KTH Royal Institute of Technology

SF2568
Parallel computations for Large-Scale problems

Year 2017 - 2018

Homework 2

Group report

Picó Oristrell Sandra 940531 – 2308 sandrapo@kth.se
Corominas Larsson Miquel Sven 920614 – 5998 miquell@kth.se

Professors :
Hanke Michael

March 2, 2018

Question 1: The broadcast operation is a one-to-all collective communication operation where one of the processes sends the same message to all other processes.

- Assume our simple communication model for point-to-point communication. A straightforward implementation would require *P*-1 communication steps. Design an algorithm for the broadcast operation using only point-to-point communications which requires only *O(log P)* communication steps. Hint: Recursive doubling
- Do a (time-)performance analysis for your algorithm.
- How can the scatter operation be implemented using *O(log P)* communication steps?

- **1 a):** As explained in the *Question 1* description, if you perform the broadcast operation sequentially, in a straightforward way, you will need to send *P-1* messages from the source node. However, as can be deduced, this is not the best way to do it. In order to achieve an efficient one-to-all communication, we are going to use Recursive Doubling.

  In that technique, the source process,that will send the message, will start sending the first message to another process. To continue, both process will send the message simultaneously to another two processes. Then, using this technique we are going to obtain a complexity of *O (log P)*.

  For example, in the next figure you can see how it works with a situation where the node 0 is the source node and we have P= 8.

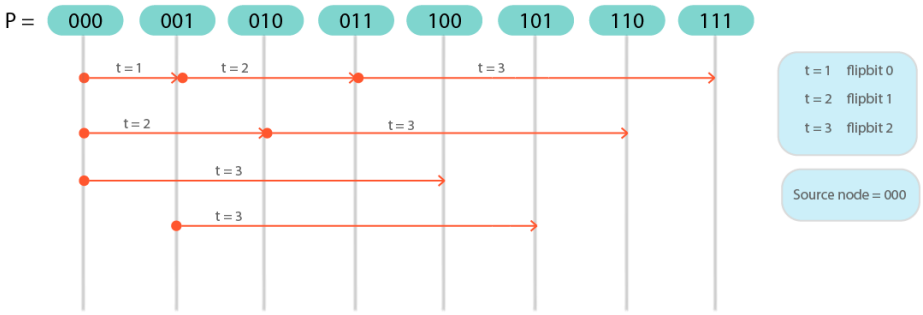

Figure 1: Recursive doubling example with P = 8 and source node = 0.

Then, based on all explained above we can start implementing the Pseudocode. However, in order to use the Recursive Doubling technique we will need to control which processes have to receive or send the message in any step of the algorithm.

To do so we will use a bit mask. The mask will be used to identify if the process has to send or receive the message. To do so, we are going to use the AND operation as implemented in the Pseudocode.

At the same time, it has to be clear that maybe the source node, representing the process that is sending the message, could not be the process number 0. Then, in that case, we need to implement a cyclic way to update the processes numbers:

Algorithm 1 Cyclic way to update the processes number

```
1: procedure CYCLIC(PROCESSES,SOURCE)
2:     Offset = Source
3:     for all Processes do
4:         Process N ← Process (N-Offset)
5:     return Processes
```

Then, using the previous implemented function, the algorithm will work in all the cases. The final implementation is as follows:

Algorithm 2 Broadcast operation using Recursive Doubling

```
1: procedure ONE-TO-ALL-COMMUNICATION()
2:     Create the bit mask with all 1.   ▷ The mask will have the same number of bits that represent the maximum number of processes.
3:     for all bits in the Mask do
4:         bitflip(Mask,bit)   ▷ Change the specific bit in the Mask to 0
5:         if Mask AND Process == 0 then
6:             if Process_bit == 0 then   ▷ If the bit of the process is a 0 then, receive.
7:                 receive(message)
8:             if Process _bit == 1 then   ▷ If the bit of the process is a 1 then, send.
9:                 send(message)
```

Then, as demonstrated in the previous Pseudocode, using Recursive Doubling you will be able to implement the Broadcast operation with a complexity of *O(log P)*

- **1b):**

  Regarding the time complexity of the algorithm, we will need to analyze two different times; the t_{comm} and the t_{comp} representing the time of communication and the time of computation respectively.

