Solution to Newtonian Gravity problem with many bodies
Computational Physics-Phy905
Project 3

Crispin Contreras

April 15, 2016

Abstract

This paper discusses the numerical solution to Newtonian Gravity with different planets in our solar system. Two methods were used to solve the problem one was the Verlet method and the other one was the Runge-Kutta to the fourth order (RK4). From these two methods I found that the RK4 was more accurate and also when calculating the conservation of energy it did a better job as was expected.

1 Introduction

I start by solving the the equations for Newtonian gravity. I begin with the simple Sun-Earth system and then move on to solve the Sun-Earth-Jupiter system with the Sun fixed and then the Sun-Earth-Jupiter with the Sun not fixed. Finally I model all the planets including Pluto. I used two different methods to solve this problem which are the Verlet and Runge-Kutta to the fourth order (RK4). This are just Taylor expansions and I discuss more about them in the Methods section. I test to see how stable the systems are by looking at different time steps and also increasing the time that the planets obit. Finally include my results and the conclusions that I arrived to.

2 Theory

2.1 Earth-Sun System

I start with the Newtonian gravity which is given by $\mathbf{F} = -\frac{GM_1M_2}{r^2}\hat{\mathbf{r}}$. Here G is the gravitational constant, r is the distance between the bodies, and M represents the mass. I decided to work in Cartesian coordinates so I use

$$\begin{aligned} \hat{\mathbf{r}} &= cos(\theta)\hat{\mathbf{x}} + sin(\theta)\hat{\mathbf{y}} \\ x = rcos(\theta), \quad y = rsin(\theta) \quad r &= \sqrt{x^2+y^2} \end{aligned} \tag{1}$$

Where θ is just the polar angle. Now using equation (1) I break the equation for gravitation into Cartesian coordinates

$$\begin{aligned} F_x &= -\frac{GM_1M_2}{r^3}x \\ F_y &= -\frac{GM_1M_2}{r^3}y \end{aligned} \tag{2}$$

With equation (2) now I can start setting up the equations of motions for the planets. I start with the Earth-Sun system where I treat the Sun as stationary so I use it as the origin. I start with the equations of motion, for simplicity I will only do the x coordinate since the only thing that changes in the others is the coordinate itself.

$$M_{Earth}\frac{d^2x}{dt^2} = -\frac{GM_{Earth}M_{Sun}}{r^3}x$$

I will simplify this equation further by introducing a new set of units for the G and mass. The Earth's orbit around the sun is almost circular around the Sun so for this type of motion the force is given by

$$F = \frac{M_{Earth}v^2}{r} = \frac{GM_{Sun}M_{Earth}}{r^2}$$

from this equation we can get rid of the gravitational constant G and mass of the sun to replace it by

$$v^2 r = GM_{Sun} = 4\pi^2(\frac{AU^3}{yr^2}) \tag{3}$$

with this transformation now we can use the astronomical units (AU) for length and year (yr) for time. The mass of the Sun is then one. Now the equation of motion for the earth reads

$$\begin{aligned} F_x = \frac{d^2v_x}{dt^2} &= -\frac{4\pi^2}{r^3}x \\ \frac{dx}{dt} &= v_x \end{aligned} \tag{4}$$

The equation for the y coordinate is the same except the x is replace by y.

2.2 Three Body Problem

I start with a simplified version of the three body problem. In this case I model the Sun, Earth, and Jupiter but I will keep the Sun fixed. For the Earth the only thing that changes now is that it feel the force from Jupiter. The force now reads

$$F_x^{Earth} = \frac{dv_{x_E}}{dt} = -\frac{4\pi^2}{r_{ES}^3}x_E - \frac{4\pi^2(M_{Jupiter}/M_{Sun})}{r_{EJ}^3}(x_E - x_J) \tag{5}$$

where now include the coordinates for Jupiter, r_{EJ} is the distance between the Earth and Jupiter, and r_{ES} is the distance between the Earth and the Sun. Here x_J is the position of Jupiter and x_E is the position of the Earth and M represents the mass of the bodies. The same equation for (5) is obtained for y except the x is replace by either y. r_{EJ} is given by

$$r_{EJ} = \sqrt{(x_E - x_J)^2 + (y_E - y_J)^2 +}$$

For Jupiter I obtain the equation

$$F_x^{Jupiter} = \frac{dv_{x_J}}{dt} = -\frac{4\pi^2}{r_{JS}^3}x_J - \frac{4\pi^2(M_{Earth}/M_{Sun})}{r_{EJ}^3}(x_J - x_E) \tag{6}$$

here r_{JS} is the distance between Jupiter and the Sun, in order to get the y coordinate I just replace x by y.

