PEKING UNIVERSITY, MATHEMATICS

Introduction to Applied Mathematics

1700010780

July 31, 2019

1 INTRODUCTION TO FINITE ELEMENT METHOD

1.1 Basic concepts of finite element method

1.1.1 a

Proof. Since u is twice continuous differentiable, by integration by parts, we can get:

$$-\int_0^L \frac{d^2u}{dx^2} v dx = \int_0^L f v dx$$

Assume $w = f + \frac{d^2u}{dx^2}$, then w is continuous and orthogonal to space V_0 in L^2 norm. Since any continuous function w on $[0, L]$ can be uniformly approximated by polynomials w_n. We need to adjust the end points of w_n to zero, so we introduce smooth function $\mu_n(x)$

$$\mu_n(x) = \begin{cases} 1 & \frac{1}{n} \le x \le L - \frac{1}{n} \\ 0 & x = 0 \text{ or } L \end{cases}$$

Then $v_n(x) := w_n(x)\mu_n(x) \in V_0$ still converge to $w(x)$ uniformly. Since $\forall \epsilon > 0$, pick n large enough such that $|u(x)| < \frac{\epsilon}{2} \forall x \in [0, \frac{1}{n}] \cup [L - \frac{1}{n}, L]$, and $|u_n(x) - u(x)| < \frac{\epsilon}{2} \forall x \in [0, L]$, then $|v_n(x) - u(x)| < \epsilon \forall x \in [0, L]$. Thus, by the first equation, let n tend to infinity, we can get $\int_0^L u^2 dx = 0$, which means $u = 0 \forall x \in [0, L]$. □

1.1.2 b

Proof. Since

$$\int_0^L \frac{du_h}{dx} \frac{dv}{dx} dx = \int_0^L f v dx \forall v \in V_{h,0}$$

and by the definition of u

$$\int_0^L \frac{du}{dx} \frac{dv}{dx} dx = \int_0^L f v dx \forall v \in V_{h,0}$$

By subtraction at the same time, the result follows. □

1.1.3 c

The variational form of Problem (0.1) is

$$\int_0^1 \frac{du}{dx}\frac{dv}{dx}dx = \int_0^L v dx, \forall v \in V_0$$

Discretize it we get that

$$A = \begin{bmatrix} \frac{1}{h_1}+\frac{1}{h_2} & -\frac{1}{h_2} & & \\ -\frac{1}{h_2} & \frac{1}{h_2}+\frac{1}{h_3} & -\frac{1}{h_3} & \\ & \ddots & \ddots & \ddots \\ & & -\frac{1}{h_{n-1}} & \frac{1}{h_{n-1}}+\frac{1}{h_n} \end{bmatrix}, \quad b = \begin{bmatrix} \frac{h_1+h_2}{2} \\ \frac{h_2+h_3}{2} \\ \vdots \\ \frac{h_{n-1}+h_n}{2} \end{bmatrix}$$

When $n = 5$, we can get

$$A = \begin{bmatrix} \frac{1}{h_1}+\frac{1}{h_2} & -\frac{1}{h_2} & & \\ -\frac{1}{h_2} & \frac{1}{h_2}+\frac{1}{h_3} & -\frac{1}{h_3} & \\ & -\frac{1}{h_3} & \frac{1}{h_3}+\frac{1}{h_4} & -\frac{1}{h_4} \\ & & -\frac{1}{h_4} & \frac{1}{h_4}+\frac{1}{h_5} \end{bmatrix}$$

The programme is in the appendix, "finite_element.m" calculates the value of $x_i = i/n$ $0 \le i \le n$, denoted by length n column vector, and "err_estimte.m" calculates the error in part c.
When $n = 10, 20, 40, 80$, we can plot the picture of solutions by the following code:

```
for n = [10,20,40,80]
X = 0:1/n:1;
y = finite_element(@(x) 1, X);
plot(X,y);
hold on
end
legend('10','20','40','80')
```

The result is plotted in Figure 1.1:
For clarity, we will put the vector diagram as well.
When $n = 2, 4, 8, 16, 32, 64, 128$, we use $||\frac{d(u-u_h)}{dx}||_{L^2(\Omega)}$ as error. Since the true solution is know, we can calculate the error explicitly without any numerical integration formula, i.e. $||\frac{d(u-u_h)}{dx}||_{L^2(\Omega)} = (\sum_{i=1}^{n} \int_{\frac{i-1}{n}}^{\frac{i}{n}} (\frac{1-2x}{2} - (u_{i+1}-u_i))^2)^{1/2}$. The code is below:

```
function err = err_estimate(m)
X = 0:1/m:1;
mu = finite_element(@(x) 1, X);
err = 1/12;
for i = 1:m
    err = err + (mu(i+1)-mu(i))^2*m - (mu(i+1)-mu(i))*(1-(2*i-1)/m);
end
err = sqrt(err);
end
```

The result is presented as follows:

n	2	4	8	16	32	64
Error	0.1443	0.0722	0.0361	0.0180	0.0090	0.0045

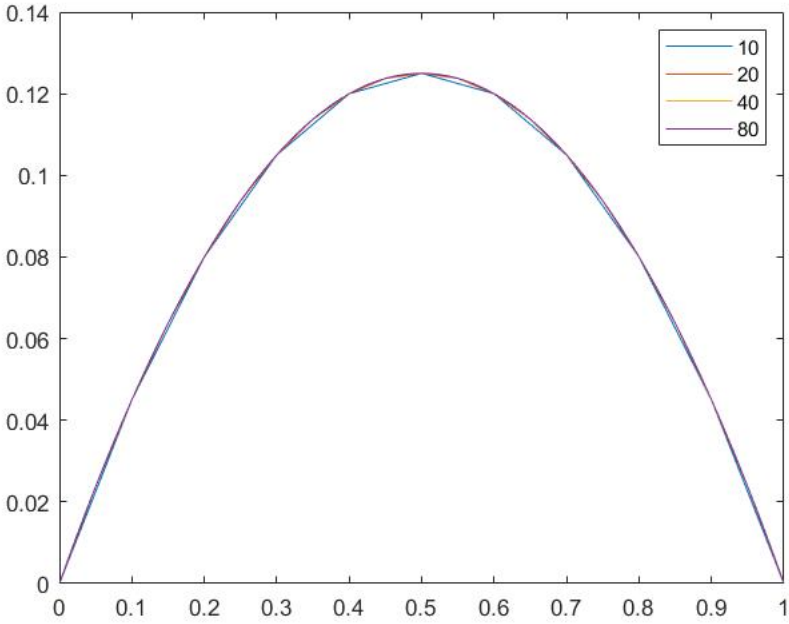

Figure 1.1: Finite elements solution.

1.2 Numerical methods and self-adaptation

1.2.1 Step 1

```
X = 0:1/10:1;
y = finite_element(@(x) exp(-100*(x-0.5)^2), X);
plot(X,y);
```

See Figure 1.2:

1.2.2 Step 2

The function takes function f and the grid X as inputs, and output is a vector defined in the problem.

```
function res = integration(f,X)
n = length(X)-1;
for i = 1:n
    res(i) = (X(i+1)-X(i))*sqrt((f(X(i+1))^2+f(X(i))^2)*(X(i+1)-X(i))/2);
end
end
```

1.2.3 Step 3

The code of self-adaptive method is below:

```
function new_X = self_adaption(f, X, alpha)
n = length(X)-1;
res = integration(f,X);
mx = max(res);
```

