MATH4064 2019–2020	**Computational Applied Mathematics** Coursework 2 Choice FML
Name **Student ID** **Date**	Isaac Oldwood 4286828 May 16, 2020
Existing codes	Names of existing codes that you used

Problem Set 1

Problem 1(a)

• Discretisation: . . .

$$u''(x) = f \quad u(0) = \alpha \quad u(1) = \beta \tag{1}$$

Let

$$x_j = jh \quad h = \frac{1}{(m+1)} \tag{2}$$

where m is the mesh width.

Next we can replace $u''(x)$ by the centered difference approximation.

$$D^2U_j = \frac{1}{h^2}(U_{j-1} - 2U_j + U_{j+1}) \tag{3}$$

This gives

$$\frac{1}{h^2}(U_{j-1} - 2U_j + U_{j+1}) = f(x_j) \qquad for\ j = 1, 2, ..., m \tag{4}$$

Note that j=1 and j-m are satisfied by the boundary conditions. This leaves a linear system of m equations with m unknowns which is of the form

$$AU = F \tag{5}$$

• System: Where

$$U = [U_1, U_2, ..., U_m]^T \tag{6}$$

and

$$A = \frac{1}{h^2}\begin{bmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -2 \end{bmatrix}, F = \begin{bmatrix} f(x_1) - \alpha/h^2 \\ f(x_2) \\ f(x_3) \\ \ddots \\ f(x_{m-1}) \\ f(x_m) - \beta/h^2 \end{bmatrix} \tag{7}$$

We can then solve this for any specific U and F using a simple algorithm that solves tridiagonal linear systems. We can use algorithm 2.5 from Eperson, used in coursework one.

Problem 1(b)

Print-out of code

```
import numpy as np
import matplotlib.pyplot as plt

#Setting up the problem
m=4
h=1/(m+1)
u=(1/(h**2))*np.ones(m-1)
d=(1/(h**2))*(-2.0)*np.ones(m)
```

```
l=(1/(h**2))*np.ones(m-1)
F=np.empty(m)
U=np.empty(m)
x=np.arange(0,(1+h),h)
alpha=0
beta=1
#Edit for given function
def f(x):
    return (np.pi**2.0)*np.sin(np.pi*x)

#Creating F
F[0]=f(x[1]) - ( alpha/(h **2) )
F[m-1]=f(x[m-1]) - ( beta/(h **2) )
for i in range(1,m-1):
    F[i]=f(x[i+1])


#Solving the tridiagonal problem
#Elimination stage
for i in range(1,(m)):
    d[i] = d[i] - u[i-1] *l[i-1] / d[i-1]
    F[i] = F[i] - F[i-1] *l[i-1] / d[i-1]

#backsolve stage
U[(m-1)]= F[(m-1)]/d[(m-1)]
for i in range((m-2),-1,-1):
    U[i] = (F[i] - u[i] * U[i+1])/d[i]

#Insert end values
U=np.insert(U,0,alpha)
U=np.insert(U,m+1,beta)
#Print answer
print(U)
```

Problem 1(c)

- Figure with plots of the coarse approximation and exact solution:

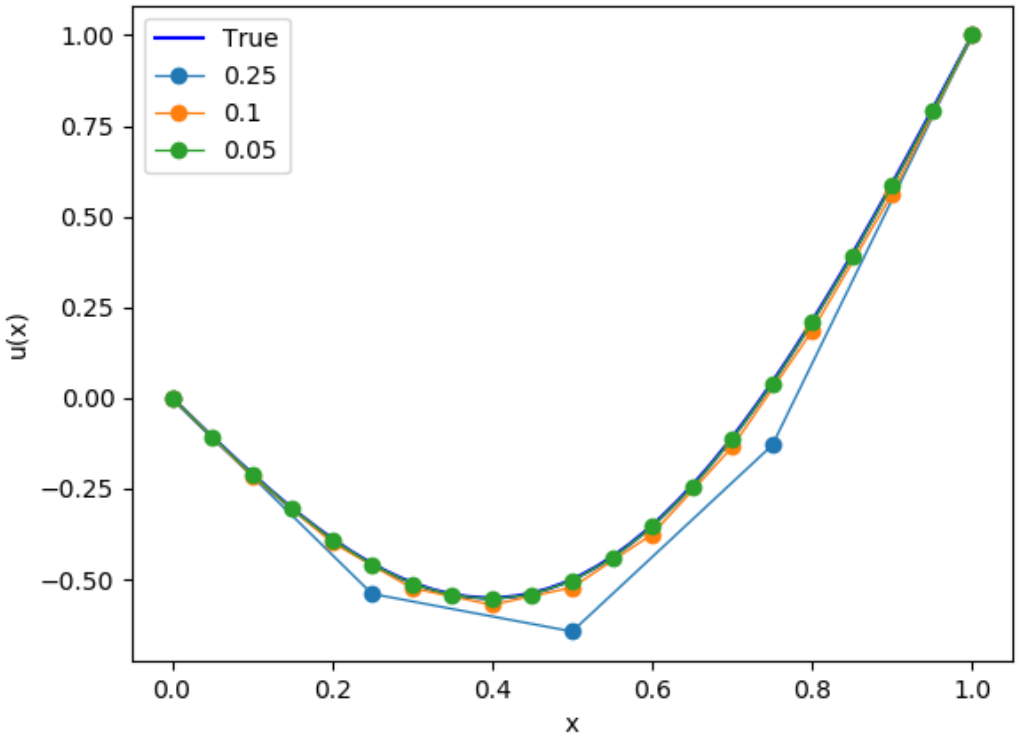


Figure 1: Plot of the approximations and true value

- Convergence study:

  Definition of norm: $\|E\|_{(2)} = \sqrt{h \sum_j \mid E_j \mid^2}$ versus h with $E_j := u(x_j) - U_j$

  Table with convergence results:

h	$Error$	Ratio of Errors
0.25	0.11970412997730362	NaN
0.1	0.019836309388939814	6.0345968410861435
0.05	0.0033792608423897246	5.870014276528028
0.04	0.0019204888124199388	1.7595837166745272
0.02	0.00035434291577207737	5.41985948339164
0.01	7.293217557206249e-05	4.858526610411612
0.005	1.634265083065889e-05	4.462689457651593

  Figure with plot of error-in-norm versus mesh-width (use $\log_{10}$ axes):

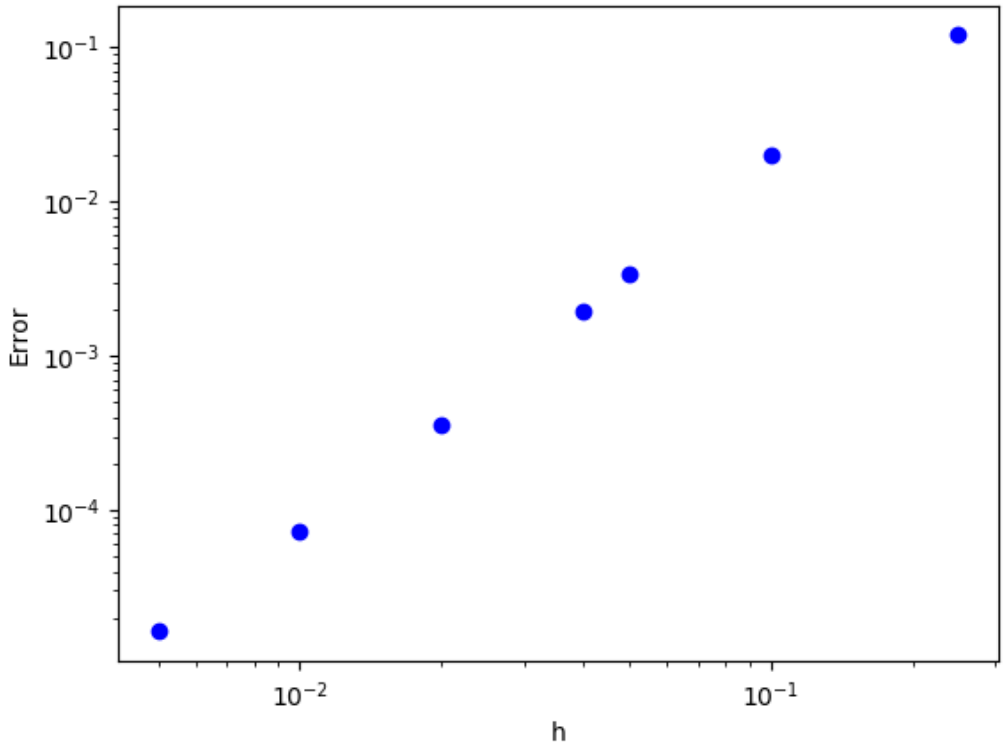


Figure 2: Plot of the values of h versus error

Comment on observed rates of convergence:

When the mesh width is halved the error is 4-6 times smaller. This is supported by the table. It is clear from the plot (straight line) and the table that the rate of convergence is $O(h^2)$ when h is 2 times smaller, the error is 4 times smaller. So the rate of convergence is quadratic.

