad t \in [\,0,1\,], \quad (\,x,y\,) \in (\,0,1\,)^2. \tag{2}$$

Here the initial and boundary conditions are given by the function, $u(t,x,y) = \sin x \sin y \exp(-2t)$.

2 Approach

2.1 Finite Difference and Crank-Nicolson Method

Given a function $U(x)$, we have[2] below the *forward-difference* formula for the first derivative and the *central-difference* formula for the second derivative, respectively:

$$U'(x) \simeq [U(x+h) - U(x)]/h, \tag{3}$$

$$U''(x) \simeq [U(x+h) - 2U(x) + U(x-h)]/h^2. \tag{4}$$

We can apply the above approximations for eqn (1) and (2), which gives rise to two kinds of methods, explicit or implicit. The explicit method is computationally simple, however, it has one serious drawback, that is, we require for the stability that $\lambda = k/h^2 \leq 1/2$, where $k = \delta t$ and $h = \delta x$. To avoid restriction from stability, we apply the *Crank-Nicolson implicit method*. For simplicity, we first concern 1D case. We consider the partial differential equation as being satisfied at the midpoint $\{ih, (j+1/2)k\}$ and replaced $\partial^2 U/\partial x^2$ by the mean of its central-difference approximation at the jth and $(j+1)$th time-levels. In other words, we approximate the equation

$$\left(\frac{\partial U}{\partial t}\right)_{i,j+\frac{1}{2}} = \left(\frac{\partial^2 U}{\partial x^2}\right)_{i,j+\frac{1}{2}} \tag{5}$$

by

$$\frac{u_{i,j+1} - u_{i,j}}{k} = \frac{1}{2}\left\{\frac{u_{i+1,j+1} - 2u_{i,j+1} + u_{i-1,j+1}}{h^2} + \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}\right\}. \tag{6}$$

Therefore, we need to solve a system of linear equations for u at the $(j+1)$th time-level given computed u at the jth time-level.

[1] This is the term project in math600, *Special Topics in Numerical Analysis*, given by Professor Proskurowski, USC, Summer 2003.

[2] See Smith [1], page 7 and 20.

2.2 Newton's Linearization Method

Supppose we have a system of N (nonlinear) equations[3] in the N dependent variables $u_1, u_2, \ldots, u_N$:

$$f_i(u_1, u_2, \ldots, u_N) = 0, \qquad i = 1(1)N, \tag{7}$$

Let V_i be a known approximation to the exact solution value u_i, $i = 1(1)N$. Put $u_i = V_i + \epsilon_i$ and substitute into eqn (7). Then by Taylor's expansion to the first-order terms in ϵ_i, $i = 1(1)N$,

$$f_i(V_1, V_2, \ldots, V_N) + \left[\frac{\partial f_i}{\partial u_1}\epsilon_1 + \frac{\partial f_i}{\partial u_2}\epsilon_2 + \ldots + \frac{\partial f_i}{\partial u_N}\epsilon_N\right]_{u_i = V_i} = 0, \tag{8}$$

The subscript notation on the second bracket indicates that the dependent variables $u_1, u_2, \ldots, u_N$ appearing in the coefficients of $\epsilon_1, \epsilon_2, \ldots, \epsilon_N$ are replaced by $V_1, V_2, \ldots, V_N$ respectively after the differentiations. We solve (8) for ϵ's then update $V_i = V_i + \epsilon_i$, and repeat this until we reach certain degree of accuracy, say $||\epsilon|| < 10^{-8}$. In addition, it usually takes 2 or 3 Newton iterations at each time-level.

2.3 1D Case

For the 1D problem (1), we subdivide the x-t plane into sets of equal rectangles of sides $\delta x = h$, $\delta t = k$, and denote $x_i = ih$ and $t_j = jk$. Moreover, denote the value of u at the mesh point (ih, jk) by $u(ih, jk) = u_{i,j}$. Thus, the approximation equation becomes eqn (6). Put $p = h^2/k$ and denote $u_{i,j+1} = u_i$. We have

$$u_{i-1}^2 - 2(u_i^2 + pu_i) + u_{i-1}^2 + \{u_{i-1,j}^2 - 2(u_{i,j}^2 - pu_{i,j}) + u_{i-1,j}^2\} = 0 \equiv f_i(u_{i-1}, u_i, u_{i+1}). \tag{9}$$

By eqn (8), we have

$$2V_{i-1}\epsilon_{i-1} - 2(2V_i + p)\epsilon_i + 2V_{i+1}\epsilon_{i+1} + \{V_{i-1}^2 - 2(V_i^2 + pV_i) + V_{i+1}^2\} + \{u_{i-1,j}^2 - 2(u_{i,j}^2 - pu_{i,j}) + u_{i+1,j}^2\} = 0, \tag{10}$$

where V_i is an approximation to $u_{i,j+1}$.

We can write eqn (10) in matrix form, i.e., $A_{j+1}V_{j+1} = b_{j+1}$. Unlike the situation we have in the linear 1D parabolic equation, $u_t = u_{xx}$, where the matrix A is an *invariable* tridiagonal matrix, our matrix A_{j+1} in each iteration is constructed by the current approximation vector V and the vector b is generated by two parts, variable part from V and invariable part from u at the jth time-level.

[3] See Smith [1], page 142–3.