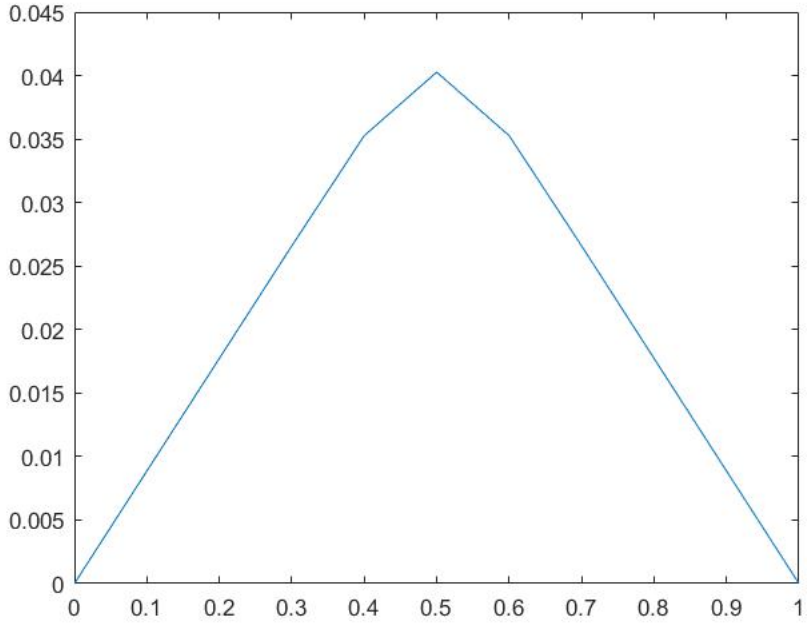

Figure 1.2: Finite elements solution.

```
new_point = [];
for i = 1:n
    if res(i) > alpha*mx
        new_point = [new_point, (X(i)+X(i+1))/2];
    end
end
new_X = sort([X,new_point]);
```

Code for threshold $n < 20$:

```
X = 0:1/10:1;
while length(X)<20
X = self_adaption(@(x) exp(-100*(x-0.5)^2), X, 0.5);
end
y = finite_element(@(x) exp(-100*(x-0.5)^2),X);
scatter(X,y)
hold on
legend('threshold=20')
hold off
```

The result is in Figure 1.3.

1.2.4 Step 4

In the Figure 1.4:

1.2.5 Step 5

In the Figure 1.5, for clarity, we still append "e5.fig" in the file:

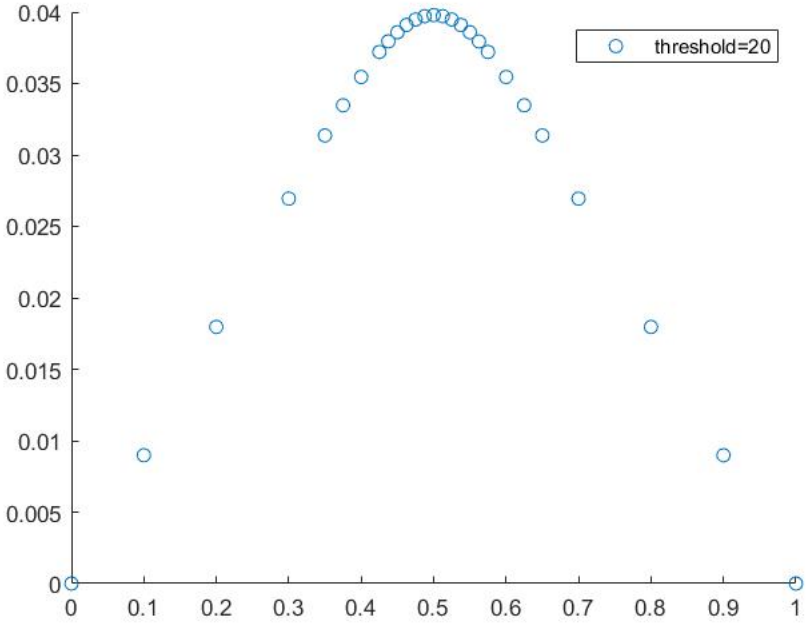

Figure 1.3: Threshold = 20

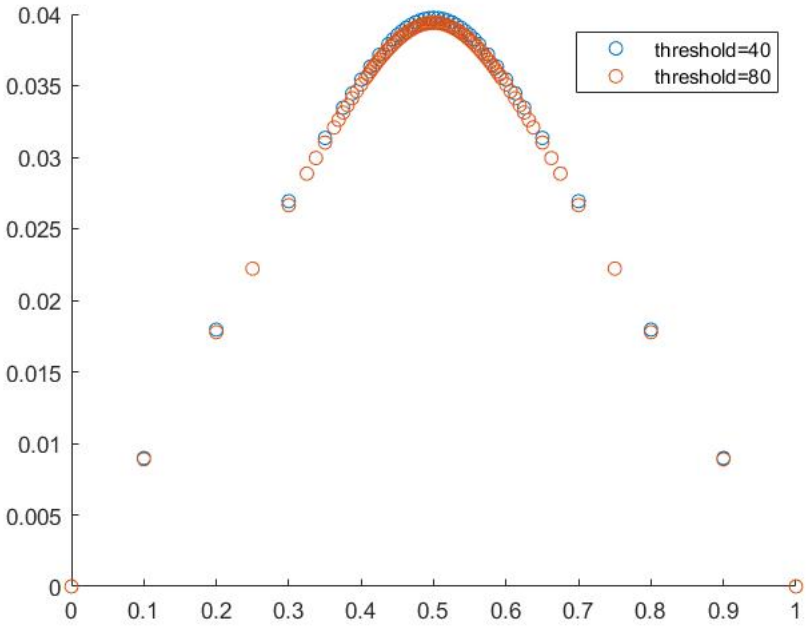

Figure 1.4: Threshold = 40,80

1.2.6 Comparison with $f = 1$

We use $n < 40$ as threshold, and $\alpha = 0.5$, then we get Figure 1.6:
Compare these four figures, it is obvious that in the former case, new sample points are concentrated in the middle, where f attains its maximum, while in the latter case, new sample points distributed rather evenly. Thus, self-adaptive method can sample more points where the values of f concentrate, which gives us more precise description of regions where the absolute value of the twice derivative of u is large. Furthermore, this method requires no prior knowledge of f, and will automatically find region needs attention, i.e. f is comparatively large.

1.3 Self-adaptation

Assume v is a smooth function, by change of variables, we can get the variational form:

$$\int_0^1 f v dx = -\int_0^1 \frac{d^2u}{dx^2} v dx = -v\frac{du}{dx}|_0^1 + \int_0^1 \frac{du}{dx}\frac{dv}{dx}dx = \int_0^1 \frac{du}{dx}\frac{dv}{dx}dx + \kappa_0 u(0)v(0)$$

Then we should discretize u as

$$u_h(x) = \sum_{i=0}^{n} \mu_i \phi_i(x)$$

where $\phi_0(x) = max(0, \frac{x_1 - x}{x_1}), x_0 \le x \le x_1$ and $\phi_n(x) = max(0, \frac{x - x_n}{x_n}), x_{n-1} \le x \le x_n$ and the remains defined as usual.
Hence, we can get:

$$\int_0^1 f\phi_i dx = \sum_{j=0}^{n} \mu_j \int_0^1 \frac{d\phi_j}{dx}\frac{d\phi_i}{dx}dx + \kappa_0\mu_0\delta_{0,i}, \text{ for } i = 0, 1, 2, \ldots, n.$$