Now to do the full three body problem I will allow the Sun to move instead of being fixed. This means that the origin of the system is now the center of mass of the three bodies. I give the equation for the Sun

$$F_x^{Sun} = \frac{dv_{x_S}}{dt} = -\frac{4\pi^2(M_{Jupiter}/M_{Sun})}{r_{JS}^3}(x_S - x_J) - \frac{4\pi^2(M_{Earth}/M_{Sun})}{r_{SE}^3}(x_S - x_E) \tag{7}$$

for equations (5) and (6) the only thing that gets modified is the term x_J which goes to $(x_J - x_S)$ and x_E which goes to $(x_E - x_S)$, the same thing occurs for the y variable. With equations (5),(6),and (7) now I can solve the three body problem in two dimensions.

Finally to model the all the planets of the solar system and Pluto, I keep the sun fixed for simplicity. In general the equation is given by

$$F_x^j = -\frac{4\pi^2}{r_{jS}^3}(x_j - x_S) - 4\pi^2 \sum_{i=1, j\neq i}^{9} \frac{M_i/M_{Sun}}{r_{ji}^3}(x_j - x_i) \tag{8}$$

here j and i can take on the values $j, i = 1, 2, 3...9$ and they represent how far the planet is from the Sun. For example Mercury will be 1, Venus 2, and so on. Now that I have all the equations setup I can move on to describe the methods.

$$\begin{pmatrix} d_1 & e_1 & 0 & \dots & \dots & \dots & 0 \\ e_1 & d_2 & e_1 & 0 & \dots & \dots & 0 \\ 0 & e_1 & d_3 & e_1 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & e_1 & d_{n-1} & e_1 \\ 0 & \dots & \dots & \dots & 0 & e_1 & d_n \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ \vdots \\ \vdots \\ u_n \end{pmatrix} = \lambda \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ \vdots \\ \vdots \\ u_n \end{pmatrix} \tag{9}$$

3 Methods

Two methods were implemented two solve the Schrödinger equation. One was the Jacobi method which consists of many similarity transformations and the other one was using Armadillo's eigenvalue solver which is called by $eig - sys$. I will discuss the Jacobi method here. I take much of the derivations from [1].

3.1 Jacobi Method

The Jacobi method consists of using many similarity transformation to reduce a matrix into diagonal form. In this case the matrix is $\mathbf{A}$. This method is chosen since doing a determinant for large matrices is impractical. The similarity transformation is defined by

$$\mathbf{B} = \mathbf{S^T A S}$$

The $\mathbf{S}$ has the following property $\mathbf{S^T} = \mathbf{S^{-1}}$. In our case the similarity transformation is defined by

$$s_{kk} = s_{ll} = cos(\theta), \quad s_{kl} = -s_{lk} = -sin(\theta), \quad s_{ii} = 1, \quad i \neq l \quad i \neq k$$

the other terms are zero. The results for the $\mathbf{B}$ matrix are

$$\begin{aligned} b_{ik} &= a_{ik}cos(\theta) - a_{il}sin(\theta), \quad i \neq k, \quad i \neq l \\ b_{il} &= a_{il}cos(\theta) + a_{ik}sin(\theta), \quad i \neq k, \quad i \neq l \\ b_{kk} &= a_{kk}cos^2(\theta) - 2a_{kl}cos(\theta)sin(\theta) + a_{ll}sin^2(\theta), \\ b_{ll} &= a_{ll}cos^2(\theta) + 2a_{kl}cos(\theta)sin(\theta) + a_{kk}sin^2(\theta), \\ b_{kl} &= (a_{kk} - a_{ll})cos(\theta)sin(\theta) + a_{kl}(cos^2(\theta) - sin^2(\theta)) \end{aligned} \tag{10}$$