  This is because we know that the total time will be the sum of the time of communication and the time of computation.

  First we are going to analyze the communication time being a message of n different words.

$$t_{comm} = (t_{setup} + t_{word} * n_{words}) * number_{processes} \tag{1}$$

  To continue analyzing the computation time, we will need to analyze how many operations we are doing:

$$t_{comp} = (n_{operations} * t_{operation}) * number_{processes} \tag{2}$$

  The operations that we are doing in the pseudocode are the next ones:

  - Create the mask
  - bitflip * Number of bits
  - AND operation * Number of bits
  - Receive or send operation * Number of cases in R.Doubling

- **1c):**

  When we talked about the Scatter operation we need to know that this basically imply dividing the message into different parts and send each part to a different process. An image that represents this behaviour is the following:

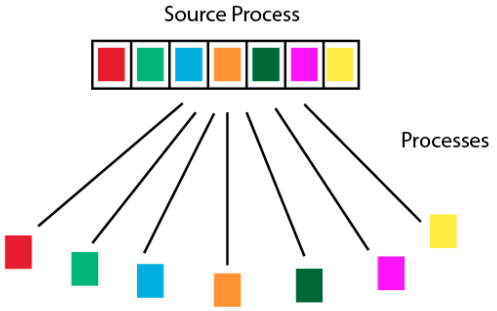

Figure 2: Scatter operation

Usually, when you are dividing the data with scatter distribution is because you do not care about the order of your message. However, in the case that you would like to distribute the data with scatter and also have an ordered message you will need to sort it. There are different ways to do that, but maybe a good one could be adding a positional identifier in the each part of the message.

However, the Pseudocode that we are attaching here do not take into account the order of the pieces of the message. This Pseudocode is also based in the Recursive Doubling algorithm. This is because is the only, or at least know by us, way to achieve a complexity of *O(log P)*. This implementation is also called *One to All Personalized Communication*.

This algorithm is based on the following rules:

- At each step, the processes will send half of their data.
- Consequentially, the size of the message will be smaller at each step.
- At the end, each process will a proportional piece of the initial data.

Algorithm 3 Scatter operation

```
1: procedure ONE-TO-ALL-COMMUNICATION-SCATTER()
2:     Create the bit mask with all 1.   ▷ The mask will have the same number of bits that represent the maximum number of processes.
3:     for all bits in the Mask do
4:         bitflip(Mask,bit)   ▷ Change the specific bit in the Mask to 0
5:         if Mask AND Process == 0 then
6:             if Process_bit == 0 then   ▷ If the bit of the process is a 0 then, receive.
7:                 receive(message/2)   ▷ They will receive half of the data
8:             if Process _bit == 1 then   ▷ If the bit of the process is a 1 then, send.
9:                 send(message/2)   ▷ They will have to send half of their data
```

Question 2: Consider a matrix A distributed on a P $\times$ P process mesh. An algorithm has been given in the lecture for evaluating the matrix-vector product $y = Ax$. While x is column distributed, y is row distributed. In order to carry out a further multiplication Ay, the vector y must be transposed.

- Design an algorithm for this transposition. You may use the results from problem 1.
- Make a performance analysis.
- An extra credit will be given for a good (!) solution in the case that A is distributed in a $P \times Q$ process mesh with $P \neq Q$.

In this Question we decide to only try to answer the two first questions. Then, we are not trying to obtain any extra point.