2.4 2D Case

For the 2D problem (2), we have the similar iteration scheme as for the 1D problem. First we subdivide the x-y plane into sets of equal rectangles of sides $\delta x = \delta y = h$. Then discretize time t as $\delta t = k$. Now we use index i, j, l for x, y, t, respectively. Since we need to approximate both $(u^2)_{xx}$ and $(u^2)_{yy}$, our approximation equation becomes

$$\frac{u_{i,j,l+1} - u_{i,j,l}}{k} = \frac{1}{2h^2}[\{\frac{u_{i+1,j,l+1} - 2u_{i,j,l+1} + u_{i-1,j,l+1}}{h^2} + \frac{u_{i+1,j,l} - 2u_{i,j,l} + u_{i-1,j,l}}{h^2}\} + \{\frac{u_{i,j+1,l+1} - 2u_{i,j,l+1} + u_{i-1,j,l+1}}{h^2} + \frac{u_{i,j-1,l} - 2u_{i,j,l} + u_{i,j+1,l}}{h^2}\}]. \quad (11)$$

Similarly, put $p = h^2/k$ and denote $u_{i,j,l+1} = u_{i,j}$, we have

$$\begin{aligned} &- u^2_{i-1,j} - u^2_{i+1,j} + (4u^2_{i,j} + 2pu_{i,j}) - u^2_{i,j-1} - u^2_{i,j+1} \\ &\quad - [u^2_{i-1,j,l} + u^2_{i+1,j,l} - (4u^2_{i,j,l} - 2pu_{i,j,l}) + u^2_{i,j-1,l} + u^2_{i,j+1,l}] \\ &\quad = 0 \equiv f_i(u_{i-1,j}, u_{i+1,j}, u_{i,j}, u_{i,j-1}, u_{i,j+1}). \end{aligned} \quad (12)$$

By eqn (8), we obtain equations for vector ϵ with currecnt vector V at the $(l+1)$th time-level,

$$\begin{aligned} &- 2V_{i-1,j}\epsilon_{i-1,j} - 2V_{i+1,j}\epsilon_{i+1,j} + (8V_{i,j} + 2p)\epsilon_{i,j} - 2V_{i,j-1}\epsilon_{i,j-1} - 2V_{i,j+1}\epsilon_{i,j+1} \\ &\quad = [V^2_{i-1,j} + V^2_{i+1,j} - (4V^2_{i,j} + 2pV_{i,j}) + V^2_{i,j-1} + V^2_{i,j+1}] + \\ &\quad [u^2_{i-1,j,l} + u^2_{i+1,j,l} - (4u^2_{i,j,l} - 2pu_{i,j,l}) + u^2_{i,j-1,l} + u^2_{i,j+1,l}]. \end{aligned} \quad (13)$$

We can write eqn (13) in matrix form, i.e., $A_{l+1}V_{l+1} = b_{l+1}$. Instead of a tridiagonal matrix in 1D problem, we have a pentadiagonal matrix A, whose dimension is the square of the number of steps in either x or y. In other words, the matrix size in 2D is the square of the one in 1D choosing same spatial stepsize h, so solving 2D problem requires much more running time as well as machine memory.

2.5 Richardson Extrapolation and Convergence Rate

Simply speaking, *Richardson Extrapolation* is to construct a better solution based on the estimate of current local error: given the order of accuracy for the solver(say p), we first obtain a solution u_1 with mesh length h,, then we compute another solution u_2 with finer mesh length $h/2$, finnaly we update our solution as $u^\star$ below,

$$u^\star = u_2 + \frac{u_2 - u_1}{2^p - 1}. \quad (14)$$

With Richardson Extrapolation in mind, we can obtain a reliable estimate of the discretization error as a function of the mesh length, or the convergence rate of iterated solution $u_{(h)}$. That is, we want to find p such that the true value $u = u_h + O(h^p)$. Suppose we obtain three

	h=.0125	h=.025	h=.05	h=.1
k=.01	217.2430	36.6320	7.5110	2.1640
k=.005	420.8550	71.0120	13.7690	3.3850
k=.0025	806.7400	140.1110	26.0280	7.0800

Table 1: Running time in 2D case.

solutions, u_1, u_2, u_3, corresponding to three mesh length, h, $h/2$, $h/4$, respectively. Then we estimate p using the formula below,

$$p = \ln \frac{|u_1 - u_2|}{|u_2 - u_3|} / \ln 2. \tag{15}$$

To compute the difference between two vectors of difference sizes, we transform the vector of larger dimension to a temporary one of the same size of the other vector, then find the inf-norm of the difference between the temporary vector and the vector of smaller dimension. See Appendix C(`Rate.m, RateS.m, Diff1.m` for more details.

3 Results

For the 1D problem, we take $h = .1, .05, .025, .0125, .00625$ for x, and $k = .01, .005, .0025$ for t, with $tol = 10^{-8}$ in the iteration stopping criteria. For the 2D problem, we have the same setup except we did not test for the case $h = .00625$, which generates a 25281×25281 matrix in each iteration. For command information, see Appendix B.

3.1 Running time(2D case)

Here we provide the runtime Table 1(in seconds) for the 2D case only, since it does not take time to solve the 1D case with our choices of h and k. Also we provides the log plot of running time curve, see Figure 1. It is clear that the running time increases linearly in k and approximately quadratically in h as h, k decrease.

3.2 Newton Iterations

For both 1D and 2D cases, the mean number of iterations at each time-level is 2, see two examples below. For 2D case, we provide the running time Table[4] 2 for single iteration at each time-level.

[4]The item 'dim' is the dimension of the matrix A in each iteration, given by $(1/h - 1)^2$.