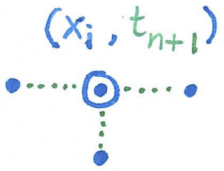
Problem Set 2

Problem 2(a)

- Discretisation:
  $u(t,x)$ is the temperature of a one dimensional piece of metal at position x and time t
  Grid points in space-time:

$$x_i = ih \quad t_n = n\Delta t \tag{8}$$

  Implicit method:



$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = a\frac{u_{i-1}^{n+1} - 2u_i^{n+1} + u_{i+1}^{n+1}}{h^2} + f(x_i, t_{n+1}) \tag{9}$$

  $\forall i = 1, 2, .., N-1$ and $\forall n = 0, 1, ..$
- System:
  Takes the form $[I + \lambda k]\underline{u}^{n+1} = \underline{u}^n + \underline{f}^{n+1}$, with $\lambda = \frac{a\Delta t}{h^2}$ (from lecture notes)
  Where

$$\underline{u}^{(\cdot)} := \begin{pmatrix} u_1^{(\cdot)} \\ u_2^{(\cdot)} \\ \vdots \\ u_{N-2}^{(\cdot)} \\ u_{N-1}^{(\cdot)} \end{pmatrix}, \underline{f}^n := \begin{pmatrix} \Delta t f(x_1, t_n) + \lambda u_0^n \\ \Delta t f(x_2, t_n) \\ \vdots \\ \Delta t f(x_{N-2}, t_n) \\ \Delta t f(x_{N-1}, t_n) + \lambda u_N^n \end{pmatrix}, K := \begin{bmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \\ & & & & -1 & 2 \end{bmatrix} \tag{10}$$

We can then solve this for any specific u and F using a simple algorithm that solves tridiagonal linear systems. We can use algorithm 2.5 from Eperson, used in coursework one.

Problem 2(b)

Print-out of code

```
import numpy as np
import matplotlib.pyplot as plt

#Set values
t=0
a=np.pi**(-2)
```

```
g0=0
g1=1

#Edit for given function
def f():
    return 0
def u0(x):
    return np.sin(2*np.pi*x)+(x)
#Setting up the problem
for m in [3,4,9]:
    h=1/(m+1)
    C=1
    delta_t=1/(4**2)
    UN=np.empty(m)
    lam=(a*delta_t)/(h**2)
    u=(lam*(-1)) *np.ones(m-1)
    d=(1+lam*(2))*np.ones(m)
    l=(lam*(-1)) *np.ones(m-1)
    little_f=np.empty(m)
    F=np.empty(m)
    UN1=np.empty(m)
    x=np.arange(0,(1+h),h)

    #Creating F
    little_f[0]=delta_t*f() + lam* (g0)
    little_f[m-1]=delta_t*f() + lam* (g1)
    UN[0]=u0(x[1])
    UN[m-1]=u0(x[m])
    for i in range(1,m-1):
        little_f[i]=delta_t*f()
        UN[i]=u0(x[i+1])



    for t in np.arange(0,(1+delta_t),delta_t):
        F=UN + little_f
        u=(lam*(-1)) *np.ones(m-1)
        d=(1+lam*(2))*np.ones(m)
        l=(lam*(-1)) *np.ones(m-1)
        #Solving the tridiagonal problem
        #Elimination stage
        for i in range(1,(m)):
            d[i] = d[i] - u[i-1] *l[i-1] / d[i-1]
            F[i] = F[i] - F[i-1] *l[i-1] / d[i-1]

        #backsolve stage
        UN1[(m-1)]= F[(m-1)]/d[(m-1)]
        for i in range((m-2),-1,-1):
            UN1[i] = (F[i] - u[i] * UN1[i+1])/d[i]

        #Reset for next iteration
        UN=UN1

    #Insert boundary values
    UN1=np.insert(UN1,0,g0)
    UN1=np.insert(UN1,m+1,g1)
```

```
#Print answer
print(UN1)
```

Problem 2(c)

- Figure with plots of the coarse approximation and exact solution
  For t=0:

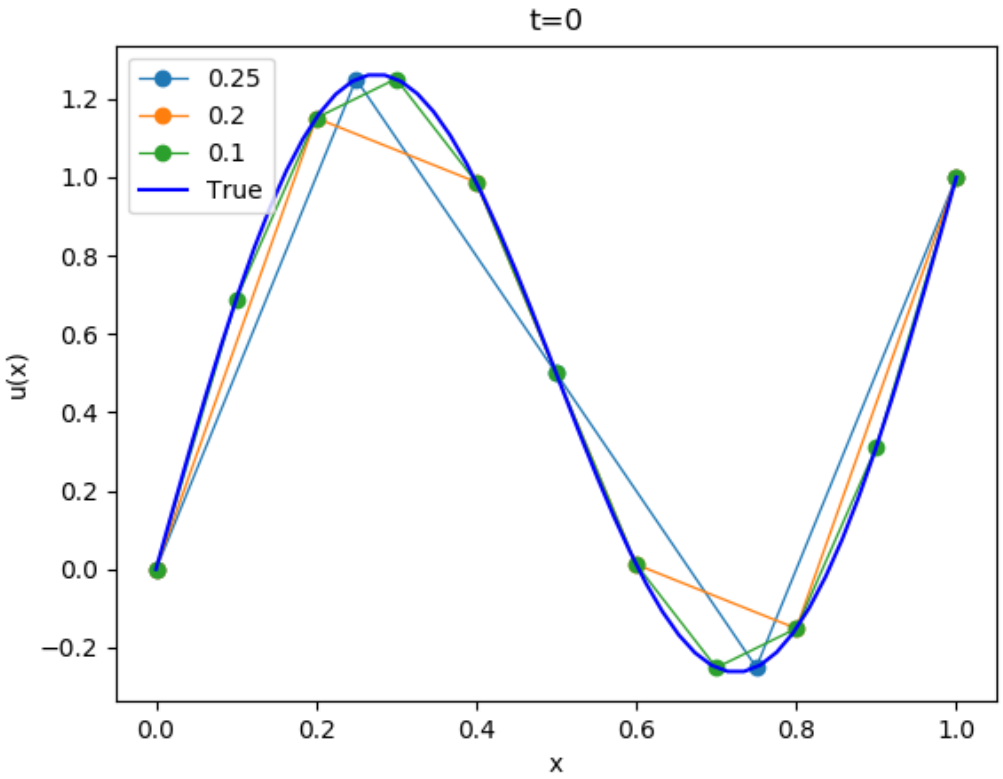


Figure 3: Plot of the approximations (at t=0) and true value

For t=0.5:

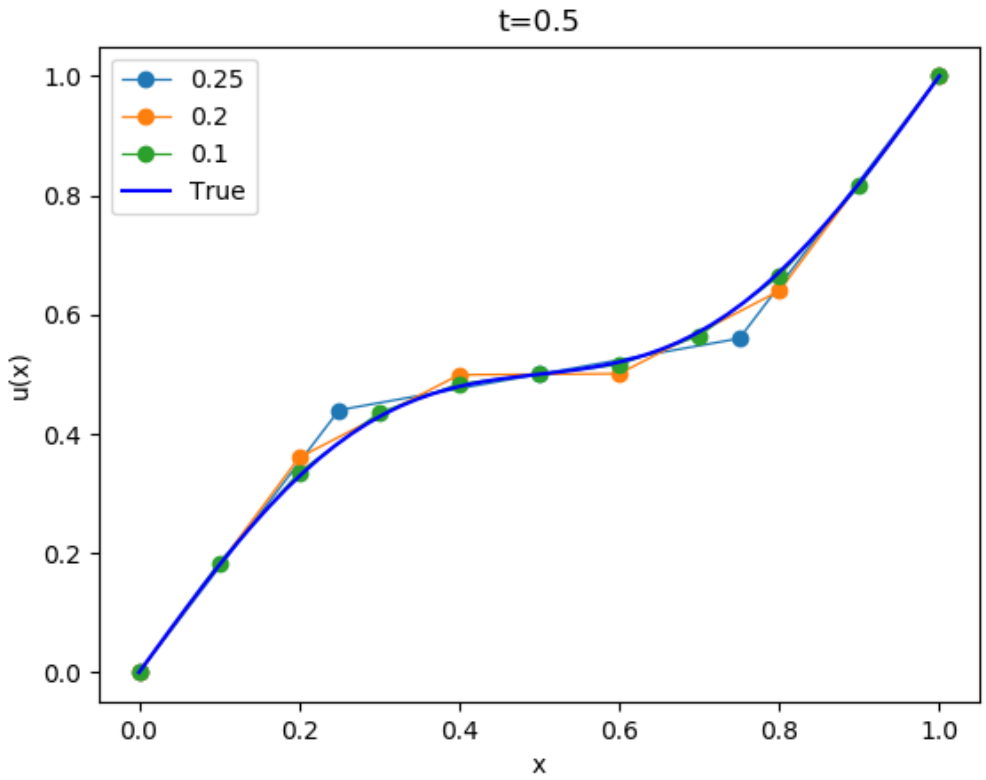


Figure 4: Plot of the approximations (at t=0.5) and true value

For t=1:

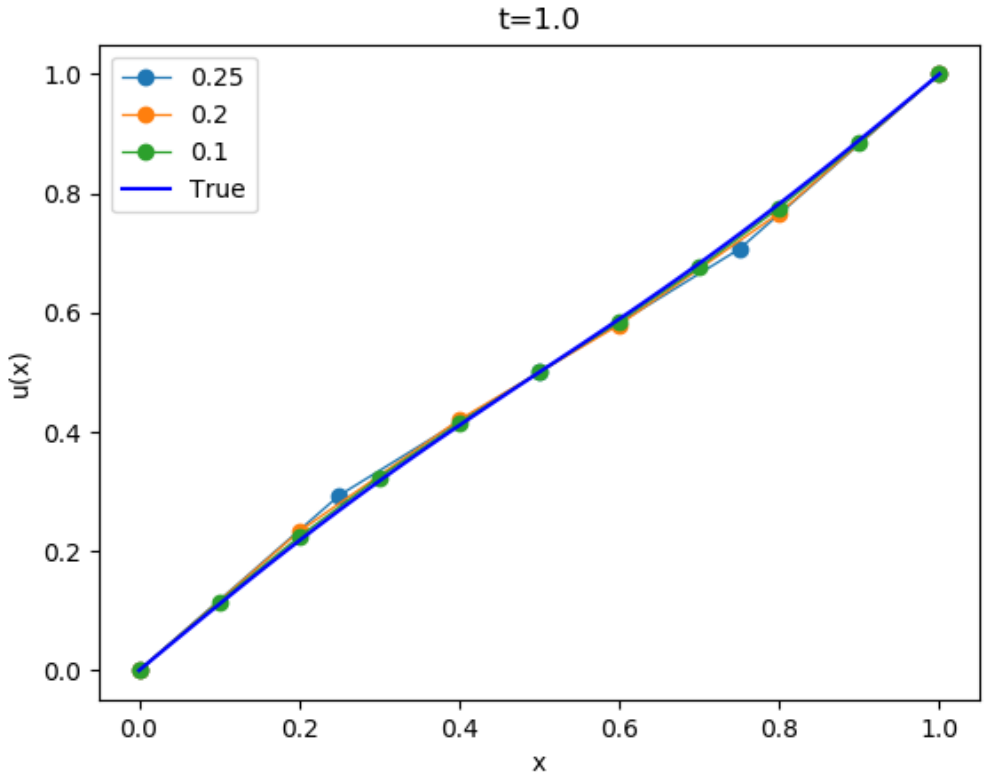


Figure 5: Plot of the approximations (at t=1) and true value

• Convergence study:

Discretization parameters: for this I will be altering h (by varying m) and let $\Delta t = C * h^2$ where $C = 2$.

Definition of norm: I will be using the max-norm at $t = T = 1$ this means that at T=1 the error of at each x_i is calculated and the maximum is the error at that point. $Error = E = \max_i \mid e_i^N \mid$ versus h with $e_j^N := u(N, x_j) - U_j^N$

Table with convergence results:

h	Δt	$Error$	Ratio of Errors
0.25	0.125	0.02847091334475782	NaN
0.125	0.03125	0.006366566443295252	4.47194160279927
0.0625	0.0078125	0.0015342693391356965	4.1495754890609655
0.03125	0.001953125	0.0003797575757754626915	4.040128307772248
0.015625	0.00048828125	9.46974949472823e-05	4.010217775641041

Figure with plot of error-in-norm versus . . . (use $\log_{10}$ axes):

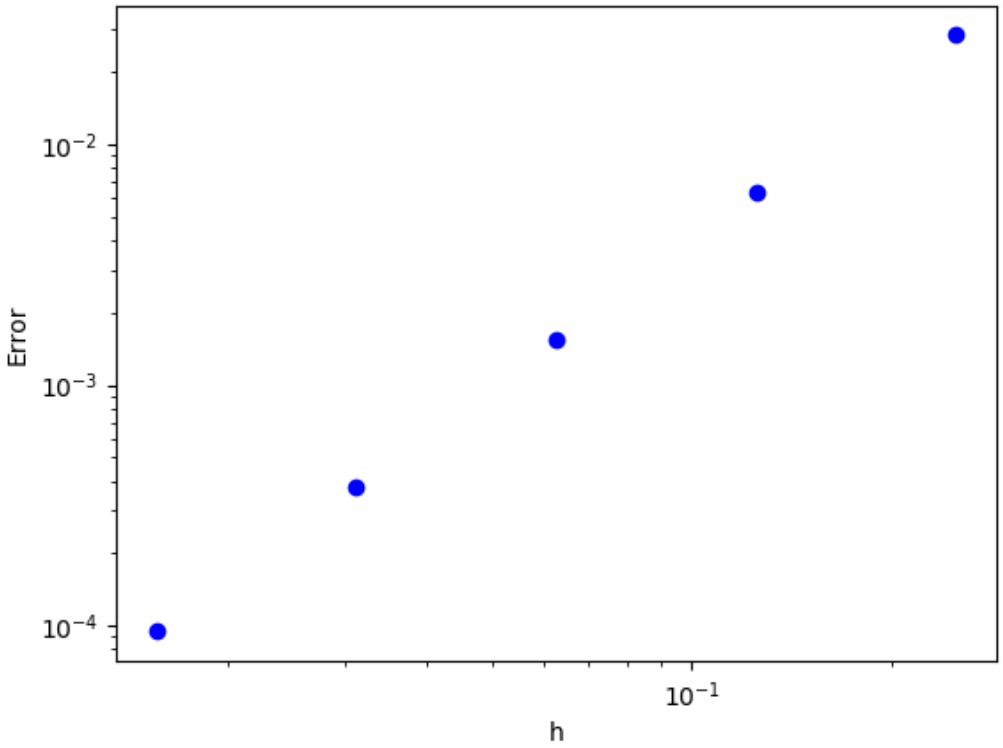


Figure 6: Plot of the values of h versus error

Comment on observed rates of convergence:

It is clear from the ratios in the table and the graph that the rate of convergence is tending to $O(h^2)$. This method has quadratic convergence and if you halve the value of the discretization parameters then the approximation is 4 times more accurate.

Problem Set 3

Problem 3(a)

• Discretisation:

Consider the black-scholes equation for $v(t,x)$ subject to time-dependant Dirichlet boundary conditions and an initial condition.

The solution $v(t,x)$ represents the value of a European option with maturity time T, at time-to-maturity t and spot price x.

Using the implicit finite-difference scheme:

Consider the PDE at an arbitrary point $(x,t) = (x_i, t_{n+1})$.