The recipe is the to chose θ so that all non-diagonal elements b_{kl} become zero. We require that $b_{kl} = b_{lk} = 0$ which then leads to

$$b_{kl} = (a_{kk} - a_{ll})cos(\theta)sin(\theta) + a_{kl}(cos^2(\theta) - sin^2(\theta)) = 0$$

if $a_{kl} = 0$ then this leads to $cos(\theta) = 1$ and $sin(\theta) = 0$. To solve the equation above we define $tan(\theta) = t = \frac{s}{c}$, $sin(\theta) = s$, $cos(\theta) = c$, and use the trigonometric identity $cos(2\theta) = \frac{1}{2(cot(\theta) - tan(\theta))}$ to obtain

$$cos(2\theta) = \tau = \frac{a_{ll} - a_{kk}}{2a_{kl}}$$

using the trigonometric identity we obtain

$$t^2 + 2\tau t - 1 = 0$$

resulting in

$$t = -\tau \pm \sqrt{1 + \tau^2}$$

and we can obtain c and s by using trigonometric tricks

$$c = \frac{1}{\sqrt{1 + t^2}}$$

and $s = tc$. In order to get the off-diagonal terms to be equal to zero or in our case $\geq 10^{-8}$ we have to complete many similarity transformations until this occurs. In order to figure this out we look at the Frobenius norm which is defined as

$$||\mathbf{A}||_F = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n}|a_{ij}|^2}$$

One of the important properties of the Frobenius norm is that it's the same after similarity transformations. This property will be used in the Jacobi algorithm. Solving the quadratic equation we find that $|\theta| \leq \frac{\pi}{4}$ having this property then leads to a minimization of the difference between $\mathbf{A}$ and $\mathbf{B}$ since

$$||\mathbf{B} - \mathbf{A}||_F^2 = 4(1 - c)\sum_{i=1, i\neq k,l}^{n}(a_{ik}^2 + a_{il}^2) + \frac{2a_{kl}^2}{c^2}.$$

To summarize, the Jacobi methods consists of many similarity transformations in order to decrease the value of the off diagonal elements. This can be done by looking at the norm. This algorithm was implemented in C++ and it's show below in listing 1.

```
//Implementing Jacobi Method
double  max = fabs(A(0,1));
int l; //indices
int k;

clock_t start, finish;
start = clock();

while(max > epsilon)
{
  max =0.0;
        for (int i = 0 ; i< (n-2) ; i++)
        {
                    for(int j =i+1 ; j<(n-1); j++)
                    {
                        if (fabs(A(i, j)) > max)
                        {
      max =fabs(A(i,j));

     //Find the indices
                            k=i;
                            l =j;
                        }
                     }
                }//End of i loop

   //Call function to rotate
   rotate(A,R,k,l,n);

   iterations++;
}
```

Listing 1: This shows how the Jacobi method was implemented. The rotate function is called here and it's responsible for carrying out the operation in (6).

4 Results

4.1 Single Electron

The Jacobi algorithm was implemented in C++ according to section 3. This was then compared to the Armadillo's eigenvalue solver $eig - sys()$ to make sure it was working properly. The energy eigenvalues are know and were discussed in the theory sections. Their values 3,7,and 11 were used for calibration for getting the ρ_{max}. By trial and error we found that the best value for the three lowest states was $\rho_{max} = 5$. The results for the eigenvalues and computational times are shown in table 1 below. From table 1 we can see that as the value of n increases the eigenvalues get closer to the correct values. Table 1 also shows the number of iterations that the Jacobi method must make in order for the off-diagonal terms to be $\leq 10^{-8}$. From table 1 we can also estimate the number of similarity transformations in order to get the off-diagonal terms to 10^{-8} which is roughly $1.7n^2$. Finally we made a plot using matplolib for the corresponding three lowest state which is shown in figure 1. In this case we made sure the wavefunctions were normalized as $\int_0^\infty |u(\rho)|^2 d\rho$. This was done by approximating the integral as a set of rectangles (like a histogram) and using the values of the normalized

eigenfunctions as the height and the width from h we can calculate the area.The the integral is just

$$\int_0^\infty |u(\rho)|^2 d\rho \approx h \left(\sum_{j=1}^{n} |u_j|^2 \right)$$

From this then we can get the normalized eigenfunctions by multiplying by $\frac{1}{h}$. This same procedure is done for the two electron case.

Figure 1: Plot of the radial wavefunctions for the three lowest energies. The eigefunctions were normalized and we used $\rho_{max} = 5$ and n=400.

Grid Points, n	3 Lowest Energy Eigenvalues For Jacobi, λ	Calculation Time, T_{Ja} [s]	Calculation Time, T_{arma} [s]	Number of iterations
50	2.9969, 6.9850, 10.9634	0.01653	0.001274	4040
100	2.9992, 6.9962, 10.9908	0.22181	0.003996	16475
200	2.9998, 6.9990, 10,9978	3.67701	0.021410	66828
300	2.9999, 6.9996, 10.9991	17.7895	0.069452	150798
350	2.9999, 6.9997, 10.9994	34.5932	0.083118	205757
400	3.0000, 6.9998, 10.9996	57.7010	0.122340	269021

Table 1: This table shows the number of grid points, the eigenvalues, Computational time, and the number of iterations. From here we can interpolate the number of similarity transformations needed to reach 10^{-8}.