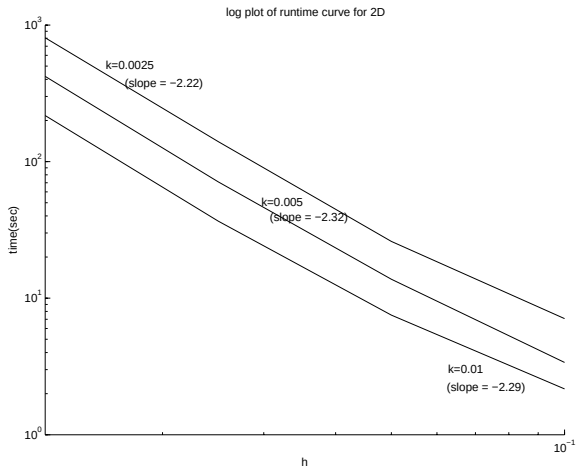

Figure 1: Runing time in 2D case

	h=.0125	h=.025	h=.05	h=.1
t=.1	1.0862	0.1832	0.0376	0.0108
t=.005	1.0521	0.1775	0.0344	0.0085
t=.0025	1.0084	0.1751	0.0325	0.0088
dim	6241	1521	361	81

Table 2: Running time of Newton iteration in 2D case.

```
[u13, i13, t13]=NLP1(0.025,0.01,1e-8);
mean(i13)
ans =
     2

[u22,i22,t22]=NLP2(.05,.005,1e-8);
mean(i22)
ans =
   1.99004975124378
```

3.3 Order of accuracy vs h and k

Here we compare the solutions u at $t = 1$ for different pair (h, k). For each k, we vary the spatial stepsize h and obtain the solutions then compute p_h by eqn (15), i.e., the order of accuracy for h. Similarly, for each h, we vary the temporal stepsize k to obtain p_k. The result is given in Table 3 and 4. Note: we are using the inf-norm for the difference of two vectors.

k	0.01	0.005	0.0025
p_h	1.999836	1.999838	1.999839

h	0.1	0.05	0.025	0.0125	0.00625
p_k	1.999860	1.999882	1.999864	1.999871	1.999854

Table 3: Order of accuracy for h and k in 1D case.

k	0.01	0.005	0.0025
p_h	2.013106	2.013097	2.013095

h	0.1	0.05	0.025	0.0125
p_k	1.998083	1.997975	1.997849	1.997891

Table 4: Order of accuracy for h and k in 2D case.

4 Discussion

It is clear that the order of accuracy for *Crank-Nicolson Method* is $O(h^2+k^2)$(see Results 3.3), which is the advantage of taking $1/2$ in the weighted central difference approximation, i.e., $\left(\partial^2 U/\partial x^2\right)_{i,j+\theta} \simeq \theta\left(\partial^2 U/\partial x^2\right)_{i,j} + (1-\theta)\left(\partial^2 U/\partial x^2\right)_{i,j+1}$. In general, we can apply Richtmyer Method using weighted difference weighted approximation to nonlinear parabolic equation of higher order degree, for example, $u_t = (u^n)_{xx}$.

From the log plot of running time curve for 2D case(see Figure 1), it is clear that the running time increases linearly in k and quadratically in h as k and h decrease. In other words, we need to apply Newton's method to $1/k$ temporal steps, and at each time-level, we need to solve a matrix of size $(1/h-1)\times(1/h-1)$ in each iteration. Unlike the situation to solve 1D case where it takes little effor(time, memory, etc) to obtain the solution, we require much more running time as well as machine memory for 2D case.

Newton's Linearizaion Method is a good tool for linearizing nonlinear PDE's, just like its use for nonlinear equations. The key idea is eqn (8), that is, if we can find and/or approximate $\partial f_i/\partial u_k$ easily(which is the case in our project), we can generate the matrix A and vector b such that the system of linear equations can be written as $Ae = b$. However, we must keep in mind that the matrix and vectors in Newton iterations are variables, and the boundary conditions at different time-level are different. In our projects, it takes 2 Newton iterations to find solution at each time-level with $tol = 1e-8$. In addition, it is usually convenient and efficient to use the 'backslash' method if you use *Matlab*, i.e., $e = A\backslash b$, whereas the matrix A is very sparse.

As to the convergence rate of solutions, the formula 15 based on the *Richardson's Extrapolation* provides fairly good estimates, which inidicates we would probably benefit applying this estimate back to estimate the local error and update to get an improved solution. Besides,

we can estimate the local error and then estimate 'optimal' stepsize to march for the next time-level, which is no longer fixed, —we can implement *adaptive* solver!

Last word is about the generalization of our approach. First consider the heat equations for a stick(1D), a plane(2D), and a cube(3D), we obtain matrix expression of the problem, $Au = b$. In nonlinear parabolic case, we have similar expression but with variable matrix and vectors. To be more precise, there is $3 = 2^1 + 1$ diagonals in the matrix A in 1D case, $5 = 2^2 + 1$ diagonals in 2D case, and (we conjecture and believe) $9 = 2^3 + 1$ diagonals in 3D case. To sum, we can apply our approach to problems of higher order, whereas the matrix size increases dramatically as the dimension increase.

A Examples

Here we present two exampels: one for 1D case and the other for 2D case. For simplicity, we study problems in which the initial and Dirichlet boundary condition correspond to the exact solution given explicity by a linear function, that is, $u(x,t) = x + 2t$ for 1D case and $u(x,y,t) = x + y + 4t$ for 2D case. In each example, we debug the program and give all the intermediate steps at the first time-level, where it takes two Newton iterations to march to the next time-level. From the examples below with moderate values of h and k, we have very high accuracy at each time-level due to the linearity of the exact solution.