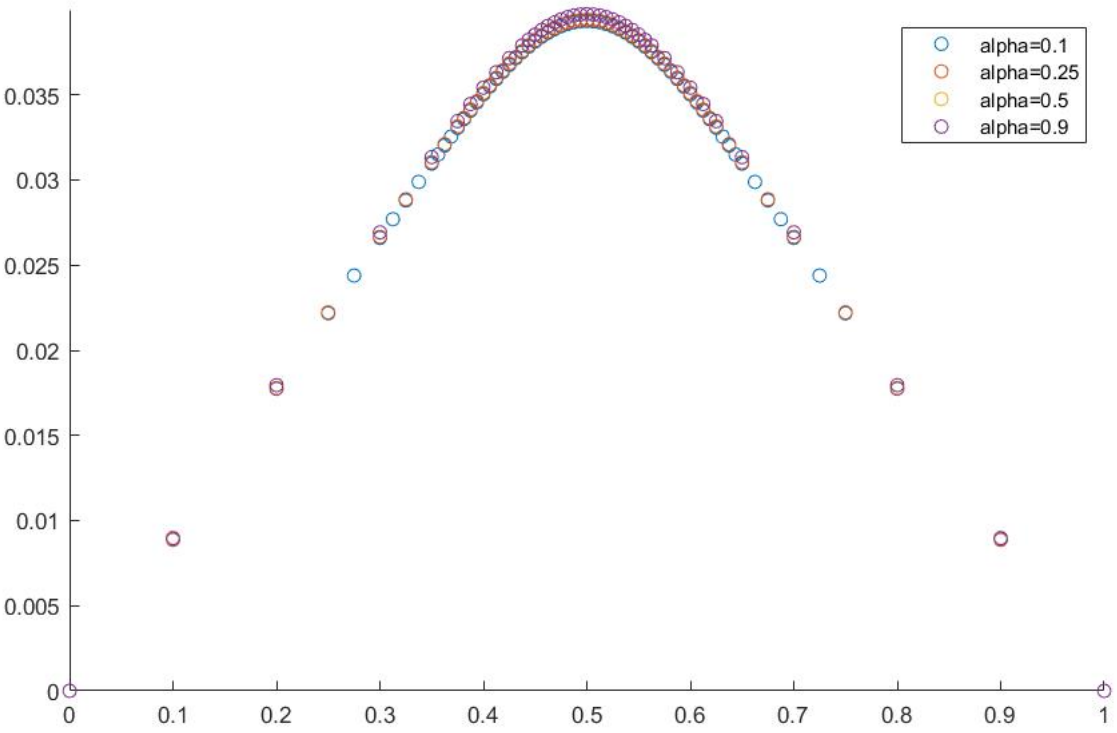

Figure 1.5: $\alpha = 0.1, 0.25, 0.5, 0.9$

If we adopt the same mark, then $A_{00} = \frac{1}{h_1}, A_{01} = -\frac{1}{h_1}, A_{nn} = \frac{1}{h_n}, An-1, n = -\frac{1}{h_n}$. Thus, if we stipulate $\vec{\mu} = (\mu_0, \mu_1, \ldots, \mu_n)^T$, we will get $A\vec{\mu} = b$ where

$$A = \begin{bmatrix} \frac{1}{h_1} + \kappa_0 & -\frac{1}{h_1} & & & & & \\ -\frac{1}{h_1} & \frac{1}{h_1} + \frac{1}{h_2} & -\frac{1}{h_2} & & & & \\ & -\frac{1}{h_2} & \frac{1}{h_2} + \frac{1}{h_3} & -\frac{1}{h_3} & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & -\frac{1}{h_{n-1}} & \frac{1}{h_{n-1}} + \frac{1}{h_n} & -\frac{1}{h_n} & \\ & & & & -\frac{1}{h_n} & \frac{1}{h_n} & \end{bmatrix}, \quad b = \begin{bmatrix} f(x_0)\frac{h_1}{2} \\ f(x_1)\frac{h_1+h_2}{2} \\ f(x_2)\frac{h_2+h_3}{2} \\ \vdots \\ f(x_{n-1})\frac{h_{n-1}+h_n}{2} \\ f(x_n)\frac{h_n}{2} \end{bmatrix}$$

We modify "finite_elements.m" to "problem1_3.m" to coincide with the new situation, whose input is the threshold for self adaptive method. The code is in the Appendix, problem 1, and the result is in the Figure 1.7.

Notice that here we only set threshold to 64, since the case of lower threshold will be clear in this graph. To visualize the mesh better, we present another graph by function "scatter" in the Figure 1.8.

It is clear from Figure 1.8 that the adaptive method can force the gird point to concentrate on where solution u have large second order derivatives, and will use relatively fewer points to describe where the solution is close to linear functions. Hence, it will use less gird size to accurately demonstrate the solution compared to even grids.

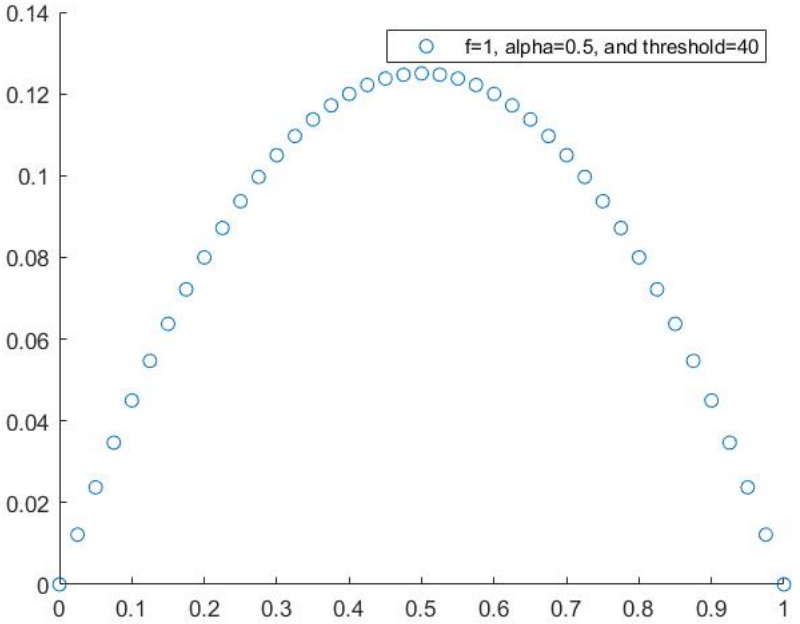

Figure 1.6: Self-adaptation when $f = 1$

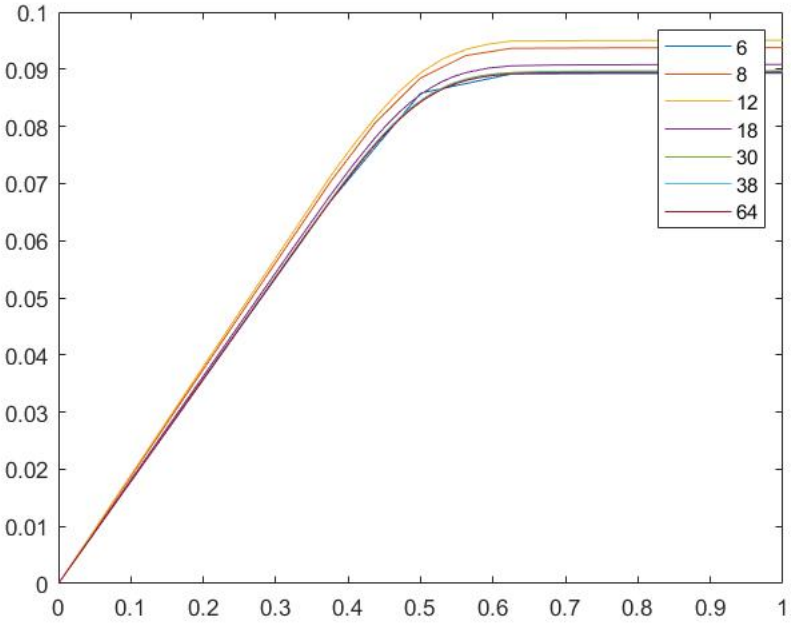

Figure 1.7: Solution by the adaptive method. $n \leq 64$

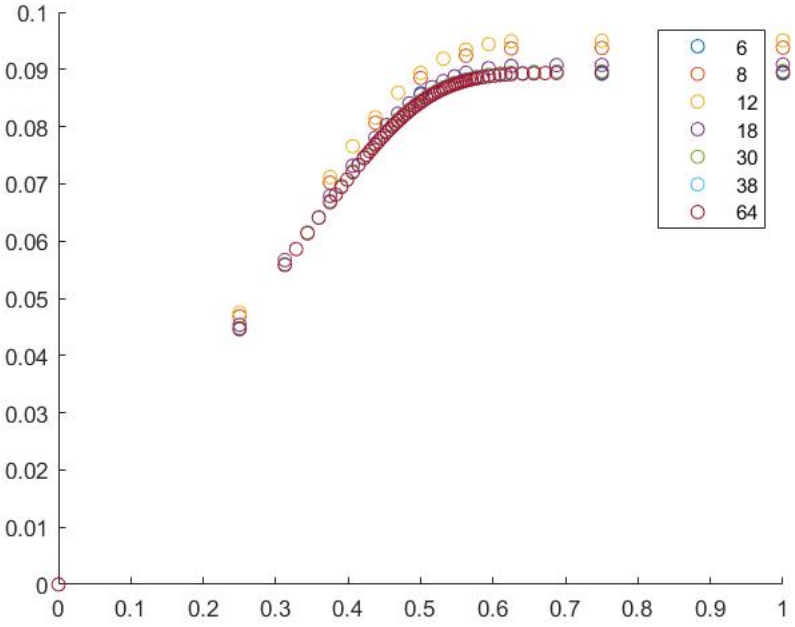

Figure 1.8: Mesh by the adaptive method. $n \leq 64$.