4.2 Two Interacting Electrons

We use the same code for the single electron and modify it to include the Coulomb interaction. To see if the values are correct we use [2] and the values given by this source. This is shown in table 2.

Figure 2: This figure show the normalized eigenfunction for ω_r =0.01 and ω_r=0.5. The values for ρ_{max} are 60 and 8 repectively witn $n = 400$.

Figure 3: This figure show the normalized eigenfunction for ω_r =1 and ω_r=5. The values for ρ_{max} are 8 and 2 repectively witn $n = 400$.

Oscillator Frequency ω_r	Ground State Energy Eigenvalues, λ	[2] Approximate Formula, 2 ϵ'	[2]improved formula, 2 $\epsilon'_i nt$
0.01	0.1058	0.1050	–
0.05	0.3500	0.3431	0.3500
0.25	1.2500	1.1830	1.2500
0.5	2.2300	2.0566	–
1	4.0578	3.6219	–
5	17.4485	14.1863	–

Table 2: Values for the frequency ω_r and the energy eigenvalues. Also shown are the results from [2] which are the accepted values for the energies. We used these to make sure the algorithm was working correctly.

5 Discussion

The implementation of the Jacobi algorithm produced good results as shown in table 1 since they have good agreement with the know energy values. From table 1 we obtain that approximately $1.7n^2$ similarity transformations are needed to get the off-diagonal terms close to zero. This is less since our matrix is tridiagonal, for a full matrix a total of $3n^2$ to $5n^2$ transformations are needed[1]. We also saw that the Jacobi method has a very large computational time as opposed to the Armadillo's eigenvalue solver.

In trying to get reasonable results the biggest problem with this algorithm was determining the value of n and ρ_{max}. This took some time since it was done by trial and error. Although when I figured out one of the values it was easier to determine the others by changing ρ_{max} a bit. In order to start analyzing the interacting case we needed to use [2] to make sure the code was working correctly. For this I used the values for ω_r=0.25 and 0.05 and compared to [2]. As shown in table 2 our results show that our program was working correctly. In addition to this I also checked that the norm of the lowest eigenvectors were equal to 1. This is because the Frobenius norm is no affected by transformations. Again my results shows that this was true. These were the "unit test" that I implemented to make sure my code was working correctly.

Finally I discuss the physics of plots. I use figure 1 as basis and compared to the other plots. The main result is that the eigenfunctions were stretch or squeezed depending on the size of ω_r. For small values of ω_r we can see that eigenfunctions get stretched in the horizontal direction while for large values they get squeezed. We see that the electrons are further apart from each other when we include the interaction which is what is expected.

6 Conclusion

The Jacobi algorithm was implemented and we found that it had a very large computational time compared to Armadillo's eigenvalue solver. Additionally it was a bit time consuming figuring out what the best value for ρ_{max} was and similarly for n. The only way to do this was by trial and error. For the interacting case we found that the electrons get more spread out and this is what is expected since they have the same charges. The results for the interacting case were in agreement with [2] since the eigenvalues agreed.

7 References

[1] M. Hjorth-Jensen. Computational Physics, Lecture Notes Spring 2016.
[2] M. Taut.Two electrons in an external oscillator potential: Particular analytical solutions of a coulomb correlation problem. Phys. Rev. A. 48, 3561-3566 (1993)