A.1 1D case

For the purpose of illustration, we consider a trivial solution to the 1D problem (1), $u(t,x) = x + 2t$, which gives both the initial and boundary conditions. Below we take $h = .1$, $k = .005$, $tol = 10^{-8}$, then study the iterations at the 1st time-level, i.e., $l = 2$ in the main program.

```
% Example for 1D case
% take h=.1, k=.005, tol=1e-8 for Newton iterations,
% initial and boundary conditions given by u=x+2t.
[u, i, t]=NLP1(.1, .005, 1e-8);
[h k p]
ans =
    0.1000    0.0050    2.0000

% initial condition at t=0
v'
ans =
    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000    0.7000    0.8000    0.9000
% boundary conditions at t=0 for x=0 and 1
[u0 u1]
ans =
     0     1

% start iterations with initial value V
v'
ans =
    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000    0.7000    0.8000    0.9000
% the boundary values at 1st time-level, i.e., t=.005
[v0 v1]
```

```
ans =
    0.0100    1.0100

% 1. matrix A generated from vector V , v0 and v1
full(A)
ans =
    4.4000   -0.4000         0         0         0         0         0         0         0
   -0.2000    4.8000   -0.6000         0         0         0         0         0         0
         0   -0.4000    5.2000   -0.8000         0         0         0         0         0
         0         0   -0.6000    5.6000   -1.0000         0         0         0         0
         0         0         0   -0.8000    6.0000   -1.2000         0         0         0
         0         0         0         0   -1.0000    6.4000   -1.4000         0         0
         0         0         0         0         0   -1.2000    6.8000   -1.6000         0
         0         0         0         0         0         0   -1.4000    7.2000   -1.8000
         0         0         0         0         0         0         0   -1.6000    7.6000

% 2. vector b=bu+bv
[bu'; bv'; b']
ans =
    0.4200    0.8200    1.2200    1.6200    2.0200    2.4200    2.8200    3.2200    3.6200
   -0.3799   -0.7800   -1.1800   -1.5800   -1.9800   -2.3800   -2.7800   -3.1800   -3.5599
    0.0401    0.0400    0.0400    0.0400    0.0400    0.0400    0.0400    0.0400    0.0601

% vector e=A\b
e'
ans =
    0.0100    0.0100    0.0100    0.0100    0.0100    0.0100    0.0100    0.0100    0.0100

% Updated solution after 1 iteration vs exact solution
[v'; uu(:,2)']
ans =
    0.1100    0.2100    0.3100    0.4100    0.5100    0.6100    0.7100    0.8100    0.9100
    0.1100    0.2100    0.3100    0.4100    0.5100    0.6100    0.7100    0.8100    0.9100
norm(v-uu(:,2),inf)
ans =
  2.2815e-005

% another iteration
[e'; v'; uu(:,2)']
ans =
   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000
    0.1100    0.2100    0.3100    0.4100    0.5100    0.6100    0.7100    0.8100    0.9100
    0.1100    0.2100    0.3100    0.4100    0.5100    0.6100    0.7100    0.8100    0.9100
norm(e,inf)
ans =
  2.2814e-005
norm(v-uu(:,2),inf)
ans =
  2.2498e-010

% current vector e with updated vector v
```

```
norm(e,inf)
ans =
    2.249804751195629e-010
% that is, we obtain the solution for t=0.005 after 2 iterations
% the local error is very small due to the linearity of the exact solution.
```