Then:

$$v = v_i^{n+1} \tag{11}$$

$$\frac{\partial v}{\partial t} = \frac{v_i^{n+1} - v_i^n}{\Delta t} \tag{12}$$

$$\frac{\partial v}{\partial x} = \frac{v_{i+1}^{n+1} - v_i^{n+1}}{h} \tag{13}$$

$$\frac{\partial^2 v}{\partial^2 x} = \frac{v_{i-1}^{n+1} - 2v_i^{n+1} + v_{i+1}^{n+1}}{h^2} \tag{14}$$

Substitute these into the PDE and we obtain:

$$\frac{v_i^{n+1} - v_i^n}{\Delta t} - \frac{1}{2}\sigma^2 x^2 \frac{v_{i-1}^{n+1} - 2v_i^{n+1} + v_{i+1}^{n+1}}{h^2} - rx\frac{v_{i+1}^{n+1} - v_i^{n+1}}{h} + rv_i^{n+1} = 0 \tag{15}$$

with $\sigma > 0$ and $r > 0$

ATTEMPT 1

Rearranging:

$$v_i^{n+1} - v_i^n - \Delta t\frac{1}{2}\sigma^2 x^2 \frac{v_{i-1}^{n+1} - 2v_i^{n+1} + v_{i+1}^{n+1}}{h^2} - \Delta trx\frac{v_{i+1}^{n+1} - v_i^{n+1}}{h} + \Delta trv_i^{n+1} = 0 \tag{16}$$

$$v_i^{n+1} - v_i^n - \frac{\Delta t\sigma^2 x_i^2}{2h^2}(v_{i-1}^{n+1} - 2v_i^{n+1} + v_{i+1}^{n+1}) - \frac{\Delta trx_i}{h}(v_{i+1}^{n+1} - v_i^{n+1}) + \Delta trv_i^{n+1} = 0 \tag{17}$$

$$v_i^{n+1} + \Delta trv_i^{n+1} - \frac{\Delta t\sigma^2 x_i^2}{2h^2}(v_{i-1}^{n+1} - 2v_i^{n+1} + v_{i+1}^{n+1}) - \frac{\Delta trx_i}{h}(v_{i+1}^{n+1} - v_i^{n+1}) = v_i^n \tag{18}$$

$$(1 + \Delta tr)v_i^{n+1} - \frac{\Delta t\sigma^2 x_i^2}{2h^2}(v_{i-1}^{n+1} - 2v_i^{n+1} + v_{i+1}^{n+1}) - \frac{\Delta trx_i}{h}(v_{i+1}^{n+1} - v_i^{n+1}) = v_i^n \tag{19}$$

- System:
  Takes the form:

$$[I + \Delta trI]\underline{v}^{n+1} + \frac{\Delta t\sigma^2 x_i^2}{2h^2}[K]\underline{v}^{n+1} - \frac{\Delta trx_i}{h}(v_{i+1}^{n+1} - v_i^{n+1}) = v_i^n \tag{20}$$

$$[I + \Delta trI]\underline{v}^{n+1} + \frac{\Delta t\sigma^2 x_i^2}{2h^2}[K]\underline{v}^{n+1} + \frac{\Delta trx_i}{h}(v_i^{n+1} - v_{i+1}^{n+1}) = v_i^n \tag{21}$$

$$[I + \Delta trI]\underline{v}^{n+1} + \frac{\Delta t\sigma^2 x_i^2}{2h^2}[K]\underline{v}^{n+1} + \frac{\Delta trx_i}{h}[A]\underline{v}^{n+1} = v_i^n \tag{22}$$

$$([I + \Delta trI] + \frac{\Delta t\sigma^2 x_i^2}{2h^2}[K] + \frac{\Delta trx_i}{h}[A])\underline{v}^{n+1} = \underline{v}^n \tag{23}$$

  Where

$$\underline{v}^{(\cdot)} := \begin{pmatrix} v_1^{(\cdot)} \\ v_2^{(\cdot)} \\ \vdots \\ v_{N-2}^{(\cdot)} \\ v_{N-1}^{(\cdot)} \end{pmatrix}, K := \begin{bmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \\ & & & & -1 & 2 \end{bmatrix}, A := \begin{bmatrix} 1 & -1 & & & & \\ 0 & 1 & -1 & & & \\ & 0 & 1 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 0 & 1 & -1 \\ & & & & 0 & 1 \end{bmatrix} \tag{24}$$

  We can then add the x_i values to the correct parts of the matrix to get the system:

$$((1 + \Delta tr)[I] + \frac{\Delta t\sigma^2}{2h^2}[K] + \frac{\Delta tr}{h}[A])\underline{v}^{n+1} = \underline{v}^n \tag{25}$$

  Where

$$\underline{v}^{(\cdot)} := \begin{pmatrix} v_1^{(\cdot)} \\ v_2^{(\cdot)} \\ \vdots \\ v_{N-2}^{(\cdot)} \\ v_{N-1}^{(\cdot)} \end{pmatrix}, K := \begin{bmatrix} 2x^2 & -1 & & & & \\ -1 & 2x^2 & -1 & & & \\ & -1 & 2x^2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2x^2 & -1 \\ & & & & -1 & 2x^2 \end{bmatrix}, A := \begin{bmatrix} 1x & -1 & & & & \\ 0 & 1x & -1 & & & \\ & 0 & 1x & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 0 & 1x & -1 \\ & & & & 0 & 1x \end{bmatrix} \tag{26}$$

ATTEMPT 2

  Rearranging:

$$\frac{v_i^{n+1} - v_i^n}{\Delta t} - \frac{1}{2}\sigma^2 i^2 \frac{v_{i-1}^{n+1} - 2v_i^{n+1} + v_{i+1}^{n+1}}{h^2} - ri\frac{v_{i+1}^{n+1} - v_i^{n+1}}{h} + rv_i^{n+1} = 0 \tag{27}$$

$$\frac{1}{2}\Delta triv_{i-1}^{n+1} - \frac{1}{2}\Delta t\sigma^2 i^2 v_{i-1}^{n+1} + v_i^{n+1} + \Delta t\sigma^2 i^2 v_i^{n+1} + \Delta trv_i^{n+1} - \frac{1}{2}\Delta triv_{i+1}^{n+1} + \frac{1}{2}\Delta t\sigma^2 i^2 v_{i+1}^{n+1} = v_i^n \tag{28}$$

$$\frac{1}{2}\Delta t(ri-\sigma^2 i^2)v_{i-1}^{n+1}+(1+\Delta t(\sigma^2 i^2+r))v_i^{n+1}-\frac{1}{2}\Delta t(ri+\sigma^2 i^2)v_{i+1}^{n+1}=v_i^n \tag{29}$$

$$a_i v_{i-1}^{n+1}+b_i v_i^{n+1}+c_i v_{i+1}^{n+1}=v_i^n \tag{30}$$

Where

- $a_i=\frac{1}{2}\Delta t(ri-\sigma^2 i^2)$
- $b_i=1+\Delta t(\sigma^2 i^2+r)$
- $c_i=-\frac{1}{2}\Delta t(ri+\sigma^2 i^2)$

h is the step size in space and Δt is te step size in time. Since $x\in[0,\mathbb{R}]$ it is clear that $h=R/(m+1)$ where m is the number of mesh points in space.

We can then create a tridiagonal matrix system from this:

$$[K]\underline{v}^{n+1}=\underline{v}^n+\underline{f}^{n+1} \tag{31}$$

Where

$$\underline{v}^{(\cdot)}:=\begin{pmatrix} v_1^{(\cdot)} \\ v_2^{(\cdot)} \\ \vdots \\ v_{N-2}^{(\cdot)} \\ v_{N-1}^{(\cdot)} \end{pmatrix}, \underline{f}^n:=\begin{pmatrix} -a_1 v_0^n \\ 0 \\ \vdots \\ 0 \\ -c_{N-1}v_N^n \end{pmatrix}, K:=\begin{bmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & & \\ & a_3 & b_3 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & a_{N-2} & b_{N-2} & c_{N-2} \\ & & & & a_{N-1} & b_{N-1} \end{bmatrix} \tag{32}$$

This equation is just a tridiagonal matrix of the form Ax=b. So we can use the same algorithm as the previous two questions.