2 NUMERICAL METHODS FOR TWO-DIMENSIONAL HEAT EQUATION

2.1 Three methods for the heat equation

Assume $t_i = ik, 0 \le i \le m, x_j = y_j = jh, 0 \le j \le n$. $U^p_{i,j} = u(ih, jh, pk)$. The boundary condition can be reformulated as

$$U^0_{i,j} = sin(\pi ih)sin(\pi jh),\ 0 \le i, j \le n$$

and

$$U^p_{i,0} = U^p_{i,n} = U^p_{0,i} = U^p_{n,i} = 0,\ 0 \le p \le m, 0 \le i \le n$$

2.1.1 Explicit method

We reduce the partial differential equation to:

$$\frac{U^{p+1}_{i,j} - U^p_{i,j}}{k} = \frac{U^p_{i,j+1} - 2U^p_{i,j} + U^p_{i,j-1}}{h^2} + \frac{U^p_{i+1,j} - 2U^p_{i,j} + U^p_{i-1,j}}{h^2}$$

Rearranging terms gives

$$U^{p+1}_{i,j} = (1 - \frac{4k}{h^2})U^p_{i,j} + \frac{k}{h^2}(U^p_{i,j+1} + U^p_{i,j-1}) + \frac{k}{h^2}(U^p_{i+1,j} + U^p_{i-1,j})$$

Suppose $\mathbf{U}^p := vec((U^p_{i,j})_{1 \le i,j \le n-1})$, then the iteration can be written as:

$$\mathbf{U}^0 = vec((sin(\pi ih)sin(\pi jh))_{1 \le i,j \le n-1})$$

$$\mathbf{U}^{p+1} = \begin{bmatrix} A_0 & \frac{k}{h^2}I & & \\ \frac{k}{h^2}I & A_0 & \frac{k}{h^2}I & \\ & \ddots & \ddots & \ddots \\ & & \frac{k}{h_x^2}I & A_0 \end{bmatrix} \mathbf{U}^p := A_e\mathbf{U}^p$$

where

$$A_0 = \begin{bmatrix} 1 - \frac{4k}{h^2} & \frac{k}{h^2} & & \\ \frac{k}{h^2} & 1 - \frac{4k}{h^2} & \frac{k}{h^2} & \\ & \ddots & \ddots & \ddots \\ & & \frac{k}{h^2} & 1 - \frac{4k}{h^2} \end{bmatrix}_{(n-1)\times(n-1)}$$

STABILITY Insert $u(x, y, t_p) = e^{iax}e^{iby}$ into the scheme, we will get the growth factor G satisfy:

$$\frac{G-1}{k} = \frac{e^{iah} + e^{-iah} - 2}{h^2} + \frac{e^{ibh} + e^{-ibh} - 2}{h^2}$$

Hence we get

$$G = 1 - 2\frac{k}{h^2}(1 - cos(ah)) - 2\frac{k}{h^2}(1 - cos(bh))$$

The worst case is that $ah = bh = \pi$, to ensure $|G| \le 1$, we need to keep $k \le \frac{h^2}{4}$, which is also a necessary and suffiecnt condition.

TRUNCATION ERROR By Taylor expansion, we can get the truncation error is

$$Tu(x, y, t) = \frac{1}{2}u_{tt}(x, y, t)k - \frac{h^2}{12}(u_{xxxx}(x, y, t) + u_{yyyy}(x, y, t)) + \mathcal{O}(k^2 + h^4)$$

According to the real solution, $u_{tt} = 4u_{xxxx} = 4u_{yyyy}$. Hence, we can set $k \approx h^2/12$, i.e. $m = 12n^2$ to get the least truncation error.

2.1.2 Implicit method

We reduce the partial differential equation to:

$$\frac{U_{i,j}^{p+1}-U_{i,j}^{p}}{k}=\frac{U_{i,j+1}^{p+1}-2U_{i,j}^{p+1}+U_{i,j-1}^{p+1}}{h^2}+\frac{U_{i+1,j}^{p+1}-2U_{i,j}^{p+1}+U_{i-1,j}^{p+1}}{h^2}$$

Rearranging terms gives

$$U_{i,j}^{p}=(1+\frac{4k}{h^2})U_{i,j}^{p+1}-\frac{k}{h^2}(U_{i,j+1}^{p+1}+U_{i,j-1}^{p+1})-\frac{k}{h^2}(U_{i+1,j}^{p+1}+U_{i-1,j}^{p+1})$$

Suppose $\mathbf{U}^p := vec((U_{i,j}^p)_{1\le i,j\le n-1})$, then the iteration can be written as:

$$\mathbf{U}^0 = vec((sin(\pi ih)sin(\pi jh))_{1\le i,j\le n-1})$$

$$\mathbf{U}^p=\begin{bmatrix} A_0 & -\frac{k}{h^2}I & & \\ -\frac{k}{h^2}I & A_0 & -\frac{k}{h^2}I & \\ & \ddots & \ddots & \ddots \\ & & -\frac{k}{h^2}I & A_0 \end{bmatrix}\mathbf{U}^{p+1} := A_i\mathbf{U}^{p+1}$$

where

$$A_0=\begin{bmatrix} 1+\frac{4k}{h^2} & -\frac{k}{h^2} & & \\ -\frac{k}{h^2} & 1+\frac{4k}{h^2} & -\frac{k}{h^2} & \\ & \ddots & \ddots & \ddots \\ & & -\frac{k}{h^2} & 1+\frac{4k}{h^2} \end{bmatrix}_{(n-1)\times(n-1)}$$

STABILITY Insert $u(x,y,t_p)=e^{iax}e^{iby}$ into the scheme, we will get the growth factor G satisfy:

$$\frac{G-1}{k}=G(\frac{e^{iah}+e^{-iah}-2}{h^2}+\frac{e^{ibh}+e^{-ibh}-2}{h^2})$$

Hence we get

$$G=(1+2\frac{k}{h^2}(1-cos(ah))+2\frac{k}{h^2}(1-cos(bh)))^{-1}$$

Since $|G|\le 1$ unconditionally, the implicit method is unconditionally stable. The code is in the Appendix Problem 2, implicit.m.

TRUNCATION ERROR By Taylor expansion, we can get the truncation error is

$$Tu(x,y,t)=-\frac{k}{2}u_{tt}(x,y,t)-\frac{h^2}{12}(u_{xxxx}(x,y,t)+u_{yyyy}(x,y,t))+\mathcal{O}(k^2+h^4+h^2k)$$

We can set $k=\mathcal{O}(h^2)$ to get $Tu=\mathcal{O}(h^2)$.