A.2 2D case

For the purpose of illustration, we consider a trivial solution to the 2D problem (2) with initial and boundary conditions given by $u(t, x, y) = \sin x \sin y \exp(-2t)$. Below is an example taking $h = .25$, $k = .01$, $tol = 10^{-8}$, at the 1st time-level, i.e., $l = 2$ in the main program.

```
% Example in 2D case
% h=.25, k=.01, tol=1e-8;
% initial and boundary conditions given by u=x+y+4t
[u, iter, time]=NLP2(.25, .01, 1e-8);
[h k p]
ans =
    0.2500    0.0100    6.2500

% at time-level t=0
% initial condition, i.e., u(:,1) or uu(:,2)
u(:,1)'
ans =
  Columns 1 through 8
    0.5000    0.7500    1.0000    0.7500    1.0000    1.2500    1.0000    1.2500
  Column 9
    1.5000
u(:,1)'
ans =
    0.5000    0.7500    1.0000    0.7500    1.0000    1.2500    1.0000    1.2500    1.5000
% boundary condition, i.e., if x=0 or 1, or y=0 or 1
[u_0_dot; u_dot_0; u_1_dot; u_dot_1]
ans =
    0.2500    0.5000    0.7500
    0.2500    0.5000    0.7500
    1.2500    1.5000    1.7500
    1.2500    1.5000    1.7500

% Start iterations at time-level t=.005, or l=2 in the loop
% 1. initial value for v and the exact solution at t=.005
[v'; uu(:,2)']
ans =
    0.5000    0.7500    1.0000    0.7500    1.0000    1.2500    1.0000    1.2500    1.5000
    0.5400    0.7900    1.0400    0.7900    1.0400    1.2900    1.0400    1.2900    1.5400

% boundary condition at t=.005
[v_0_dot; v_dot_0; v_1_dot; v_dot_1]
ans =
    0.2900    0.5400    0.7900
    0.2900    0.5400    0.7900
```

```
    1.2900    1.5400    1.7900
    1.2900    1.5400    1.7900

% 2. matrix A generated from vector v and its boundary values
full(A)
ans =
   16.5000   -1.5000         0   -1.5000         0         0         0         0         0
   -1.0000   18.5000   -2.0000         0   -2.0000         0         0         0         0
         0   -1.5000   20.5000         0         0   -2.5000         0         0         0
   -1.0000         0         0   18.5000   -2.0000         0   -2.0000         0         0
         0   -1.5000         0   -1.5000   20.5000   -2.5000         0   -2.5000         0
         0         0   -2.0000         0   -2.0000   22.5000         0         0   -3.0000
         0         0         0   -1.5000         0         0   20.5000   -2.5000         0
         0         0         0         0   -2.0000         0   -2.0000   22.5000   -3.0000
         0         0         0         0         0   -2.5000         0   -2.5000   24.5000

% 3. vector b=bu+bv
[bu'; bv'; b']
ans =
    6.5000    9.6250   12.7500    9.6250   12.7500   15.8750   12.7500   15.8750   19.0000
   -5.9568   -9.0834  -12.0868   -9.0834  -12.2500  -15.2534  -12.0868  -15.2534  -18.2168
    0.5432    0.5416    0.6632    0.5416    0.5000    0.6216    0.6632    0.6216    0.7832

% 4. solve e=A\b and update vector v
[e'; v']
ans =
    0.0402    0.0401    0.0402    0.0401    0.0400    0.0401    0.0402    0.0401    0.0402
    0.5402    0.7901    1.0402    0.7901    1.0400    1.2901    1.0402    1.2901    1.5402
% difference between current v and exact solution after 1 iteration
norm(v-uu(:,2),inf)
ans =
  2.1619e-004

% 5. solve e=A\b and update vector v(in the 2nd iteration)
[e'; v']
ans =
   -0.0002   -0.0001   -0.0002   -0.0001   -0.0000   -0.0001   -0.0002   -0.0001   -0.0002
    0.5400    0.7900    1.0400    0.7900    1.0400    1.2900    1.0400    1.2900    1.5400
% difference between current v and exact solution after 2 iterations
norm(v-uu(:,2),inf)
ans =
  9.2801e-009

% 6. solve e=A\b with currecnt vector v
e'
ans =
  1.0e-008 *
   -0.9280    0.0283   -0.4829    0.0283    0.2214   -0.0282   -0.4829   -0.0282   -0.2848
norm(e,inf)
ans =
  9.2801e-009
```

```
% since norm(e,inf)<1e-8, we stop iterations and march to the next time-level.
```

B Commands

Here we list the commands to obtain solutions, to get time table, and to find the convergence rate.

1. Obtain solutions.

```
% 1D problem
% for t = 0.01, 0.005, 0.0025
% for h = 0.1, 0.05, 0.025, 0.0125, 0.00625
[u11, i11, t11]=NLP1(0.1,0.01,1e-8);
[u12, i12, t12]=NLP1(0.05,0.01,1e-8);
...
[u35, i35, t35]=NLP1(0.00625, 0.0025,1e-8);
% note: u --solution matrix
%       i --iteration vector
%   t --running time
```

```
% 2D problem
% for t = 0.01, 0.005, 0.0025
% for h = 0.1, 0.05, 0.025, 0.0125
[u11, i11, t11]=NLP2(0.1,0.01,1e-8);
[u12, i12, t12]=NLP2(0.05,0.01,1e-8);
...
[u34, i34, t34]=NLP2(0.0125, 0.0025,1e-8);
% note: same notations for u, i, and t.
```

2. Construct time table for the 2D case.

```
% 2D case
tol=1e-8;
k=[0.01 0.005 0.0025];
h=[0.0125 0.025 0.05 0.1];
t=[t14 t13 t12 t11; t24 t23 t22 t21; t34 t33 t32 t31]
t =
  217.2430   36.6320    7.5110    2.1640
  420.8550   71.0120   13.7690    3.3850
  806.7400  140.1110   26.0280    7.0800
```

```
% mean time for each iteration
[t(1,:)/100; t(2,:)/200; t(3,:)/400]/2
ans =
    1.0862    0.1832    0.0376    0.0108
    1.0521    0.1775    0.0344    0.0085
```

```
    1.0084    0.1751    0.0325    0.0088

% note: there are 100, 200 and 400 time steps for t=0.01, 0.05, and t=0.025,
%       respectively; moreover, there are about 2 iterations at each time-level.
```

3. Estimatte the convergence rate.

```
% 1D case
% find p for t=0.01
p13=Rate(u13,u14,u15,101)
p13 =
   1.50115859265552   1.99983691089679
...
% find p for t=0.025
p33=Rate(u33,u34,u35,401)
p33 =
   1.50030326932689   1.99983943506904
% note: the first one is in 2-norm and the second one is in inf-norm.

% find p for h=0.1
p=RateS(u11,u21,u31)
p =
   1.99937218001895   1.99986037178239
...
% find p for h=0.00625
p=RateS(u15,u25,u35)
p =
   1.99925594341070   1.99985402476488


% 2D case
...
% find p for t=0.0025
p=log(diff1(u32,u33,401)/diff1(u33,u34,401))/log(2) %inf-norm, k=0.005
p =
   2.01309523099357

% find p for h=0.1
p=log(norm(u11(:,101)-u21(:,201),inf)/norm(u21(:,201)-u31(:,401),inf))/log(2)
p =
   1.99808395859832
...
% find p for h=0.0125
p=log(norm(u14(:,101)-u24(:,201),inf)/norm(u24(:,201)-u34(:,401),inf))/log(2)
p =
   1.99789112962261
```

C Codes

Here are the m-files: `NR1.m, NR2.m, NLP1.m, NLP2.m, Rate.m, RateS.m`, and `Diff1.m`.