Problem 3(b)

Print-out of code

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm

#Set values
r=0
sigma=0.5
K=100
R=300
t=0.01
#Define funcitons
def d_plus(t,x):
    return (1/ (sigma * np.sqrt(t)) ) * ( np.log(x/K) + (r + ( (sigma**2)/2 ) ) * t )
def d_minus(t,x):
    return (1/ (sigma * np.sqrt(t)) ) * ( np.log(x/K) + (r - ( (sigma**2)/2 ) ) * t )
def f0(t):
    return K * np.exp(-r *t)
```

```
def v_exact(t,x):
    return K * np.exp(-r *t) * norm.cdf( (-d_minus(t,x)) - x * norm.cdf((-d_plus(t,x)))
def fR(t):
    return v_exact(t,R)
def g(x):
    if x<K:
        return K-x
    else:
        return 0
def a(i):
    return 0.5 * delta_t * (r*i - (sigma**2) * (i**2))
def b(i):
    return 1 + delta_t* ( (sigma**2)*(i**2) + r )
def c(i):
    return -0.5 * delta_t * (r*i + (sigma**2)*(i**2) )

#Setup problem
#Setting up the problem
for m in [3,4,9]:
    h=R/(m+1)
    C=1
    delta_t=1/(4**2)
    VN=np.empty(m)
    u=np.empty(m-1)
    d=np.empty(m)
    l=np.empty(m-1)
    little_f=np.empty(m)
    F=np.empty(m)
    VN1=np.empty(m)
    x=np.arange(0,(1+h),h)

    #Creating F
    little_f[0]=-a(1) * f0(t)
    little_f[m-1]=-c(m-1) * fR(t)
    VN[0]=g(x[1])
    VN[m-1]=g(x[m])
    for i in range(1,m-1):
        little_f[i]=0
        VN[i]=g(x[i+1])

    #Creating tridiagonal matrix K
    for i in range(0,m):
        if i != (m-1):
            u[i]=c(i+1)
            l[i]=a(i+2)
        d[i]=b(i+1)



    for t in np.arange(0,(1+delta_t),delta_t):
        F=VN + little_f
        u=np.empty(m-1)
        d=np.empty(m)
        l=np.empty(m-1)
        #Creating tridiagonal matrix K
        for i in range(0,m):
```

```
            if i != (m-1):
                u[i]=c(i+1)
                l[i]=a(i+2)
            d[i]=b(i+1)

        #Solving the tridiagonal problem
        #Elimination stage
        for i in range(1,(m)):
            d[i] = d[i] - u[i-1] *l[i-1] / d[i-1]
            F[i] = F[i] - F[i-1] *l[i-1] / d[i-1]

        #backsolve stage
        VN1[(m-1)]= F[(m-1)]/d[(m-1)]
        for i in range((m-2),-1,-1):
            VN1[i] = (F[i] - u[i] * VN1[i+1])/d[i]

        #Reset for next iteration
        VN=VN1

    #Insert boundary values
    VN1=np.insert(VN1,0,f0(t))
    VN1=np.insert(VN1,m+1,fR(t))

    #Print answer
    #print(VN1)
    #Plotting
    x_approx=np.linspace(0,1,num=m+2)
    plt.plot(x_approx,VN1,'o-',label=f'{h}',linewidth=0.9)


#Calculate true values and plot
x_true=np.linspace(0,1)
t_true=0.01
y_true=v_exact(t_true,x_true)

plt.plot(x_true,y_true,'b',label='True')
plt.legend(loc=0)
plt.title(f't={t}')
plt.xlabel('x')
plt.ylabel('u(x)')
plt.show()
```

Problem 3(c)

The first step in verifying the code is to see if the approximation and exact function match up for t=0.

I am unable to run my code due to an error.

The error is from the d_minus() and d_plus() functions. When trying to to perform these calculations for t=0 I get a "dividing by 0" error.

In the question it states

$$d_{\pm}(t,x) = 1/(\sigma\sqrt{t})\ldots \tag{33}$$

So when I try calculate the true values at t=0 I am unable to. This means I cannot calculate an error and I am unable to proceed.

However if I could this is the method I would use:

• Convergence study:

Discretization parameters: for this I will be altering h (by varying m) and let $\Delta t = C * h^2$ where $C = 2$.

Definition of norm: I will be using the max-norm at $t = T = 1$ this means that at T=1 the error of at each x_i is calculated and the maximum is the error at that point. $Error = E = \max_i \mid e_i^N \mid$ versus h with $e_j^N := u(N, x_j) - U_j^N$

Problem 3(d)

I am unable to run my code due to an error.

The error is from the d_minus() and d_plus() functions. When trying to to perform these calculations for t=0 I get a "dividing by 0" error.

In the question it states

$$d_{\pm}(t, x) = 1/(\sigma\sqrt{t}) \dots \tag{34}$$

So when I try calculate the true values at t=0 I am unable to. This means I cannot calculate an error and I am unable to proceed.

However if I could this is the method I would use:

• Convergence study:

Discretization parameters: for this I will be altering h (by varying m) and let $\Delta t = C * h^2$ where $C = 2$.

Definition of norm: I will be using the max-norm at $t = T = 1$ this means that at T=1 the error of at each x_i is calculated and the maximum is the error at that point. $Error = E = \max_i \mid e_i^N \mid$ versus h with $e_j^N := u(N, x_j) - U_j^N$

Problem Set 4

Problem 4(a)

The problem is a 1D-input-1D-output classification problem where the goal is construct the map $F : (0, 1) \to \mathbb{R}$ that minimises:

$$Cost = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}(y(x_i) - F(x_i))^2 = \frac{1}{N}\sum_{i=1}^{N} C_{x_i} \tag{35}$$

where $y(x_i)$ is a given output for the input data. For this problem set let $F(\cdot)$ be the simple linear polynomial

$$F(x) = Wx + b \tag{36}$$

where W and b are to be trained.

For gradient descent we need to find the gradient of the cost function with respect to the two values in the parameter $p = (W, b)$. So our cost function is:

$$Cost = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}(y(x_i) - F(x_i))^2 = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}(y(x_i) - (Wx_i + b))^2 = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}(y(x_i) - Wx_i - b)^2 \tag{37}$$

So we need to differentiate F with respect to the two parts of the parameter p:

$$F_W = x_i \tag{38}$$

$$F_b = 1 \tag{39}$$

We can then use the chain rule to find our two partial derivatives of the cost function:

$$Cost_W = \frac{1}{N}\sum_{i=1}^{N} -x_i(-Wx_i + y(x_i) - b) = \frac{1}{N}\sum_{i=1}^{N} x_i(Wx_i - y(x_i) + b) \tag{40}$$

$$Cost_b = \frac{1}{N}\sum_{i=1}^{N} -(-Wx_i + y(x_i) - b) = \frac{1}{N}\sum_{i=1}^{N}(Wx_i - y(x_i) + b) \tag{41}$$

Print-out of code

```
import numpy as np
import matplotlib.pyplot as plt

#Create data
x=np.array([0.,0.5,1.])
y_exact=np.array([0.,1.,1.])

#Intiliase W and b
W=0.5
b=0.5

#Set other constants
```

```
N=3
eta=0.75
MaxIter=64

#Initialise approximation
F=W * x + b

#Functions
def cost():
    return (1/N) * np.sum( 0.5* (y_exact - F)**2 )
#Partial derivates of cost
def cost_W():
    return (1/N) * np.sum( x*(W*x- y_exact +b) )
def cost_b():
    return (1/N) * np.sum( W*x - y_exact + b )

#Cost_vec
cost_vec=np.empty(MaxIter)
j=np.arange(0,MaxIter,1)

#Peform gradient descent
for i in range(0,MaxIter):
    #Forward pass
    F=W*x+b
    #Alter weights and biases
    W= W - eta * cost_W()
    b= b - eta * cost_b()
    #Calculate newcost
    newcost=cost()
    cost_vec[i]=newcost