- **2 a)** To design the algorithm for the transposition problem we basically have to understand the main idea behind it. As we have a matrix distributed on a P x P process mesh we know that the transposition will be implemented following the next rules:
  - P = Q processes distribution.
  - If (p == q), not send or receive.
  - Else (p,q) $\longleftrightarrow$ (q,p)

  Then, the pseudocode is implemented as:

Algorithm 4 Transposition

```
1: procedure TRANSPOSITION(P,Q,DATA_PART)
2:     P = Q
3:                                   ▷ Suppose that we are the (p,q) process
4:     if q != p then
5:         rank = P*p + 2    ▷ Calculate the rank of the process that you need to exchange the information.
6:         SendReceive(data_part,rank,data_part) ▷ Change the information with the process (q,p) to update
   your data_part
```

- **2 b)**

  To continue we are going to make a performance analysis. As we did in the previous exercise, the time complexity will be analyzed through the communication time and the computation time.

$$t = t_{comm} + t_{comp} \tag{3}$$

Regarding the communication time we can define it as follows:

$$t_{comm} = (t_{setup} + t_{exchange\_information}) * (P * Q - P) \tag{4}$$

In the previous equation we basically compute the communication time for all the communications needed. As we only exchange information when *p* and *q* are different, we are going to compute (P*Q-P) communications.

Then, regarding the computation time, we need to take into account the number of operations. In our case we only need to compute the rank of the process and compute the SendReceive operation. Then, based on that, we are able to define the computation time as:

$$t_{comp} = (2 * t_{operation}) * number_{processes} \tag{5}$$

Question 3: We consider the problem of solving the 1-dimensional differential equation:

$$u'' + r(x)u = f(x), 0 < x < 1$$

$$u(0) = 0, u(1) = 0$$

Assume that $r(x) \leq 0$ for all $x \in (0, 1)$. The equation can be discretized as follows: For a give N > 0 let h $= 1/(N + 1)$ and $x_n = nh$, $0 \leq n \leq N + 1$. Then the discrete system reads:

$$\frac{1}{h^2}(u_{n-1} - 2u_n + u_{n+1}) + r(x_n)u_n = f(x_n), n = 1, ..., N$$

where $un \approx u(xn)$ and $u_0 = u_{N+1} = 0$. One possibility for solving this linear system of equation is Jacobi iteration. Let $u_n^{[0]}$ be a given guess of the solution. The sequence $u^{[k]}$ given by:

$$u_n^{[k+1]} = (u_{n-1}^{[k]} + u_{n+1}^{[k]} - h^2 f(x_n))/(2.0 - h^2 r(x_n))$$

converges (slowly!) towards the exact solution. Your task is to write a program which implements this algorithm using MPI. Test it out on a parallel computer. As a result, hand in a matlab plot of the solution and the error of a nontrivial problem of your choice, that is, *r(x)* should be a non-constant function. Additionally, hand in the source code of your program.

Use N = 1000. Since the speed of convergence is very slow, around 1000000 steps may be necessary.

To answer this problem, we basically start deciding which functions we would like to use. Then, following the restrictions specified in the description of the problem, we choose the next functions:

Regarding the *r(x)* function we need to be sure that $r(x) \leq 0$ for all $x \in (0, 1)$. Then, the *r(x)* chosen is:

$$r(x) = -10(x - 0.5)^2 \tag{6}$$

To continue, we define the *u(x)* function that verifies *u(0) = 0* and *u(1) = 0*. Our function is as follows:

$$u(x) = 40(x - 0.5)^2 - 10 \tag{7}$$

We are going to use the *u(x)* function described above to calculate the error towards the discrete approximation.

However, first of all we calculate the *f(x)* function that solves the 1-dimensional differential equation:

$$u(x)'' + r(x)u = f(x), 0 < x < 1 \tag{8}$$

The function *f(x)* that verifies that is the next one:

$$f(x) = 80 - 10(40(x - 0.5)^2 - 10)(x - 0.5)^2 \tag{9}$$

Using the functions specified above we can plot the real functions that verifies the differential equation:

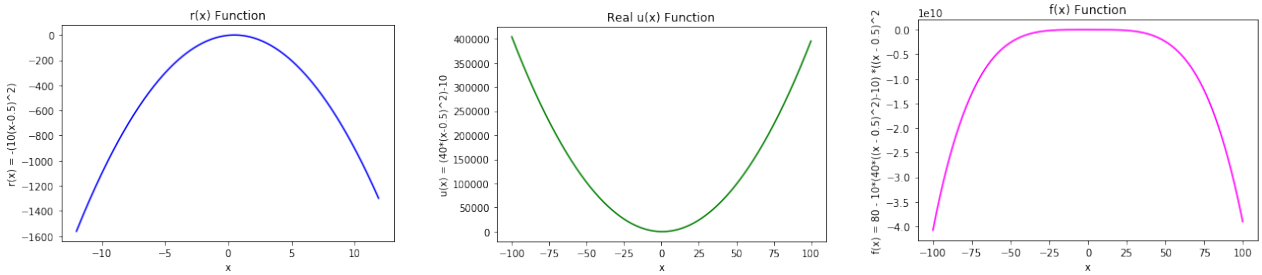

Figure 3: Functions that verifies the differential equation u(x)" + r(x)u(x) = f(x) , 0 < x < 1

Once we have the functions that verifies the differential equation we are able to start with exercise and apply the discrete equation. However, it is also needed to state that all the steps that we are going to describe and explain can be deducted through the implementation of the code *"C code Question 3"* found in the appendixes section.

First of all you need to initialize the MPI library. Then, we are going to define the data distribution. In this exercise, we decided to implement a linear data distribution with overlap, concretely with overlap = 2. The implementation of this data distribution is found in the lecture slides of the course.

Using the linear data distribution, each process will be able to know how many elements must compute and also which x have to use. The x are calculated through the following equation:

$$x_n = nh, 0 \leq n \leq N + 1 \tag{10}$$

Being h:

$$h = \frac{1}{N+1} \tag{11}$$

Then, at this point, each process will be able to compute its own values for the *real u(x)* function, r(x) function and f(x) function. For this reason it will start with the iterative process of calculating the Jacobi iteration as specified in the exercise description. As the convergence is slowly we use 1000000 iterations to achieve a good result.

To fill the ghost cells for the computation we use the red-black communication system. This system is used to be able to fill and send the ghost cells per each process.

Then, to plot all the results obtained we decided to create a file per each process. Each file will store the computed *u(x)* and the x_n value that corresponds. And finally, to plot the error achieved in each iteration, we also create a file. However, what we need to store is the total error computed for all the process per each iteration and, to achieve that, we use the MPI function called: MPI_Allreduce using the MPI_SUM operator. The computed error is the mean squared error.

Finally, here we attached the different plots that we achieve. The code implemented can be found in the appendixes section behind the subsection called *"Python code Question 3"*.

First, we decide to plot the u(x) function obtained for different iterations. As we are comparing with the real u(x) calculated previously, we can see that as much iterations better approximation we are obtaining.