NR1.m

M-file `NR1.m` for the 1D problem.

```
function [u,iter,time]=NR1(t,x);
% u = NR1(t, x) with tol=1E-8
% Newton-Ralson method to solve f(u,t,x)=0 given t and x,
% where f(u,t,x) is given implicitly below
%  (2u-3)+log(u-1/2)-2(2t-x)=0.
% The initial guess for u is taken as (2t-x+3)/2, i.e., we
% igore the logarithm part.

tol=1e-8;   % to set the error bound
tic;    % to start time counter

u=(2*t-x+3);     % u --value obtained before iteration
f=(2*u-3)+log(u-1/2)-2*(2*t-x); % f --is f(u,t,x)
fp=2+1/(u-1/2);     % fp --is f'(u,t,x)
uu=u-f/fp;      % uu --value after iteration
iter=1;

while (abs(u-uu) > tol)
    % to start another iteration
    iter=iter+1;
    u=uu;
    f=(2*u-3)+log(u-1/2)-2*(2*t-x); % f --is f(u,t,x)
    fp=2+1/(u-1/2);     % fp --is f'(u,t,x)
    uu=u-f/fp;      % uu --value after iteration
end

time=toc;   % to obtain the elasped time

% u=x+2*t;      % uncomment this one for testing ONLY

%%%%%%%%%%%%%%%%%%%%%%%%%%end of mfile %%%%%%%%%%%%%%%%%%%%%%%%%%
```

NR2.m

M-file `NR2.m` for the 2D problem.

```
function u=NR2(t, x, y);
% to compute u(t, x, y) for t from [0,1] and
% (x,y) from [0,1]^2.
```

```
% Here the function is choose as 20sin(x)sin(y)exp(-2t),

u=sin(x)*sin(y)*exp(-2*t);
%u=x+y+4*t;   % uncomment this one for testing ONLY

%%%%%%%%%%%%%%%%%%%%%%%%%end of mfile %%%%%%%%%%%%%%%%%%%%%%%%%
```

C.1 1D Solver: NLP1()

M-file `NLP1.m` for solving 1D nonlinear parabolic equation of the form (1).

```
function [u, iter, time]=NLP1(h, k, tol);
% [u, iter, time] = NLP1(h, k, tol)
% Solver for Non-Linear Parabolic equation, du/dt=dd(u^2)/dd(x)
% (dd() is the 2nd differential operator).
%
% We consider the initial conditions and Dirichlet boundary
% conditions corresponding to the eaxct solution obtained by
% NR1(t,x)(see NR1.m for the implicit function f(u,t,x)). Here
% choose fixed stepsize for x and t, i.e., h=Delta(x) and
% k=Delta(t). Moreover, we have x from [0,1] and t from[0, 1].


tic;    % to start time counter

% to initialize...
ns=1/h-1;   % the number of unkonwn points for each row(spatial)
x=(h:h:1-h)';  % spatial vector for later reference

nt=1/k;     % the number of time points to be solved
t=(0:k:1)';  % temporal vector for later reference

% u=zeros([ns 2]);    % for test purpose only
u=zeros([ns nt+1]);     % u(:,t) gives solution at time t.
p=h*h/k;
iter=zeros([nt 1]);


% to get initial and Dirichlet boundary conditions
for i=1:ns
    u(i,1)=NR1(0,x(i));
end

% temprory vector
b=zeros([ns 1]);
v=zeros([ns 1]);

% % now to compute the exact solution for u
% uu=zeros([ns nt+1]);
% uu(:,1)=u(:,1);     % case: t=0;
```

```
% for i=1:ns
%     for j=1:nt
%         uu(i,j+1)=NR1(t(j+1),x(i));
%     end
% end


% main loop to obtain u(t_(j+1)) from u(t_j)
for j=2:nt+1
    % to solve e_(j) s.t. u(t_j)=u(t_(j-1))+e_j
    % the equation is A_j*e_j=b_j
    % Both A and b are generated from vector v(t_(j-1)),
    % which is initially u(t_(j-1))
    v=u(:,j-1);
    A=spdiags([-2*v 4*v+2*p -2*v], [-1 0 1], ns, ns);
    % to use the action of matrix
    v2=v.^2;
    u0=NR1(t(j-1),0);   % i.e., u(, 0)
    u1=NR1(t(j-1),1);   % i.e., u(t_j, 1)
    v0=NR1(t(j),0);   % i.e., u(t_(j+1),0), left bound for vector v
    v1=NR1(t(j),1);   % i.e., u(t__(j+1),0), right bound for vector v
    bu=[u0^2; v2(1:ns-1)] - 2*(v2-p*v) + [v2(2:ns); u1^2];  % unchanged, here v=u(:,j)
    bv=[v0^2; v2(1:ns-1)] - 2*(v2+p*v) + [v2(2:ns); v1^2];
    b=bu+bv;
    e=A\b;

    counter=1;
    while (norm(e,inf)>tol)
        v=v+e;
        A=spdiags([-2*v 4*v+2*p -2*v], [-1 0 1], ns, ns);
        v2=v.^2;
        bv=[v0^2; v2(1:ns-1)] - 2*(v2+p*v) + [v2(2:ns); v1^2];
        b=bu+bv;
        e=A\b;

        counter=counter+1;  % to record iteration number
    end

    u(:,j)=v;
    iter(j-1)=counter;
end

time=toc;

%%%%%%%%%%%%%%%%%%%%%%%%%%%end of mfile%%%%%%%%%%%%%%%%%%%%%%%%%%
```