    #print(newcost)

plt.plot(j,cost_vec)
plt.title('Cost')
plt.xlabel('Iteration')
plt.ylabel('Cost')
plt.show()
```

Problem 4(b)

Generate a table with the first 16 values of the parameters $(w(j), b(j))$ and the first 16 values of the corresponding Cost.

W	b	Cost
0.5	0.5	0.05208333333333333
0.53125	0.42578125	0.05208333333333333
0.58056640625	0.38873291015625	0.04639434814453125
0.6283645629882812	0.36154651641845703	0.042514410490791
0.671420693397522	0.3386038690805435	0.039328247059377944
0.7096252758055925	0.3185414888430387	0.03680405034210665
0.7434143188002054	0.30085500266068266	0.034826595543181196
0.7732767181773852	0.2852349813486512	0.03328150719381237
0.7996646257412081	0.27143451068420976	0.03207495304702297
0.8229814886905019	0.25924056941211426	0.031132880347759746
0.8435845599451772	0.2484659323735871	0.030397334567949658
0.8617896603222142	0.23894536047256648	0.029823043201923763
0.8778758812943098	0.23053288463277544	0.029374654904449772
0.8920898366525472	0.22309953241348868	0.02902456779467873
0.9046494380435679	0.2165313438370342	0.028751231097540388
0.9157472347160651	0.21072762294073413	0.028537818644404828

Plot in a figure the Cost versus the iteration number j =0,1,2,..., MaxIter.

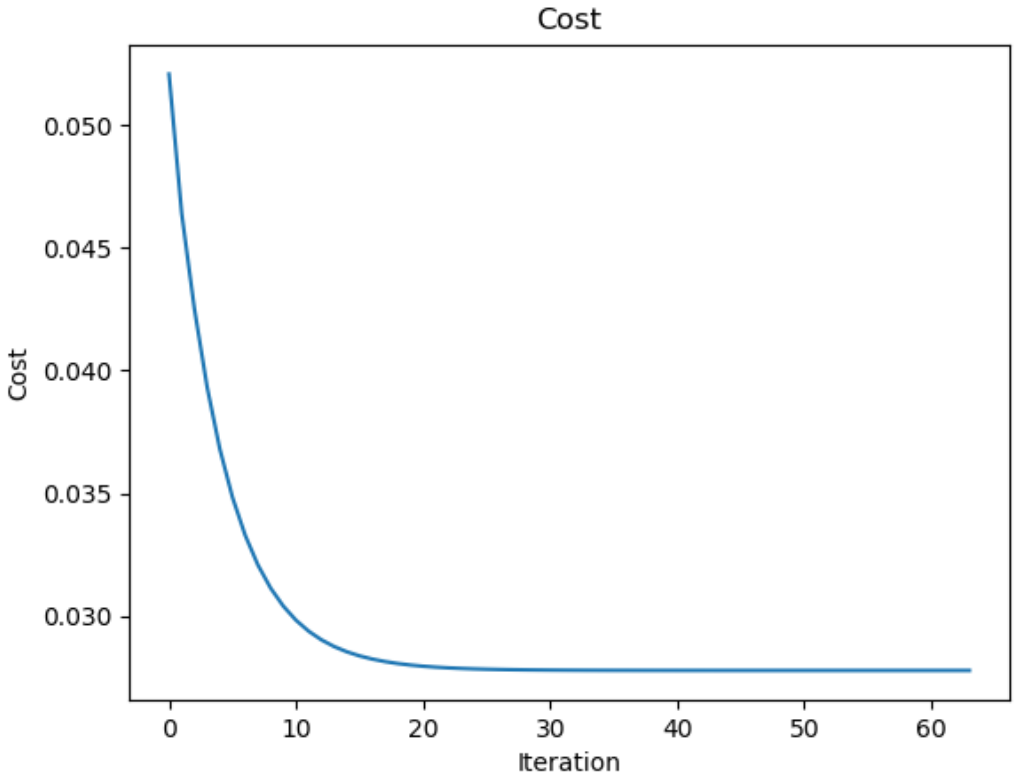


Figure 7: Plot of iterations vs cost

It is clear from the graph that the cost function is converging to 0 although it is very slow after a few iterations. It is finding a minimum quickly so I believe my code is performing as expected.

Problem 4(c)

Code for stochastic gradient descent. The algorithm and process is the same as before except this time I just use one value to calculate the gradient of the cost. This one value is chosen randomly

Print-out of code

```
import numpy as np
import matplotlib.pyplot as plt
import random

#Create data
x=np.array([0.,0.5,1.])
y_exact=np.array([0.,1.,1.])

#Intiliase W and b
W=0.5
b=0.5

#Set other constants
N=3
eta=0.75
MaxIter=64

#Initialise approximation
F=W * x + b

#Functions
def cost():
    return (1/N) * np.sum( 0.5* (y_exact - F)**2 )
#Partial derivates of cost
def cost_W(k):
    return (1/N) * np.sum( x[k]*(W*x[k]- y_exact[k] +b) )
def cost_b(k):
    return (1/N) * np.sum( W*x[k] - y_exact[k] + b )

#Cost_vec
cost_vec=np.empty(MaxIter)
j=np.arange(0,MaxIter,1)

#Peform stochastic gradient descent
for i in range(0,MaxIter):
    #Pick random index
    k=random.randint(0, (N-1))
    #Forward pass
    F=W*x+b
    #Alter weights and biases
    W= W - eta * cost_W(k)
    b= b - eta * cost_b(k)
    #Calculate newcost
    newcost=cost()
    cost_vec[i]=newcost

    #print(newcost)