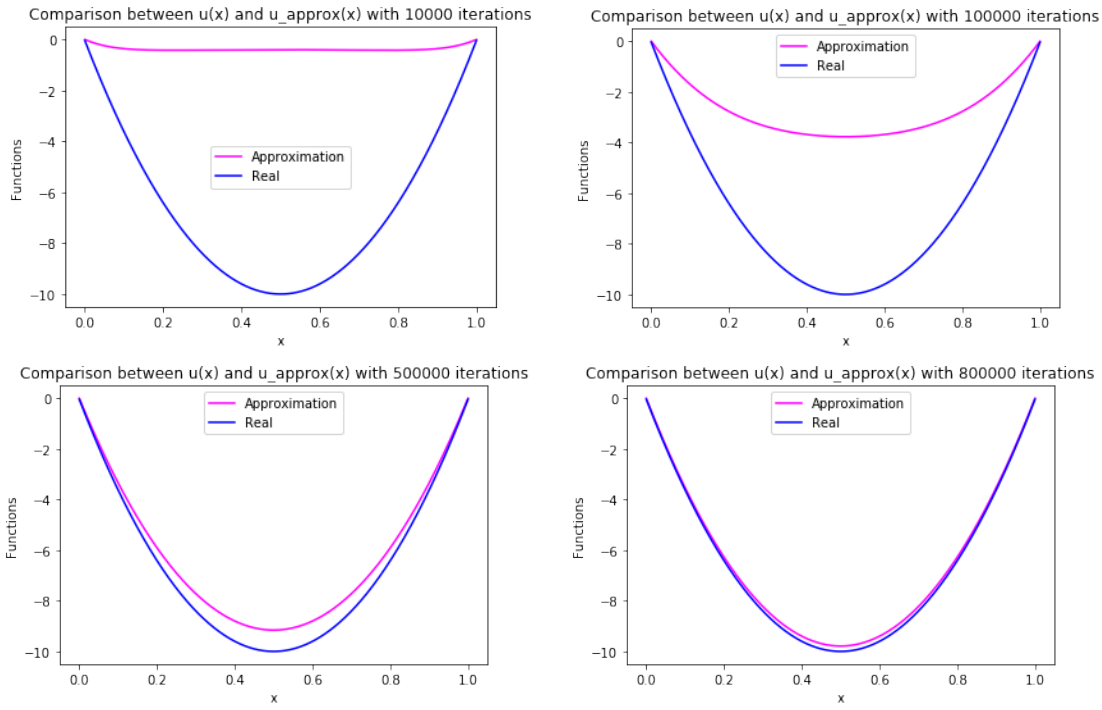

Figure 4: Comparison between the real u(x) and the approximated u(x) obtained per different iterations.

And, to finalize, we decided to plot the error obtained for all the different iterations. And, as it can be deducted through the figure attached above, the error decrements as the iterations increment.

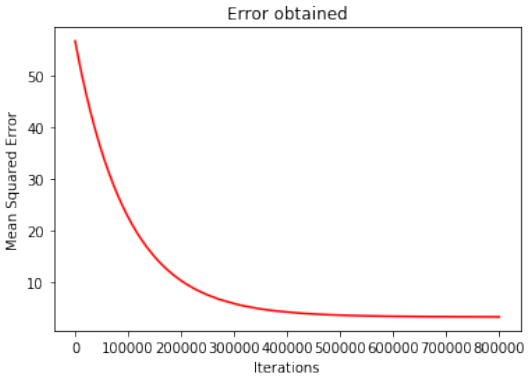

Figure 5: Mean Squared Error per different iterations.