C.2 2D Solver: NLP2()

M-file `NLP2.m` for solving the 2D nonlinear parabolic equation of the form (2).

```
function [u, iter, time]=NLP2(h, k, tol);
% [u, t, iter, time, uu] = NLP1(h, k, tol)
% Solver for 2D Non-Linear Parabolic equation, du/dt=dd(u^2)/dd(x)
% (dd() is the 2nd differential operator).
%
% We consider the initial conditions and Dirichlet boundary
% conditions corresponding to the eaxct solution obtained by
% NR2(t,x,y)(see NR2.m for the implicit function f(u,t,x,y)). Here
% choose fixed stepsize for x,y, and t, i.e., h=Delta(x)=Delta(y) and
% k=Delta(t). Moreover, we have (x,y) from [0,1]^2 and t from[0, 1].


tic;    % to start timer

% to initialize...
ns=1/h-1;   % the number of unkonwn points for each row(spatial)
            % and ns*ns is the size of solution vector for each time t

nt=1/k;     % the number of time points to be solved
t=(0:k:1)';  % temporal vector for later reference

% u=zeros([ns*ns 2]);  % for test purpose
u=zeros([ns*ns nt+1]);     % u(:,t) gives solution at time t.

p=h*h/k;
iter=zeros([nt 1]);

% to get initial conditions
for i=1:ns
    for j=1:ns
        l=(i-1)*ns+j;   % index in solution vector for [i,j]
        u(l,1)=NR2(0, i*h, j*h);
    end
end

% fixed boundary conditions for [0,:], [:,0], [1, :], and [:, 1]
for i=1:ns
    v_0_dot(i)=NR2(0, 0, i*h);  % [0, :]
    v_dot_0(i)=NR2(0, i*h, 0);  % [:, 0]
    v_1_dot(i)=NR2(0, 1, i*h);  % [1, :]
    v_dot_1(i)=NR2(0, i*h, 1);  % [:, 1]
end

% % to compute the exact solution
% uu=zeros(ns*ns, 2);
% uu(:,1)=u(:,1);
%     for i=1:ns
%         for j=1:ns
%             ll=(i-1)*ns+j;
%             uu(ll,l)=NR2(t(l),i*h,j*h);
%         end
```

```
%     end
% end

% temporary vectors
bu=zeros(ns*ns, 1);
bv=zeros(ns*ns, 1);

% Main Loop to compute U(t_l) from U(t_(l-1))
n=ns*ns;    % the size of solution vector
for l=2:nt+1
    round=l;

    % to solve e_(l) s.t. u(t_l)=v_l+e_l
    % the equation is A_l*e_l=b_l
    % Both A and b are generated from vector v_l,
    % which is initially u(t_(l-1))

    v=u(:,l-1);

    % to generate the sparse pentadiagonal matrix A_l
    v1_plus=v;  v1_minus=v;
    for i=1:ns-1
        v1_plus(i*ns+1)=0;
        v1_minus(i*ns)=0;
    end

    A=spdiags([-2*v  -2*v1_minus  8*v+2*p  -2*v1_plus  -2*v ], [-ns -1 0 1 ns], ns*ns, ns*ns);

    % first to generate bu which is invariable in the current loop.
    % note: v=u(:,l-1) right now
    u_0_dot=v_0_dot;
    u_dot_0=v_dot_0;
    u_1_dot=v_1_dot;
    u_dot_1=v_dot_1;

    for i=1:ns
        for j=1:ns
            ll=(i-1)*ns+j;   % index for [i,j] in vector bu
            tmp=-4*v(ll)^2+2*p*v(ll);     % u(t, i,j)
            if (i==1)
                tmp=tmp + u_0_dot(j)^2; % u(t, 0,j)
            else
                tmp=tmp + v(ll-ns)^2;    % u(t, i-1,j)
            end

            if (j==1)
                tmp=tmp + u_dot_0(i)^2; % u(t, i,0)
            else
                tmp=tmp + v(ll-1)^2;     % u(t, i, j-1)
            end
```

```
        if (i==ns)
            tmp=tmp + u_1_dot(j)^2; % u(t, 1,j)
        else
            tmp=tmp + v(ll+ns)^2;    % u(t, i+1,j)
        end

        if (j==ns)
            tmp=tmp + u_dot_1(i)^2; % u(t, i,1)
        else
            tmp=tmp + v(ll+1)^2;     % u(t, i,j+1)
        end

        bu(ll)=tmp;
    end
end

% then to generate bv in the similar way above
% first to generate boundary condition for v

tj=t(l);
for i=1:ns
    v_0_dot(i)=NR2(tj, 0, i*h);  % [0, :]
    v_dot_0(i)=NR2(tj, i*h, 0);  % [:, 0]
    v_1_dot(i)=NR2(tj, 1, i*h);  % [1, :]
    v_dot_1(i)=NR2(tj, i*h, 1);  % [:, 1]
end

for i=1:ns
    for j=1:ns
        ll=(i-1)*ns+j;   % index for [i,j] in vector bu
        tmp=-4*v(ll)^2 - 2*p*v(ll);     % v(t, i,j)
        if (i==1)
            tmp=tmp + v_0_dot(j)^2; % u(t, 0,j)
        else
            tmp=tmp + v(ll-ns)^2;    % v(t, i-1,j)
        end

        if (j==1)
            tmp=tmp + v_dot_0(i)^2; % u(t, i,0)
        else
            tmp=tmp + v(ll-1)^2;     % v(t, i, j-1)
        end

        if (i==ns)
            tmp=tmp + v_1_dot(j)^2; % u(t, 1,j)
        else
            tmp=tmp + v(ll+ns)^2;    % v(t, i+1,j)
        end

        if (j==ns)
            tmp=tmp + v_dot_1(i)^2; % u(t, i,1)
```

```
        else
            tmp=tmp + v(ll+1)^2;     % v(t, i,j+1)
        end

        bv(ll)=tmp;
    end
end

% finally to get b=bu+bv
b=bu+bv;

% to solve e_l from equation A_l*e_l=b
e=A\b;
counter=0;

while (norm(e,inf) > tol)
    % to continue iteration, i.e., to generate A and bv using updated v
    v=v+e;

    % to generate the sparse pentadiagonal matrix A_l
    v1_plus=v;  v1_minus=v;
    for i=1:ns-1
        v1_plus(i*ns+1)=0;
        v1_minus(i*ns)=0;
    end

    A=spdiags([-2*v  -2*v1_minus  8*v+2*p  -2*v1_plus  -2*v ], [-ns -1 0 1 ns], ns*ns, ns*ns);

    % to generate bv in the similar way above
    for i=1:ns
        for j=1:ns
            ll=(i-1)*ns+j;   % index for [i,j] in vector bu
            tmp=-4*v(ll)^2 - 2*p*v(ll);     % v(t, i,j)
            if (i==1)
                tmp=tmp + v_0_dot(j)^2; % u(t, 0,j)
            else
                tmp=tmp + v(ll-ns)^2;    % v(t, i-1,j)
            end

            if (j==1)
                tmp=tmp + v_dot_0(i)^2; % u(t, i,0)
            else
                tmp=tmp + v(ll-1)^2;     % v(t, i, j-1)
            end

            if (i==ns)
                tmp=tmp + v_1_dot(j)^2; % u(t, 1,j)
            else
                tmp=tmp + v(ll+ns)^2;    % v(t, i+1,j)
            end
```

```
                if (j==ns)
                    tmp=tmp + v_dot_1(i)^2; % u(t, i,1)
                else
                    tmp=tmp + v(ll+1)^2;     % v(t, i,j+1)
                end

                bv(ll)=tmp;
            end
        end

        % to get b=bv+bu
        b=bv+bu;

        % to solve for A*e=b
        e=A\b;
        counter=counter+1;
    end % end of while()

    u(:,l)=v;
    iter(l)=counter;

end

time=toc;
%%%%%%%%%%%%%%%%%%%%%%%%end of mfile%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

C.3 Vector Comparisons

1. M-file `Rate.m` to find the inf-norm difference between two solution vectors of different sizes in 1D case.