plt.plot(j,cost_vec)
plt.title('Cost')
```

```
plt.xlabel('Iteration')
plt.ylabel('Cost')
plt.show()
```

Problem 4(d)

Stochastic gradient descent is much more sensitive to values of η because there is only one value being used in calculating the gradient. This loses some robustness of the algorithm so it is more important that good values of η and MaxIter are chosen.

The below plot shows the cost function for the same data as before with the same values as gradient descent.

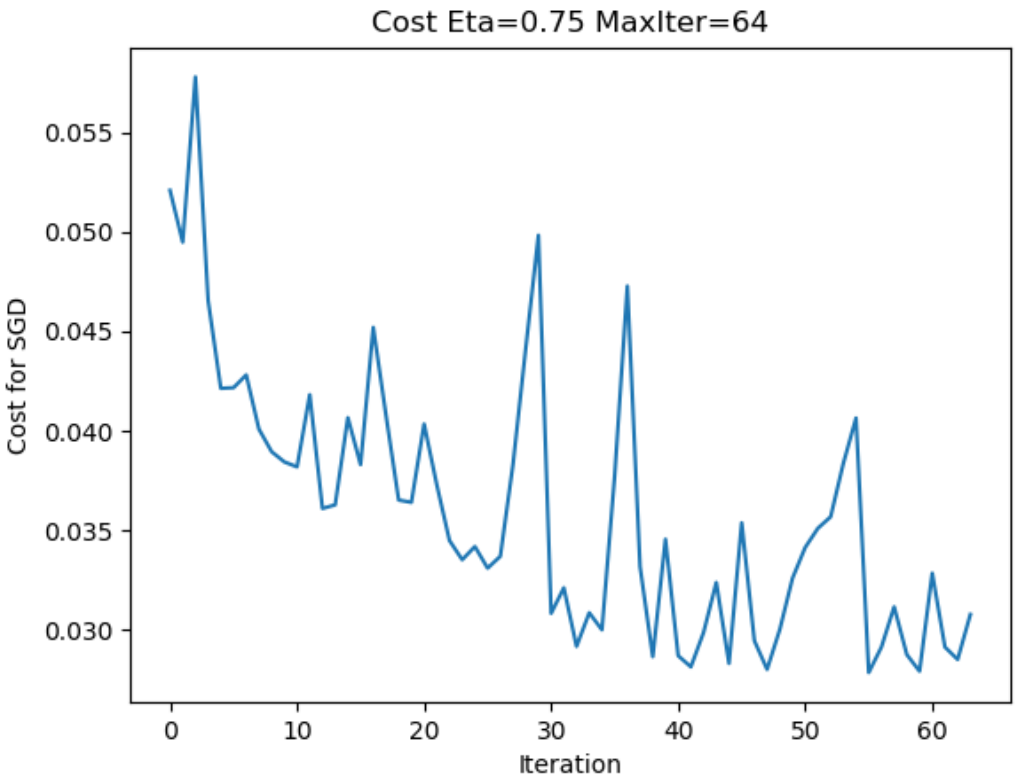
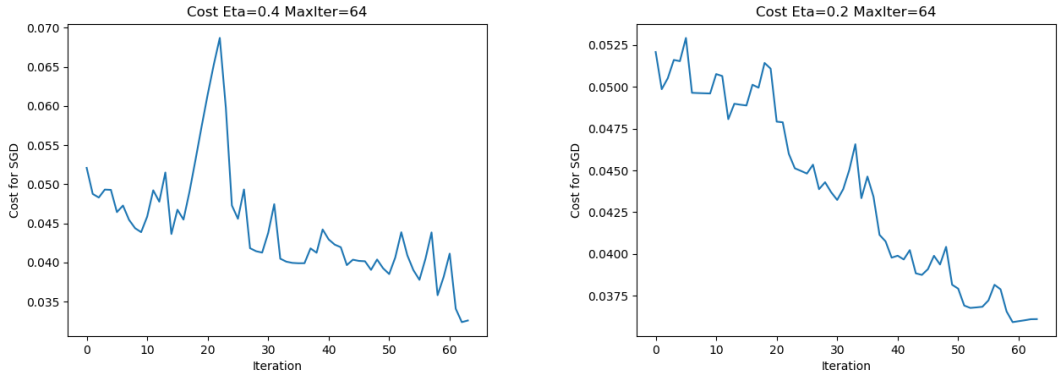


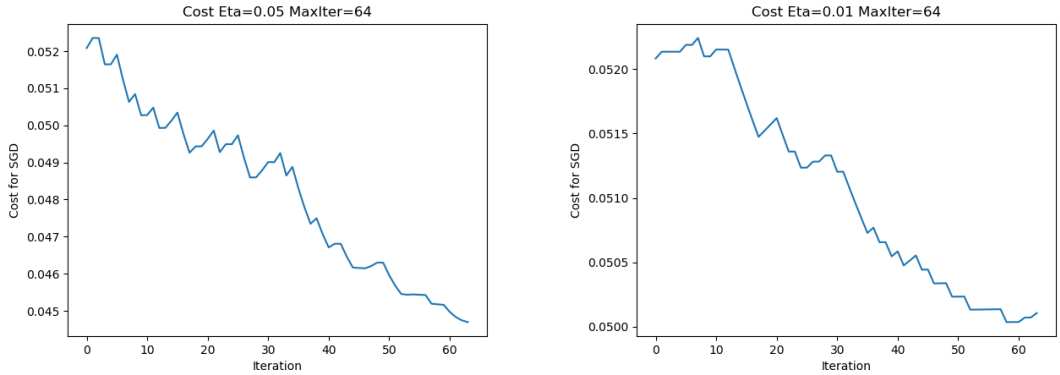
Figure 8: Plot of iterations vs cost for SGD ($\eta = 0.75$)

Clearly the behaviour of the cost is erratic and not converging to 0. To encourage the correct behaviour we have to reduce the size of the learning rate η, however it will take more iterations to meet the same point. Below are some graphs for varying values of η.



(a)

(b)



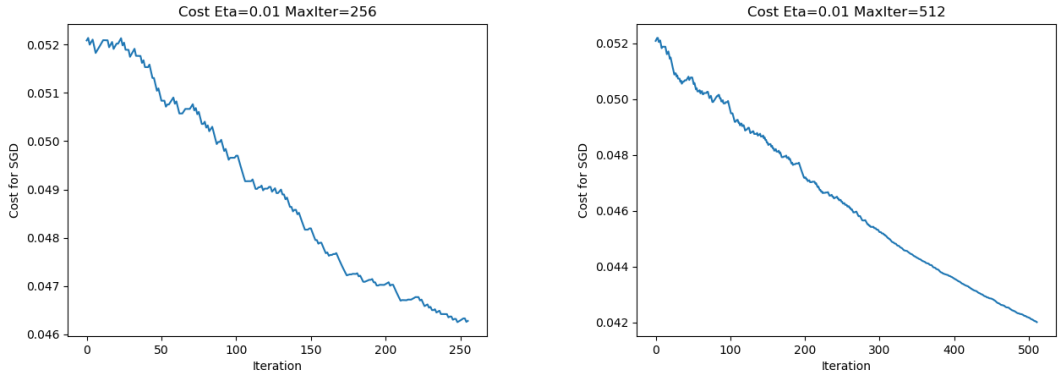
(c)

(d)

Figure 9: Plot of iterations vs cost for SGD and varying η

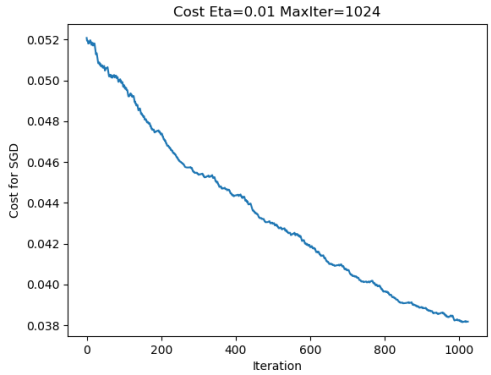
It is clear from these graphs that smaller η gives a smoother cost function. However it is also clear that as η gets smaller, we need to increase MaxIter to make up for the slow convergence. In (b) the cost function drops quickly at the start then sort of plateaus. The cost fucntions in (c) and (d) have not made it that far yet. So we need to increase the value of MaxIter to get to this point.

For the next plot I will use the same $\eta = 0.01$ as plot (d) and vary MaxIter.

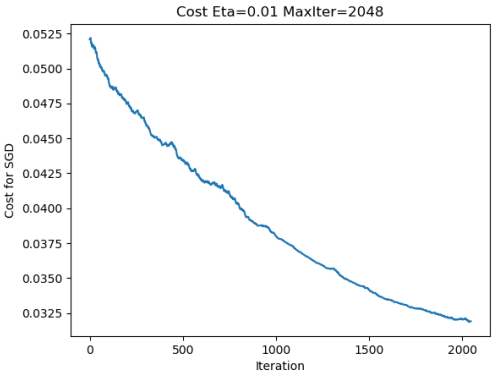


(a)

(b)



(c)



(d)

Figure 10: Plot of iterations vs cost for SGD and varying η

These graphs better represent a cost function converging to 0. By the end of (c) the method starts to slow down. In plot (d) the rate of convergence is beginning to slow down but considering it did not take that long to iterate through I with 2048 is a good value for MaxIter.

Below is a table of the first 16 values with $\eta = 0.01$ and MaxIter = 2048.

W	b	Cost
0.5	0.5	0.05208333333333333
0.5	0.49833333333333335	0.05194583333333334
0.5	0.49667222222222224	0.05181155557098766
0.500422212962963	0.49751594445987657	0.05186133450222669
0.5008426678780608	0.49835615353188023	0.051912041408608764
0.5012613720656093	0.49919286406666463	0.05196366299172936
0.5012613720656093	0.49752888785310906	0.0518266763241234
0.5012654045325469	0.4975329068784902	0.051826832983308854
0.5012654045325469	0.49587446385556194	0.051693046847410176
0.5012654045325469	0.4942215489760434	0.05156244336167201
0.501690647447143	0.4950713260670447	0.051610741927154875
0.501690647447143	0.49342108831348785	0.05148132225535663
0.5021168700937478	0.49427282323562	0.05152913768525738
0.5021168700937478	0.4926252471581679	0.051400889039434086
0.5025440639567395	0.49347892289437945	0.05144822316882611
0.5029694790319515	0.49432904401967825	0.05149652239659516

It is clear from this table and the plots that stochastic gradient descent takes a lot more iterations to minimise the cost function compared to gradient descent.

Plot in a figure the Cost versus the iteration number j = 0, 1, 2, ...,MaxIter.

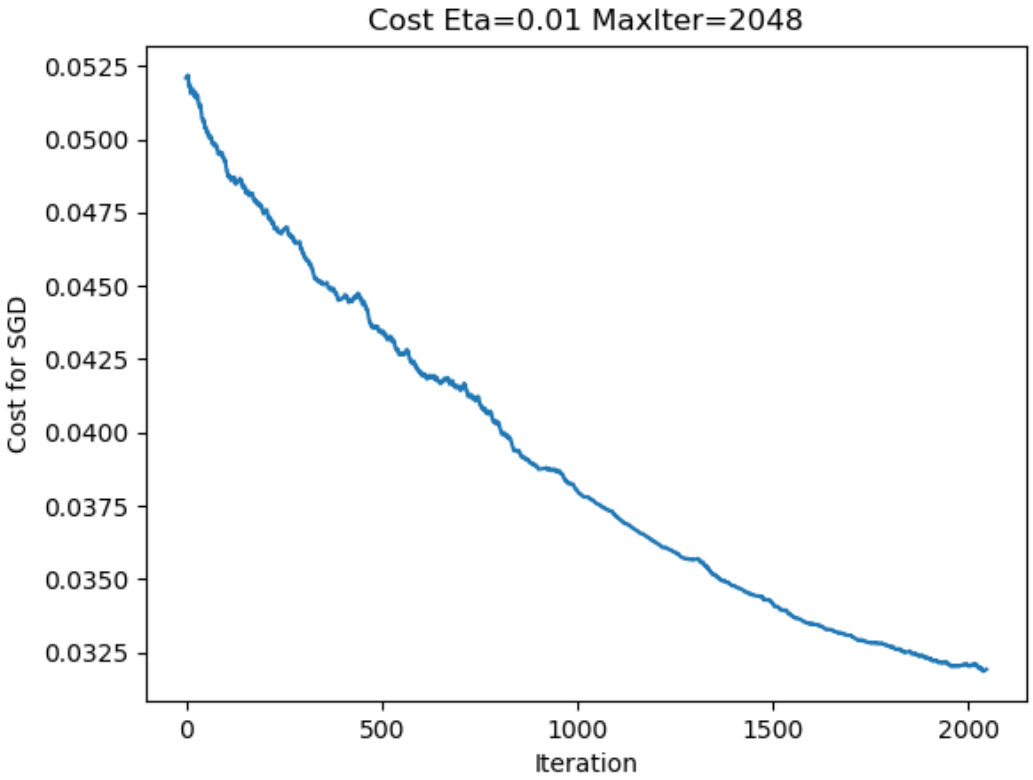


Figure 11: Plot of iterations vs cost for SGD

Problem Set 5

Problem 5(a)

Write code that implements the same neural network and the same stochastic gradient method with backpropagation step, as described in Section 6 of [Higham, Higham, 2019]. In other words, you can simply convert the Matlab code in Section 6 into your own language. Provide a print-out of your code in your report.

Print-out of code