Appendices

A C code Question 3

```
#include <stdio.h>
#include <stdlib.h>
#include <complex.h>
#include "math.h"
#include <mpi.h>
#include <string.h>

MPI_Status status;

#define N 1000
#define iterations 500000
#define h 1.0/(N+1)


//r(x) = (-(10(x-0.5)^2))
double r_function(double x){
    double result = -(10*(pow((x-0.5),2)));
    return result;
}

//Just to compare with the exact result (generate error)
//u(x) = (40(x-0.5)^2-10)
double u_function(double x){
    double result = (40*(pow((x-0.5),2))- 10);
    return result;
}


//f(x) = 80 - 10(40(x - 0.5)^2-10)(x - 0.5)^2
double f_function(double x){
    double result = 80 - 10*(40*(pow((x - 0.5),2))-10) *(pow((x - 0.5),2));
    return result;
}


// Main Function
int main (int argc, char **argv) {

    //To store the errors per iteration and then be able to plot it.
    double error_final[iterations];

    //Variables needed for the MPI implementation
    int tag = 10;
    int size = 0;
    int rank = 0;
    int rc = 0;

    //MPI Initialization
    rc = MPI_Init(NULL,NULL);
    rc = MPI_Comm_size(MPI_COMM_WORLD, &size);
    rc = MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    //The size variable define the number of processes.
    int P = size;

    //Linear data distribution with overlap (Lecture slides)
    int a = 2;
```

```
58      int M = N + (P-1)*a;
59      int L = M/P;
60      int R = M % P;
61      int Ip;
62
63      //Compute Ip ( How many numbers/elements will have each processor.)
64      //Lecture slides
65      if ( rank < R){
66          Ip = L+1;
67      }else{
68          Ip = L;
69      }
70
71      double x[Ip];
72      //Init value for each process. Depending of the p value.
73      int int_value = (L-a)*rank;
74      int aux = 0;
75      if (R ≥ rank){
76          aux = rank;
77      }else{
78          aux = R;
79      }
80      int_value = int_value + aux;
81
82      //Compute the x_n values.
83      for (int i = 0; i<Ip; i++ ){
84          x[i] = h*(int_value+i+1);
85      }
86
87      //Compute the f(xn) i r(xn) per each value.
88      double fx[Ip];
89      double rx[Ip];
90      double u_exact[Ip];
91
92      //Compute the three functions needed per all values of x (per each process)
93      for (int i = 0; i < Ip; i++){
94          fx[i] = f_function(x[i]);
95          rx[i] = r_function(x[i]);
96          u_exact[i] = u_function(x[i]);
97      }
98
99      double u[Ip];
100     double u_error[Ip];
101     double new_u[Ip];
102
103     //Init
104     for (int i = 0; i < Ip; i++){
105         u[i] = 0;
106         u_error[i] = 0;
107         new_u[i] = 0;
108     }
109
110     //Going through all the iterations.
111     //The convergence is slowly so you need to do a lot of iterations.
112     for (int iter = 0; iter <iterations;iter ++){
113
114         //Red-black communication to fill the ghost cells.
115         if((rank % 2 ) == 0){
116             //To fill the ghost cells you will need to identify the first process and ...
                    the last one
117             //(only send/fill the specific ghost cells per each case)
118             //Even
119             if (rank != (size -1)){
120                 MPI_Send(&u[Ip-2],1, MPI_DOUBLE, rank+1,tag, MPI_COMM_WORLD);
121                 MPI_Recv(&u[Ip-1],1, MPI_DOUBLE, rank+1,tag, MPI_COMM_WORLD,&status);
```

```
122             }
123             if (rank != 0){
124                 MPI_Send(&u[1],1, MPI_DOUBLE, rank-1,tag, MPI_COMM_WORLD);
125                 MPI_Recv(&u[0],1, MPI_DOUBLE, rank-1,tag, MPI_COMM_WORLD,&status);
126             }
127         }else{
128             //Odd
129             if (rank != 0){
130                 MPI_Recv(&u[0],1, MPI_DOUBLE, rank-1,tag, MPI_COMM_WORLD,&status);
131                 MPI_Send(&u[1],1, MPI_DOUBLE, rank-1,tag, MPI_COMM_WORLD);
132             }
133             if (rank != (size-1)){
134                 MPI_Recv(&u[Ip-1],1, MPI_DOUBLE, rank+1,tag, MPI_COMM_WORLD,&status);
135                 MPI_Send(&u[Ip-2],1, MPI_DOUBLE, rank+1,tag, MPI_COMM_WORLD);
136             }
137         }
138 
139         //Compute u(x) but not filling the ghost cells.
140         int init = 1;
141         for (int r = init; r ≤ Ip -2 ; r++){
142             //The sequence u[k] is given by...
143             new_u[r] = (u[r-1]+u[r+1]-(pow(h,2))*(fx[r]))/(2.0-((pow(h,2))*rx[r]));
144         }
145 
146         //Update u
147         for (int i = 0; i < Ip; i++){
148             u[i] = new_u[i];
149         }
150 
151         //Compute error comparing with the exact function.
152         for (int i = 0; i < Ip; i++){
153             u_error[i] = pow(u[i] - u_exact[i],2);
154         }
155 
156         //Compute the sum of the u_error:
157         double sum_error = 0;
158         double globalsum = 0;
159         for ( int i = 0; i < Ip; i++){
160             sum_error = sum_error + u_error[i];
161         }
162 
163         //Now we need to sum all the errors per all the processes.
164         MPI_Allreduce(&sum_error,&globalsum,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
165         //Now, global sum has all the errors regarding/iteration.
166         //This is important to be able to plot the error.
167         error_final[iter] = globalsum/N;
168     }
169 
170     //Now we have all the errors. We are going to store the errors in a file and then ...
        plot them.
171     FILE *error_file;
172     error_file = fopen("error.txt","wb");
173     for(int i = 0; i < iterations; i++){
174         fprintf(error_file,"%.5f\n",error_final[i]);
175     }
176     fclose(error_file);
177 
178     //After all the iterations we are going to create a file per each process.
179     //The file will store the u values, and then we are going to print them through an ...
        external python code.
180     FILE *fptr;
181     char name[50];
182     //The name of the file will have the name of the process.
183     sprintf(name,"%d",rank);
184     char direction[4] = ".txt";
```

```
    strcat(name,direction);
    printf("Name: %s\n",name);
    fptr = fopen(name,"wb");

    //Store the u(x) per different x cases.
    if (rank == 0){
        for(int i = 0; i<(Ip-1); i++){
            fprintf(fptr,"%.5f",u[i]);
            fprintf(fptr,",%.5f\n",x[i]);
        }
    }else{
        //Take into account the ghost cells
        if(rank == size-1){
            for(int i = 1; i<Ip; i++){
                fprintf(fptr,"%.5f",u[i]);
                fprintf(fptr,",%.5f\n",x[i]);
            }
        }else{
            for (int i = 1; i< Ip-1; i++){
                fprintf(fptr,"%.5f",u[i]);
                fprintf(fptr,",%.5f\n",x[i]);
            }
        }
    }
    //Close the file
    fclose(fptr);
    //Close the MPI library/communication.
    MPI_Finalize();
}
```