2.1.3 Crank-Nicolson method

We reduce the partial differential equation to:

$$\frac{U_{i,j}^{p+1}-U_{i,j}^{p}}{k}=\frac{U_{i,j+1}^{p+1}-2U_{i,j}^{p+1}+U_{i,j-1}^{p+1}+U_{i,j+1}^{p}-2U_{i,j}^{p}+U_{i,j-1}^{p}}{2h^2}+\frac{U_{i+1,j}^{p+1}-2U_{i,j}^{p+1}+U_{i-1,j}^{p+1}+U_{i+1,j}^{p}-2U_{i,j}^{p}+U_{i-1,j}^{p}}{2h^2}$$

Rearranging terms gives:

$$U_{i,j}^{p+1}(1+\frac{2k}{h^2})-\frac{k}{2h^2}(U_{i,j+1}^{p+1}+U_{i,j-1}^{p+1})-\frac{k}{2h^2}(U_{i+1,j}^{p+1}+U_{i-1,j}^{p+1})$$
$$=U_{i,j}^{p}(1-\frac{2k}{h^2})+\frac{k}{2h^2}(U_{i,j+1}^{p}+U_{i,j-1}^{p})+\frac{k}{2h^2}(U_{i+1,j}^{p}+U_{i-1,j}^{p})$$

Suppose $\mathbf{U}^p := vec((U_{i,j}^p)_{1\le i,j\le n-1})$, then the iteration can be written as:

$$\mathbf{U}^0 = vec((sin(\pi ih)sin(\pi jh))_{1\le i,j\le n-1})$$

$$\begin{bmatrix} A_0 & -\frac{k}{2h^2}I & & \\ -\frac{k}{2h^2}I & A_0 & -\frac{k}{2h^2}I & \\ & \ddots & \ddots & \ddots \\ & & -\frac{k}{2h^2}I & A_0 \end{bmatrix}\mathbf{U}^{p+1} = \begin{bmatrix} A_0' & \frac{k}{2h^2}I & & \\ \frac{k}{2h^2}I & A_0' & \frac{k}{2h^2}I & \\ & \ddots & \ddots & \ddots \\ & & \frac{k}{2h^2}I & A_0' \end{bmatrix}\mathbf{U}^{p}$$

which can be summarized as

$$A_c\mathbf{U}^{p+1} = A_c'\mathbf{U}^p$$

where

$$A_0 = \begin{bmatrix} 1+\frac{2k}{h^2} & -\frac{k}{2h^2} & & \\ -\frac{k}{2h^2} & 1+\frac{2k}{h^2} & -\frac{k}{2h^2} & \\ & \ddots & \ddots & \ddots \\ & & -\frac{k}{2h^2} & 1+\frac{2k}{h^2} \end{bmatrix}_{(n-1)\times(n-1)}$$

$$A_0' = \begin{bmatrix} 1-\frac{2k}{h^2} & \frac{k}{2h^2} & & \\ \frac{k}{2h^2} & 1-\frac{2k}{h^2} & \frac{k}{2h^2} & \\ & \ddots & \ddots & \ddots \\ & & \frac{k}{2h^2} & 1-\frac{2k}{h^2} \end{bmatrix}_{(n-1)\times(n-1)}$$

STABILITY Insert $u(x,y,t_p) = e^{iax}e^{iby}$ into the scheme, we will get the growth factor G satisfy:

$$\frac{G-1}{k} = \frac{G+1}{2}(\frac{e^{iah}+e^{-iah}-2}{h^2}+\frac{e^{ibh}+e^{-ibh}-2}{h^2})$$

Hence we get

$$G = \frac{1-\frac{k}{h^2}(1-cos(ah))-\frac{k}{h^2}(1-cos(bh))}{1+\frac{k}{h^2}(1-cos(ah))+\frac{k}{h^2}(1-cos(bh))}$$

Obviously, $|G|\le 1$ is unconditionally satisfied, so the Crank-Nicolson method is unconditionally stable.

TRUNCATION ERROR

$$Tu(x,y,t) = \frac{k^2}{24}u_{ttt}(x,y,t+\frac{k}{2})+\frac{k^2}{8}(u_{xx}(x,y,t+\frac{k}{2})+u_{yy}(x,y,t+\frac{k}{2}))+$$
$$\frac{h^2}{12}(u_{xxxx}(x,y,t+\frac{k}{2})+u_{yyyy}(x,y,t+\frac{k}{2}))+\mathcal{O}(k^4+h^4+k^2h^2)$$

We set $k=\mathcal{O}(h)$, and try to find suitable ratio. Since $u_t=-2\pi^2u, u_{xx}=u_{yy}=-u$. Hence, we set $k=\sqrt{\frac{2\pi^2}{3+4\pi^4}}h$, i.e. $m=3n$, to make the truncation error to be $\mathcal{O}(h^4)$

2.1.4 Implementation of the three methods

We use "explicit.m", "implicit.m", and "crank.m" to implement the algorithm, which are presented in the Appendix Problem 2. The output of these codes is an $(n-1)^2 \times (m+1)$ matrix, where each column represents spatial distribution of u, and is vectorized by the original $(n-1) \times (n-1)$ matrix. To illustrate stability, we choose different ratio of grid and calculate the L^2 norm of the error. The result is put in the table below, which coincide with our stability analysis.

(m,n)	$\|\|explicit-real\|\|_2$	$\|\|implicit-real\|\|_2$	$\|\|crank-real\|\|_2$	k/h^2
(100,6)	3.6271e+06	0.3902	0.0671	0.36
(1000,10)	0.0291	0.3207	0.1462	0.1
(200,12)	7.5944e+117	0.5117	0.0470	0.72

Table 2.1: Stability of three methods.

2.2 Improvement on the implicit method.

We will use the Cholesky decomposition, Gauss Seidel iteration, conjugate gradient method, and multi-grid method for solving the implicit method. Since the scale of the problem is large, we will formulate our algorithm specified for this problem. We will use "timing.m" to compare the efficiency of different method. The input of this function is among "squareroot", "gauss", "gradient", each of which coincides with aforementioned algorithm. For multi-grid algorithm, we write a independent script "grid.m'.
"grid.m" takes m, n, v, r as input, where m, n are defined as usual, $r+1$ is the number of layers of grid, v is a vector with length 2 whose first coordinate means the number of Gauss-Seidel iteration under restriction part and second means the number iteration under improvement part. [1]
We use the l^2 norm of the last layer to illustrate convergence. The results are presented in the table below. It is intriguing that multi-grid method does not behave better than Gauss-Seidel

Method	square-root	Gauss-Seidel iteration	conjugate gradient	Multi-grid
Time/s	193.808916	184.533899	11.117251	7.415421
Error	7.7145e-08	4.7089e-12	4.7806e-12	7.7145e-08

Table 2.2: Time and error by four methods.

iteration, it may be the reason that the limitation of Matlab's accuracy. By setting $m = 100, n = 6$, we discover that the multi-grid method get the same error as directly employing linear solver "ïn Matlab.
From the table, we can see that conjugate gradient method is the most efficient, hence we will use it in the next problem.[2]

2.3 Error estimate

BILINEAR INTERPOLATION For unit K with center (x_c, y_c), we map it to a standard unit by $\hat{F}$

$$\hat{F} : \xi = \frac{2(x-x_c)}{h}, \mu = \frac{2(y-y_c)}{h}$$

[1] We set $v = [10, 10]$ and $r = 5$ here.
[2] Considering $||X_{real}|| = 1.7122e-07$

On the standard unit, we construct a bilinear form on four basis function $u(\xi,\mu) = a_1\xi_1 + a_2\xi_2 + a_3\xi_3 + a_4\xi_4$, where

$$\phi_1 = \frac{(1-\xi)(1-\mu)}{4}, \phi_2 = \frac{(1-\xi)(1+\mu)}{4}$$
$$\phi_3 = \frac{(1+\xi)(1-\mu)}{4}, \phi_4 = \frac{(1+\xi)(1+\mu)}{4}$$

NUMERICAL INTEGRATION We present the trapezoidal rule in 2D. The integral that we consider is $I = \int\int_{[0,1]\times[0,1]} f(x,y)dxdy$. The grid we have is (ih, jh) $0 \le i, j \le n$. We can write the integral in the following form:

$$I = \int_0^1 F(x)dx$$

where $F(x) := \int_0^1 f(x,y)dy$.
Hence, we can use

$$I \approx h(\frac{1}{2}F(x_0) + F(x_1) + \cdots + F(x_{n-1}) + \frac{1}{2}F(x_n))$$

. In order to compute $F(x_i)$, we can have the following approximation:

$$F(x_i) = h(\frac{1}{2}f(x_i,y_0) + f(x_i,y_1) + \cdots + f(x_i y_{n-1}) + \frac{1}{2}f(x_i,y_n))$$

Thus,

$$I \approx \sum_{i=0}^{n}\sum_{j=0}^{n} h^2 f(x_i,y_j)w_{i,j}$$

where

$$w_{i,j} = \begin{cases} 1, & \text{if } 1 \le i,j \le n-1 \\ \frac{1}{4}, & \text{if}(i,j) = (0,n),(0,0),(n,0),(n,n). \\ \frac{1}{2}, & \text{others} \end{cases}$$

Note in our cases, on the boundary, both the numerical solution and the real solution are zero. Hence, we can simplify the quadrature to

$$I \approx \sum_{i=1}^{n-1}\sum_{j=1}^{n-1} h^2 f(x_i,y_j)$$

where f is the error of quadrature on the last layer.