```
#All credit to Catherine F. Higham Desmond J. Higham  in
#Deep Learning: An Introduction for Applied Mathematicians
import numpy as np
import matplotlib.pyplot as plt
import random

#Activation function
def activate(x,W,b):
    return 1/(1+np.exp((-1)*(W@x+b)))

#Uses backpropogation to train a network. This code is a transcription
#of the matlab code by Catherine F. Higham Desmond J. Higham
#Data
x1 = np.array([0.1,0.3,0.1,0.6,0.4,0.6,0.5,0.9,0.4,0.7])
x2 = np.array([0.1,0.4,0.5,0.9,0.2,0.3,0.6,0.2,0.4,0.6])
y = np.array([1.,1.,1.,1.,1.,0.,0.,0.,0.,0.,0.,0.,0.,0.,1.,1.,1.,1.,1.])
y=np.reshape(y,(2,10))

#Initialise weights and biases
np.random.seed(5000)
W2 = 0.5*np.random.rand(2,2)
W3 = 0.5*np.random.rand(3,2)
W4 = 0.5*np.random.rand(2,3)
b2 = 0.5*np.random.rand(2,1)
b3 = 0.5*np.random.rand(3,1)
b4 = 0.5*np.random.rand(2,1)

#Forward and back propogate
eta=0.05 #Learning rate
Niter=int(1e6) #Iterations
savecost=np.empty(Niter)

#Cost fucntion
def cost(W2,W3,W4,b2,b3,b4):
    costvec=np.zeros(10)
    for i in range(0,10):
        x=np.array([[x1[k]],[x2[k]]])

        #Forward pass
        a2= activate(x,W2,b2)
        a3= activate(a2,W3,b3)
        a4= activate(a3,W4,b4)
        costvec[i]=(1/10)*np.linalg.norm(np.reshape(y[:,k],(2,1))-a4)

    return np.linalg.norm(costvec)**2
```

```
for counter in range(0,Niter):
    k=np.random.randint(10) #Choose a random training point
    x=np.array([[x1[k]],[x2[k]]])

    #Forward pass
    a2= activate(x,W2,b2)
    a3= activate(a2,W3,b3)
    a4= activate(a3,W4,b4)
    #Backward pass
    delta4= a4*(1-a4)*(a4- np.reshape(y[:,k],(2,1)) )
    delta3= a3*(1-a3)*(np.transpose(W4)@delta4)
    delta2= a2*(1-a2)*(np.transpose(W3)@delta3)
    #Gradient step
    W2 = W2 - eta*delta2@np.transpose(x)
    W3 = W3 - eta*delta3@np.transpose(a2)
    W4 = W4 - eta*delta4@np.transpose(a3)
    b2 = b2 - eta*delta2
    b3 = b3 - eta*delta3
    b4 = b4 - eta*delta4
    #Monitor progress
    newcost = cost(W2,W3,W4,b2,b3,b4)
    savecost[counter]=newcost

    if counter % 500 == 0:
        #Display cost every 500 iterations
        print(newcost)

x_plot=range(0,Niter,int(1e4))
#Shows decay of cost
plt.plot(x_plot,savecost[x_plot],'b')
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.title('Cost for MATLAB Code')
plt.yscale("log")
plt.show()
```

Problem 5(b)

To help verify my code I have attempted a couple of things. Firstly I am plotting the cost over the course of training. This shows that the cost is decaying. Secondly I am inputting a test point to see if it is identified correctly.

The input parameters I am using are $\eta = 0.05$ and Niter=1e6. I am using the sigmoid activation function and the given cost function.

Below is a plot of the cost function for the code:

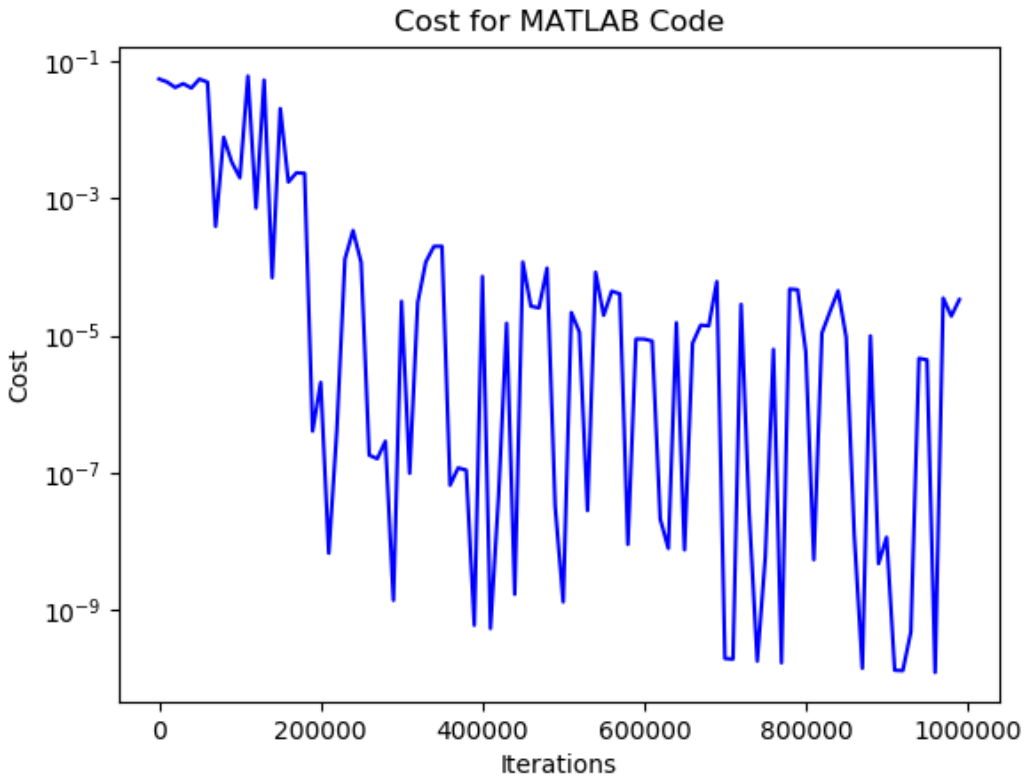


Figure 12: Plot of iterations vs cost for MATLAB Code

The cost value is erratic but it is certainly a lot smaller than it was at the start. Considering this is a graph with logarithm y scale, the cost function clearly decreases a lot even though it has some spikes.At the end even the highest peaks are much lower than the starting values and the absolute value of the cost is very low $\approx 1x10^{-}5$ so I believe it is clear that the cost is decaying and the optimal Weights and biases are being calculated. This verifies my code.

I used a test point to see if the neural network had trained accurately. I believe that is the cost shows clear decay and a test point is accurately classified then the code is verified!

The test point I have chosen is $(0.2, 0.2)$ the neural network should classify it as a circle. Numerically a circle is represented as $[1; 0]$.

I input the coordinates to the neural network. I propagated the data through the layers and the neural network predicted $[0.999941387; 5.91537456e - 05]$ this is very close the real values.

Because of this and the plot I think the code is verified.