8 Code Attachment

```
/*
  Author: Crispin Contreras
  Class: Physics 905
  Purpose: Solves the 3D radial Schrodinger equation in a harmonic potetial
  with two electrons. One way is using the Jacobi method taking into account
  the interaction between electrons and the other using the Armadillo library.

*/

#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>
#include <stdio.h>
#include "armadillo"

using namespace arma;
using namespace std;
ofstream ofile;


void rotate(mat& , mat& , int , int , int);

int main()
{
  int n; //number of steps
  int iterations=0;
  int inter;
  char outfile[30]; //Rho and eigenfunctions
  char Eigen[30]; // Eigenvalues, elapsed time, iterations
  double epsilon = 1.0e-8;
  double rho_max, rho_min, w_r;
  double *rho_i, *V_Pot, *V_Pin;
  double h;

  //Read Input
  cout<<"Enter the number of steps n: ";
  cin>>n;
  cout<<"Enter the value of rho_max: ";
  cin>>rho_max;
  cout<<"Enter the value of rho_min: ";
  cin>>rho_min;
  cout<<"Do you want non-interacting(0) or interacting (1): ";
  cin>>inter;
  cout<<"Enter the value of the frequency w_r: ";
  cin>>w_r;
  cout<<"Enter the name of the outputfile for Rho and Eigenfunctions: ";
```

```
    cin>>outfile;
49  cout<<"Enter the name of file for Eigenvalues, elapse time, and total number
      of iterations: ";
    cin>>Eigen;
51


53  //calculate the step size
    h = (rho_max - rho_min)/n;
55
    //Allocate Memory
57  rho_i = new double [n+1];
    V_Pot = new double [n-1];
59  V_Pin = new double [n-1];

61  //Fillout value of potential
    for(int i=0; i<=n; i++)
63  {
       rho_i[i] = rho_min + i*h;
65
    }
67
    for (int i=0; i< (n-1); i++)
69  {
        //Non-Interacting Potential
71      V_Pot[i]=w_r*w_r*rho_i[i+1]*rho_i[i+1];

73      //Interacting Potential
        V_Pin[i] = V_Pot[i] + (1/rho_i[i+1]);
75
    }
77


79  //Declare variables
    mat A(n-1,n-1);
81  mat R(n-1,n-1);

83  //Filling the matrices
    A.zeros(n-1, n-1);
85  R.eye(n-1,n-1);

87  for (int i=0 ; i<(n-1); i++)
    {
89
        if(inter == 0)
91      {
           A(i, i) = 2/(h*h) + V_Pot[i];
93      }
        if(inter == 1)
95      {
  A(i,i) = 2/(h*h) + V_Pin[i];
97
        }
99
        if( i<(n-2))
101     {
  A(i,i+1) = A(i+1, i)= -1/(h*h);
103
```

```
        }
105
    }
107

109 //Copy for armadillo
    mat C = A;
111

113 //Implementing Jacobi Method
    double  max = fabs(A(0,1));
115 int l; //indices
    int k;
117
    clock_t start, finish;
119 start = clock();

121 while(max > epsilon)
    {
123   max =0.0;
          for (int i = 0 ; i< (n-2) ; i++)
125       {
                  for(int j =i+1 ; j<(n-1); j++)
127               {
                     if (fabs(A(i, j)) > max)
129                  {
       max =fabs(A(i,j));
131
      //Find the indices
133                      k=i;
                         l =j;
135                  }
                  }
137             }//End of i loop

139   //Call function to rotate
      rotate(A,R,k,l,n);
141
      iterations++;
143 }

145 finish=clock();
    double time_j = (double (finish-start)/CLOCKS_PER_SEC);
147
    //Assorting the eigevalues and eigenvectors
149 vec Max(n-1);
    int Loc[3]={0,1,2};
151 double temp;

153 for(int i=0; i<(n-1);i++)
    {
155    Max(i)=A(i,i);
    }
157
    for(int i=0; i<(n-1);i++)
159 {
       for(int j=i+1; j<(n-1);j++)
```

```
161         {
                if(Max(i)>Max(j))
163             {
            temp=Max(i);
165         Max(i)=Max(j);
            Max(j)=temp;
167         if(i<3)
            {
169           Loc[i]=j;
            }
171             }

173         }

175     }

177     //Solving using Armadillo
              vec eigval(n-1);
179           mat eigvec(n-1,n-1);

181           start = clock();
              eig_sym(eigval, eigvec, C);
183           finish = clock();

185           double time_ar =(double(finish-start)/CLOCKS_PER_SEC);

187     //three lowest states Armadillo