B Python code Question 3

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import animation
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def r_function(x):
    return (-(10*(x-0.5)**2))

def u_function(x):
    return (40*(x-0.5)**2-10)

def f_function(x):
    return 80 - 10*(40*(x - 0.5)**2-10)*(x - 0.5)**2


def Plot_rx():
    #Plot the real r(x)
    x = np.arange(-12,12,0.1)
    r = []
    for i in range(0,(x.size)):
        r.append(r_function(x[i]))

    plt.title("r(x) Function")
    plt.plot(x, r,'blue')
    plt.ylabel('r(x) = -(10(x-0.5)^2)')
    plt.xlabel('x')
    plt.show()

```

```
def Plot_ux_real():
    #Plot the real u(x)
    x = np.arange(-100,100,0.1)
    u = []
    for i in range(0,(x.size)):
        u.append(u_function(x[i]))
    plt.title("Real u(x) Function")
    plt.plot(x, u,'green')
    plt.ylabel('u(x) = (40*(x-0.5)^2)-10')
    plt.xlabel('x')
    plt.show()


def Plot_fx():
    #Plot the real f(x)
    x = np.arange(-100,100,0.1)
    f = []
    for i in range(0,(x.size)):
        f.append(f_function(x[i]))
    plt.title("f(x) Function")
    plt.plot(x,f,'magenta')
    plt.ylabel('f(x) = 80 - 10*(40*((x - 0.5)^2)-10) *((x - 0.5)^2')
    plt.xlabel('x')
    plt.show()


def Plot_ux_uxreal():
    #We are going to plot the approximation:
    #Compare u(x) with u_approx(x):
    #1) Compute the u_approx(x):
    x = []
    u_approx = []
    for i in range(0,30):
        name = str(i)+".txt"
        file = open(name,"r")
        for line in file:
            u_aux, x_aux = line.split(",")
            u_approx.append(u_aux)
            x.append(x_aux)

    #Compute the real u(x) using the same x(n).
    u = []
    x_aux = []
    for element in x:
        e = float(element)
        u.append(u_function(e))

    plt.title("Comparison between u(x) and u_approx(x)")
    plt.plot(x,u_approx,'magenta')
    plt.plot(x,u,'blue')
    plt.ylabel('Functions')
    plt.xlabel('x')
    plt.show()


def Plot_error():
    #Compute the mean error regarding the different iterations.
    iterations = np.arange(500000)
    file = open("error.txt","r")
    errors = []
    for line in file:
        errors.append(line)

    plt.title("Error obtained")
    plt.plot(iterations,errors,'red')
```

```
95      plt.ylabel('Mean Squared Error')
96      plt.xlabel('Iterations')
97      plt.show()
98
99  def main():
100     Plot_rx()
101     Plot_fx()
102     Plot_ux_real()
103     Plot_ux_uxreal()
104     Plot_error()
105
106
107 if __name__ == "__main__":
108     main()
```