```
function p=Rate(u1,u2,u3, nt);
% find the convergence rate from three solutions
% u1, u2, u3 with stepsize h, h/2, and h/4, respectively.

n=size(u1);
n1=n(1);    % to get the size of vector u1
n2=2*n1+1;    n3=2*n2+1;

p=zeros(1,2);
% using 2-norm
p(1)=log(norm(u1(:,nt)-u2(2:2:n2-1,nt))/norm(u2(:,nt)-u3(2:2:n3-1,nt)))/log(2);
% using inf-norm
p(2)=log(norm(u1(:,nt)-u2(2:2:n2-1,nt),inf)/norm(u2(:,nt)-u3(2:2:n3-1,nt),inf))/log(2);

%%%%%%%%%%%%%%%%%%%%%%%%end of mfile%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

2. M-file `RateS.m` to find the inf-norm difference between two solution vectors of same sizes but different time steps in 1D case.

```
function p=RateS(u1,u2,u3)
% vectors u1, u2, u3 are of same size

s=size(u1);
nt=s(2);    % number of steps in time
n1=nt;  n2=2*n1-1;  n3=2*n2-1;

p=zeros(1,2);
% using 2-norm
p(1)=log(norm(u1(:,n1)-u2(:,n2))/norm(u2(:,n2)-u3(:,n3)))/log(2);
% using inf-norm
p(2)=log(norm(u1(:,n1)-u2(:,n2),inf)/norm(u2(:,n2)-u3(:,n3),inf))/log(2);

%%%%%%%%%%%%%%%%%%%%%%%%%end of mfile%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

3. M-file `Diff1.m` to find the inf-norm difference between two solution vectors of different sizes in 2D case.

```
function d=Diff1(u1, u2,n);
% to find the inf-norm of u1-u2 where u2 is a finer version of u1
% the idea is to construct a vector from u2 with same size as u1
% the transformation of (i,j) in u1 is (2i,2j) in u2

s=size(u1);
s1=sqrt(s(1));
s=size(u2);
s2=sqrt(s(1));

tmp=zeros(s1^2,1);
% to construct a vector from u2
for i=1:s1
    for j=1:s1
        l1=(i-1)*s1+j;
        l2=(2*i-1)*s2+2*j;
        tmp(l1)=u2(l2,n);
    end
end

d=norm(u1(:,n)-tmp,inf);

%%%%%%%%%%%%%%%%%%%%%%%%%end of mfile%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

References

[1] G.D. Smith, *Numerical solution of partial differential equations: Finite Difference Methods*, 3rd ed, Oxford, 1985.