        vec V0 = eigvec.col(0);
189     vec V1 = eigvec.col(1);
        vec V2 = eigvec.col(2);
191     //From Jacobi
        vec R0 = R.col(Loc[0]);
193     vec R1 = R.col(Loc[1]);
        vec R2 = R.col(Loc[2]);
195
        //Unit Test, the norm should be equal to one
197     double V0Sum=0.0, V1Sum=0.0, V2Sum=0.0;
        double R0Sum=0.0, R1Sum=0.0, R2Sum=0.0;
199
        for(int i=0; i<(n-1); i++)
201     {
            V0Sum+=V0(i)*V0(i);
203        V1Sum+=V1(i)*V1(i);
           V2Sum+=V2(i)*V2(i);
205        R0Sum+=R0(i)*R0(i);
           R1Sum+=R1(i)*R1(i);
207        R2Sum+=R2(i)*R2(i);

209     }

211     cout<<"Norm of Ground (Ar) "<<V0Sum<<endl;
              cout<<"Norm of 1st (AR) "<<V1Sum<<endl;
213           cout<<"Norm of 2nd (AR) "<<V2Sum<<endl;
              cout<<"Norm of Ground (Ja) "<<R0Sum<<endl;
215           cout<<"Norm of 1st (Ja) "<<R1Sum<<endl;
              cout<<"Norm of 2nd (Ja) "<<R2Sum<<endl;
217
```

```
219   //Writing to file , Rho, EigenFunctions
      ofile.open(outfile);
221   ofile<<setiosflags(ios::showpoint | ios::uppercase);
      ofile<<"#Rho"<<setw(23)<<"Ground(Ar)";
223   ofile<<setw(20)<<"1st(Ar)"<<setw(20)<<"2nd(Ar)";
      ofile<<setw(20)<<"Ground(Ja)"<<setw(20)<<"1st(Ja)";
225   ofile<<setw(20)<<"2nd(Ja)"<<endl;

227   for(int i=0; i<(n-1); i++)
      {
229         ofile<<rho_i[i+1];
          ofile<<setw(20)<<(1/h)*V0(i)*V0(i);
231       ofile<<setw(20)<<(1/h)*V1(i)*V1(i);
          ofile<<setw(20)<<(1/h)*V2(i)*V2(i);
233       ofile<<setw(20)<<(1/h)*R0(i)*R0(i);
          ofile<<setw(20)<<(1/h)*R1(i)*R1(i);
235       ofile<<setw(20)<<(1/h)*R2(i)*R2(i)<<endl;
      }
237
      ofile.close();
239
      //Wrting to file with Eigenvalues
241
      ofile.open(Eigen);
243   ofile<<setiosflags(ios::showpoint | ios::uppercase);
      ofile<<"#Eig(Ja)"<<setw(25)<<"Time(Ja)";
245   ofile<<setw(25)<<"iterations"<<setw(25)<<"Eig(Ar)";
      ofile<<setw(25)<<"Time(Ar)"<<endl;
247
      for (int i=0; i<5; i++)
249   {
        ofile<<setprecision(6)<<Max(i);
251     ofile<<setw(25)<<setprecision(6)<<time_j;
        ofile<<setw(25)<<iterations;
253     ofile<<setw(25)<<eigval(i);
        ofile<<setw(25)<<time_ar<<endl;
255   }


257
      ofile.close();
259
      delete [] V_Pot;
261   delete [] V_Pin;
      delete [] rho_i;
263
}//End Main Program
265
void rotate(mat &A, mat &R, int k, int l, int n)
267 {

269   //Rotation of the Matrix
            double tau;
271         double t;
            double s;
273         double c;
```

```
275         if(A(k,l) != 0.0)
            {
277           tau = (A(l, l)-A(k,k))/(2*A(k,l));

279           if(tau > 0)
              {
281             t = -tau + sqrt(1 + tau*tau);
              }
283           else
              {
285             t = -tau - sqrt(1+ tau*tau);
              }
287           c = 1/sqrt(1 + t*t);
              s = t*c;
            }
            else
291         {
              c =1.0;
293           s= 0.0;
    }
295
    double a_kk, a_ll, a_ik, a_il, r_ik, r_il;
297         a_kk = A(k,k);
            a_ll = A(l,l);
299
            //Changing the matrix elemensts with indeces k an l
301         A(k, k) = c*c*a_kk - 2.0*c*s*A(k,l) + s*s*a_ll;
            A(l, l) = s*s*a_kk + 2.0*c*s*A(k,l) + c*c*a_ll;
303         A(k,l) = 0.0;
            A(l,k) = 0.0;
305
            //Changing the remaining elements
307         for(int i= 0; i< (n-1); i++)
            {
309             if( i != k && i != l)
                {
311                 a_ik = A(i,k);
                    a_il = A(i,l);
313                 A(i,k) = a_ik*c - a_il*s;
                    A(k,i) = A(i,k);
315                 A(i,l) = a_il*c + a_ik*s;
                    A(l,i) = A(i,l);
317              }

319             //Compute the new eigenvectors
                r_ik = R(i,k);
321             r_il = R(i,l);
                R(i,k) = c*r_ik - s*r_il;
323             R(i,l) = c*r_il + s*r_ik;
    }
325 return;
}
```