TRUNCATION ERROR According to the analysis, in the explicit method, we set $m = 12n^2$, in the implicit method, we set $m = n^2$,[3] in the Crank-Nicolson method, we set $m = 3n$.
By depicting the error in Figure 2.4,2.5,2.6, we realize we must have at least one additional point for numerical integration in each square, since the error on the grid point is the smallest. Hence, each time we calculate the error, we use a 1024×1024 even grid. The graph is in Figure 2.1, 2.2, 2.3. Due to the limitation of calculation power, we will not consider $n = 512$ under some circumstances.

n	8	16	32	64	128	256	512
error	3.4566e-11	8.9194e-12	2.2467e-12	5.6204e-13	1.3982e-13	3.4190e-14	

Table 2.3: Explicit

[3]For $n = 512$, it takes more than 2 hours

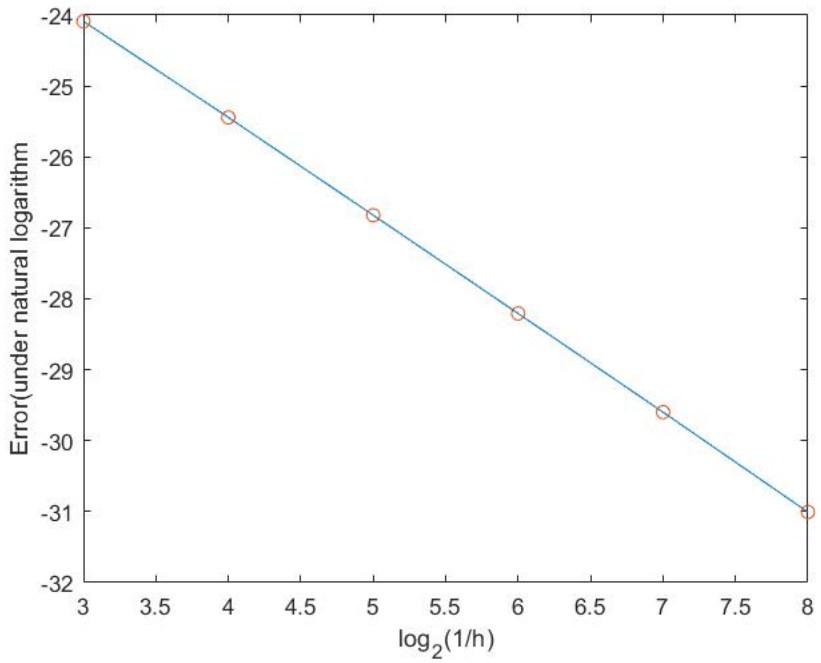

Figure 2.1: Explicit method

n	8	16	32	64	128	256	512
error	1.8605e-08	1.5699e-09	2.9908e-10	6.9926e-11	1.7194e-11	4.2823e-12	1.0773e-12

Table 2.4: Implicit

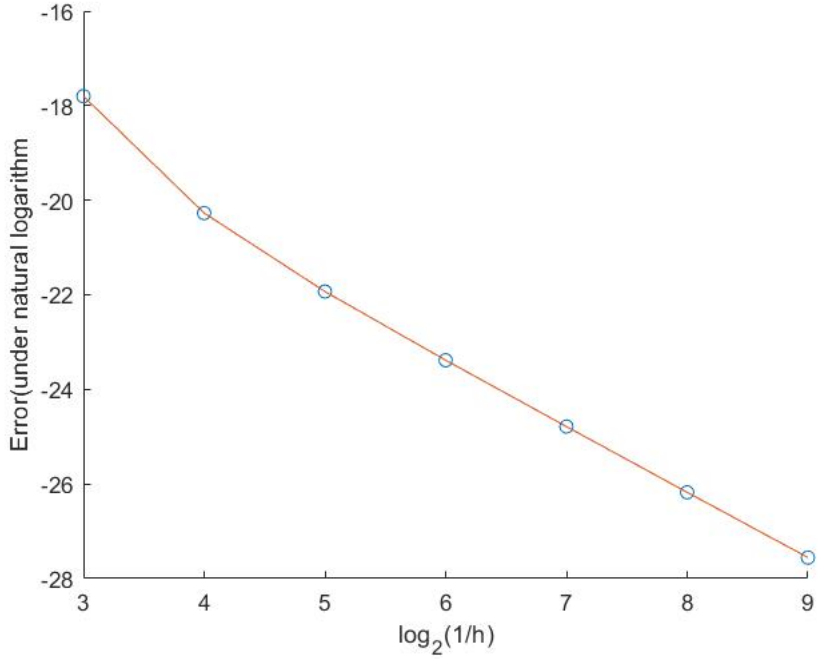

Figure 2.2: Implicit method

n	8	16	32	64	128	256	512
error	8.2759e-10	2.7034e-10	7.2332e-11	1.8405e-11	4.5919e-12	1.1644e-12	2.9522e-13

Table 2.5: Crank-Nicolson

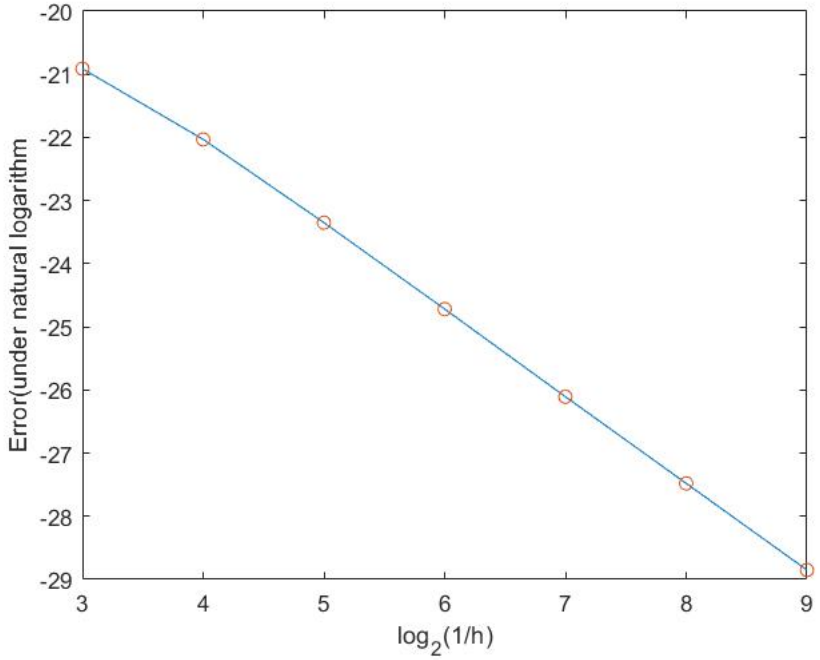

Figure 2.3: Crank-Nicolson method

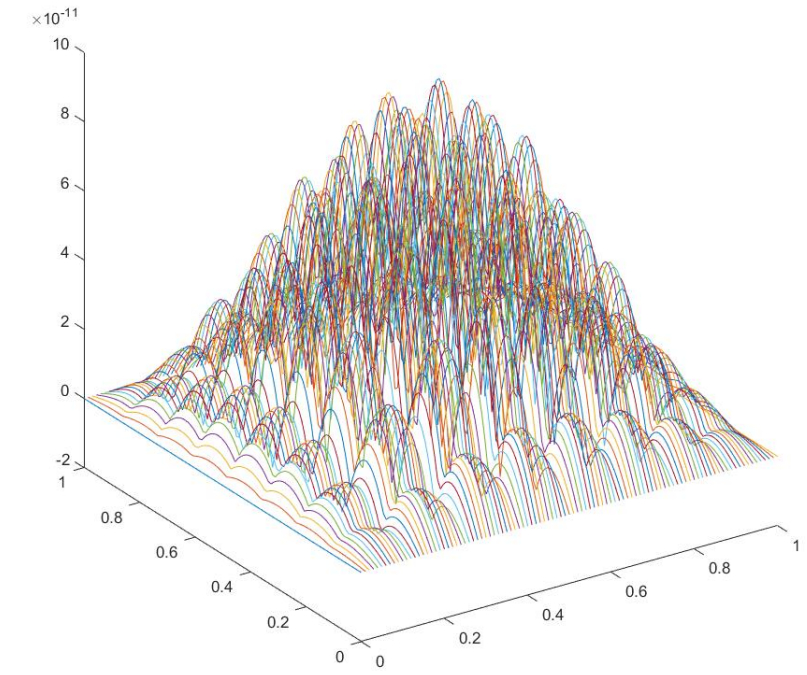

Figure 2.4: Error of Explicit method when $n = 8$

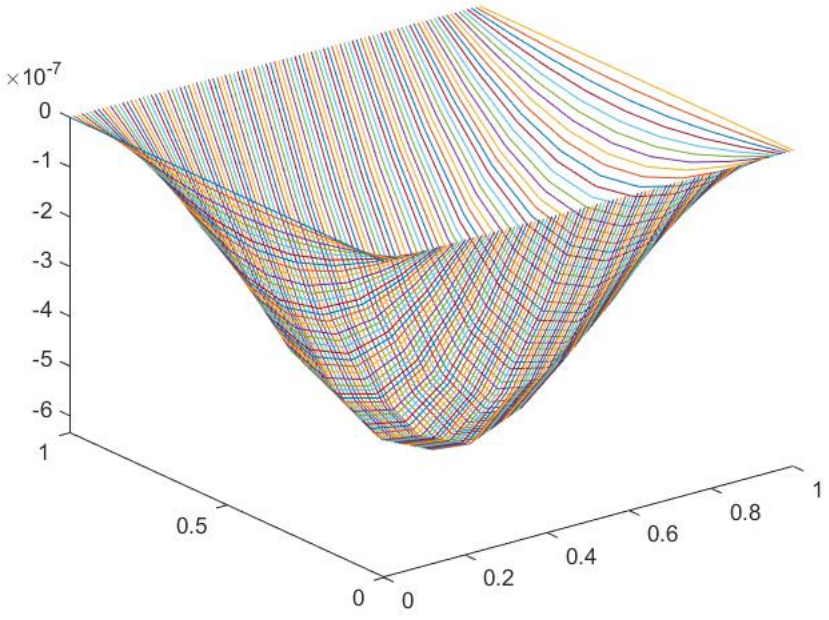

Figure 2.5: Error of Implicit method when $n = 8$

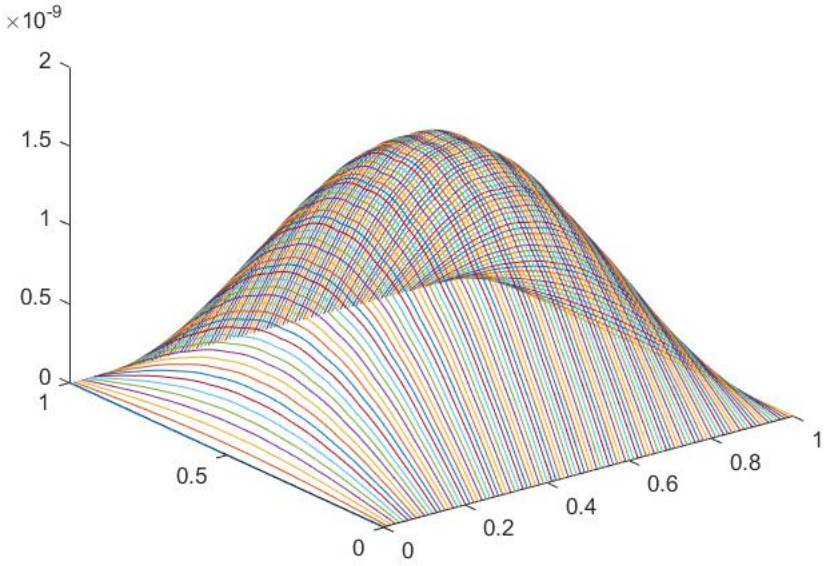

Figure 2.6: Error of Crank-Nicolson method when $n = 8$

3 ROOTS OF NON-LINEAR FUNCTIONS

We implement this problem in file "problem3.ipynb".

3.1 Newton method

The code is in the function "newton", and the input is the initial point and the admitted error. When $x_0 = -2.17731898$, we will get the number of iterative steps is 18, and the result is -1.7875×10^{-4}.
However, if we set $x_0 = -4$ and keep tol unchanged. x_1 will be 23.2899, x_2 will be -6.5111×10^9, indicating that it diverges. The situation occurs because the derivative of f tend to zero quickly when x tend to infinity, leading to non-stability, which indicate the limitation of Newton method.

3.2 Homotopy method

We construct the function

$$h(t,x) = tf(x) + (1-t)(f(x) - f(x_0)) \quad t \in [0,1]$$

For any t, we can get the zero of $h(t,x)$ with respect to x, denoted as $x(t)$. Then $h(t,x(t)) = 0$. Differentiate with respect to t, we get $x'(t) = -\frac{h'_t}{h'_x}$, and obviously $x(0) = x_0$. The root of f is exactly $x(1)$. Hence we can introduce the numerical ordinary differential equation methods to solve the equation. Specifically, the problem can be formulated as follows in the problem.

$$x'(t) = -f(x_0)e^x(1+e^{-x})^2, \;\; x(0) = x_0$$

Since the right side is continuous differentiable with respect to y, the solution is unique. The function are seperately named as "explicit_euler", "implicit_euler', "trapezoid", "improved_euler","runge_kutta"

to represent the explicit Euler method, the implicit Euler method, the trapezoidal method, the improved euler method, and the Runge-Kutta fourth-order method separately. Then we will use $\hat{x}$ as initial value for Newton method. Specifically, in implicit Euler method and trapeziod method, we set the criterion of stopping the iteration is that the difference between consecutive items are less then 10^{-5}.
Here, we derive the Runge-Kutta method in a special form, which is adopted by our code.
In general a Runge-Kutta method of order s can be written as:

$$y_{t+h} = y_t + h \cdot \sum_{i=1}^{s} a_i k_i + \mathcal{O}(h^{s+1})$$

where:

$$k_i = f\left(y_t + h \cdot \sum_{j=1}^{s} \beta_{ij} k_j,\ t + \alpha_i h\right)$$

are increments obtained evaluating the derivatives of y_t at the i-th order.
When $s = 4$, we first get the Taylor series of y_{t+h} around y_h up to fourth order, we can get

$$y_{t+h} = y_t + h\dot{y}_t + \frac{h^2}{2}\ddot{y}_t + \frac{h^3}{6}y_t^{(3)} + \frac{h^4}{24}y_t^{(4)} + \mathcal{O}(h^5)$$

Substitute the derivative of y with the derivative of f, then

$$y_{t+h} = y_t + hf(y_t,\ t) + \frac{h^2}{2}\frac{d}{dt}f(y_t,\ t) + \frac{h^3}{6}\frac{d^2}{dt^2}f(y_t,\ t) + \frac{h^4}{24}\frac{d^3}{dt^3}f(y_t,\ t) + \mathcal{O}(h^5)$$

Then we shall expand the Runge-Kutta method when $s = 4$. To simplify calculation, we set $\alpha_1 = 0, \alpha_2 = \frac{1}{2}, \alpha_3 = \frac{1}{2}, \alpha_4 = 1, \beta_{21} = \frac{1}{2}, \beta_{32} = 1/2, \beta_{43} = 1$, and $\beta_{ij} = 0$ otherwise. Then

$$\begin{aligned}
y_{t+h} &= y_t + h\left\{a_1 \cdot f(y_t,\ t) + a_2 \cdot \left[f(y_t,\ t) + \frac{h}{2}\frac{d}{dt}f(y_t,\ t)\right] + \right. \\
&\quad + a_3 \cdot \left[f(y_t,\ t) + \frac{h}{2}\frac{d}{dt}\left[f\left(y_t,\ t\right) + \frac{h}{2}\frac{d}{dt}f(y_t,\ t)\right]\right] + \\
&\quad \left. + a_4 \cdot \left[f(y_t,\ t) + h\frac{d}{dt}\left[f(y_t,\ t) + \frac{h}{2}\frac{d}{dt}\left[f(y_t,\ t) + \frac{h}{2}\frac{d}{dt}f(y_t,\ t)\right]\right]\right]\right\} + \mathcal{O}(h^5) \\
&= y_t + (a_1 + a_2 + a_3 + a_4) \cdot hf_t + (a_2 + a_3 + 2a_4) \cdot \frac{h^2}{2}\frac{df_t}{dt} + \\
&\quad + (a_3 + 2a_4) \cdot \frac{h^3}{4}\frac{d^2 f_t}{dt^2} + a_4 \cdot \frac{h^4}{4}\frac{d^3 f_t}{dt^3} + \mathcal{O}(h^5)
\end{aligned}$$

Compare the coefficients, we get $a_1 = \frac{1}{6}, a_2 = \frac{1}{3}, a_3 = \frac{1}{3}, a_4 = \frac{1}{6}$. Hence,

$$y_{t+h} = y_t + \frac{h}{6}(k_1 + 2k_2 + 2_k 3 + k_4)$$

where, $k_1 = f(y_t, t), k_2 = f(y_t + \frac{h}{2}k_1, t + \frac{h}{2}), k_3 = f(y_t + \frac{h}{2}k_2, t + \frac{h}{2}), k_4 = f(y_t + hk_3, t + h)$.
The $\hat{x}$ is presented in the following table. We utilize $\hat{x}$ as initial value of the Newton iterative

	Explicit Euler	Implicit Euler	Trapezoidal	Improved Euler	Runge-Kutta
$x_0 = -2.17731898$	0.088298	-0.074070	0.006953	0.000658	0.000018
$x_0 = 4$	0.665728	-0.211814	0.100679	0.040965	0.018358

Table 3.1: $\hat{x}$ obtained by different schemes.

method. The result is presented in the following tables.

	Explicit Euler	Implicit Euler	Trapezoidal	Improved Euler	Runge-Kutta
$x_0 = -2.17731898$	3	3	2	2	2
$x_0 = 4$	5	3	3	3	2

Table 3.2: The number of steps of the Newton method until convergence.

	Explicit Euler	Implicit Euler	Trapezoidal	Improved Euler	Runge-Kutta
$x_0 = -2.17731898$	-5.922454e-18	-3.008797e-16	-1.118896e-16	3.935654e-17	-3.029058e-16
$x_0 = 4$	-2.389920e-16	-3.176712e-16	-2.553025e-16	-1.373989e-16	8.354074e-17

Table 3.3: Numerical estimation of the zero point.

3.3 Conclusion

By the discussion above, we find that numerically, the Newton method can only converge in the neighbourhood of the zero point. If the derivative of initial point is too small, the method is highly unstable since the derivative appears in the denominator. Thus, we use homotopy method to move the initial point to the vicinity of the zero point, since the homotopy method is not iterative, and will not be influence drastically by the decaying derivative. However, note that the homotopy method alone is not accurate enough, since the truncation error is at most polynomial in the size of the mesh, h, while the Newton method converges quadratically in a proper neighbourhood of the zero point. In summary, a combination of homotopy method and Newton method will lead to fast and accurate convergence.

4 Assignment problem

Since the assignment problem is combinatorial, we will use Python to implement our algorithm. We will present details of our algorithm below. For simplicity, we assume the elements in matrix C is uniform in $[0,1]$ in our algorithm.

4.1 Division method

Assume in the i-th step, we consider n situations where "1" in the i-th row lies in the j-th column, where $j = 1,\dots,n$. In each situation, we sample $n-i$ "1"s for remaining $n-i$ rows, under the constraints in the optimization problem. Furthermore, when $i > 1$, for the backtracking step, given that "1" in the $i-1$-th row lies in the j-th column, we will sample $k \in \{1,2,\dots,n\}-\{j\}$ uniformly as the position of "1" in the $i-1$-th row. And "1"s in the remaining $n-i+1$ rows will be sampled uniformly.
Specifically, we will utilize disordered data structure "set' to record rows that have not be occupied by "1", and the method "set.pop()" is able to pop elements uniformly in the given set, which constitute our sample algorithm for each situation. And we will utilize ordered data structure 'list' to record rows that have be occupied by "1", where the i-th element in the list is the number of columns of "1" in the i-th row.
After each iteration, $i = i+1$ if the minimum occurs in the forward step and we proceed to the next row, and $i = i-1$ if the minimum occurs in the backtracking step and we return to the last row. The stopping criterion is when $i = n+1$.

4.2 Hungarian method

We will detail the Hungarian method below (for iterations of rows or columns, we always proceed in the index's order):

1. For every row of C, subtract the minimum of the row from each element in the row.
2. For every column of C, subtract the minimum of the column from each element in the column.
3. Check the number of zeros in each row in order, until we find a row which contains a single zero. Mark this zero as "positive", and then mark the zeros in the same column with the "positive" zero as "negative". Repeat this step, until the last row.
4. Check and mark zeros column by column with the same rule as Step 3.
5. If there exists rows of columns that contains more than one zeros which has not been marked, we randomly mark one as "positive", and mark the zeros in the same row or column with the "positive" zero as "negative".
6. Tick rows that do not have "positive" zeros.
7. Tick columns that contain zeros in the ticked rows.
8. Tick rows that contain "positive" zeros in the ticked columns.
9. Repeat Step 7 and 8, until no row or column can be ticked.
10. Find the minimum value among elements in the intersection of ticked rows and non-ticked columns.
11. Subtract the minimum from each element in the ticked row, and add the minimum to each element in the ticked column.
12. Return to Step 2. Until each row have a "positive" zero.
13. "1"s in matrix X should lie in the position of the positive zeros.

Specifically, we use a dictionary "mark" to record the marks of zeros due to its versatility and efficiency. We define some local functions to achieve terse representation of the main part. Function "count_zeros_row" implements Step 1; function "count_zeros_col" implements Step2; function "mark_zero" implements Step 3, 4 and 5; function "select" implements Step 6~11. The "while" loop in the main function implements the whole iteration above, and the stopping criterion is whether the number of rows which have one "positive" zero equals n.

4.3 Comparison of the two methods

We will first analyze the Hungarian method. If after Step 5, there are rows do not have "positive" zeros, each of them must only have "negative" zeros. We want to subtract some value to produce a "new" zero in these rows. To avoid negative values, we must consider columns that contain such zeros. Step 6~11 achieve this aim. Notice that even though we tick those rows that have a "positive" zero, in the next iteration, this line will also have a "positive" zero, and there is at least one row with a new zero that can be marked as "positive". Thus, the complexity is at most polynomial. As aforementioned analysis, we can get the complexity is actually $\mathcal{O}(n^3)$ However, the division method do not have such guarantee, we expect it to be much slower. As a matter of fact, the division method cannot proceed case larger than 100, yet Hungarian method can solve $n = 512$ in a relatively short period of time.
We present our code and result in the file "problem4.ipynb".