بسم الله الرحمن الرحيم

AECENAR

Association for Economical and Technological Cooperation in the Euro-Asian and North-African Region

www.aecenar.com

IAP INSTITUTE FOR ASTROPHYSICS

TYPE II SUPERNOVA

Author:

Maryam ABDEL-KARIM

22-Jun-2019

Contents

Introduction مدخل 3

Chapter 1: Basics 4

1. **Introduction** 4
2. **Star evolution** 4
3. **Stellar composition** 4
4. **Assumption** 5
5. **Numerical model** 6
6. **Physics of supernova** 6
    - **6.1. Pressure** 6
    - **6.2. Energy** 7
    - **6.3. Temperature** 7

Chapter 2: Program 9

1. **Classes** 9
2. **Results saving** 9

Chapter 3: Results 11

1. **Input** 11
2. **Results** 11
3. **Results viewed on Paraview** 12

Annex A 14

Introduction مدخل

The code presented here is inspired from the report: << ‘’Modeling a Type-II supernova”- F.S. Nobels, J. Ubink, H.W. de vries, Guided by: Prof. Dr. O. Scholten. January 21,2015 >>. The aim was to model the hydrodynamics of a type-II core collapse supernova using C++. code was created for modelling. The typical supernova explosion was not observed due to instability of the code, but a test on the earth-like atmosphere was done, it can be concluded that the hydrodynamics of the supernova model probably work correctly.

البرنامج المطروح في هذا العمل مستوحى من التقرير:

<< ‘’Modeling a Type-II supernova”- F.S. Nobels, J. Ubink, H.W. de vries, Guided by: Prof. Dr. O. Scholten. January 21,2015 >>.

الهدف هو نمذجة الديناميكا المائية لنجم سوبر نوفا من النوع الثاني (Supernova type-II) باستخدام C++. لم يُلاحظ انفجار المستعر الأعظم النموذجي بسبب عدم ثبات الكود، ولكن تم إجراء اختبار على الغلاف الجوي الشبيه بالأرض، ويمكن الاستنتاج أن الديناميكا المائية لنموذج السوبرنوفا تعمل بشكل صحيح.

Chapter 1: Basics

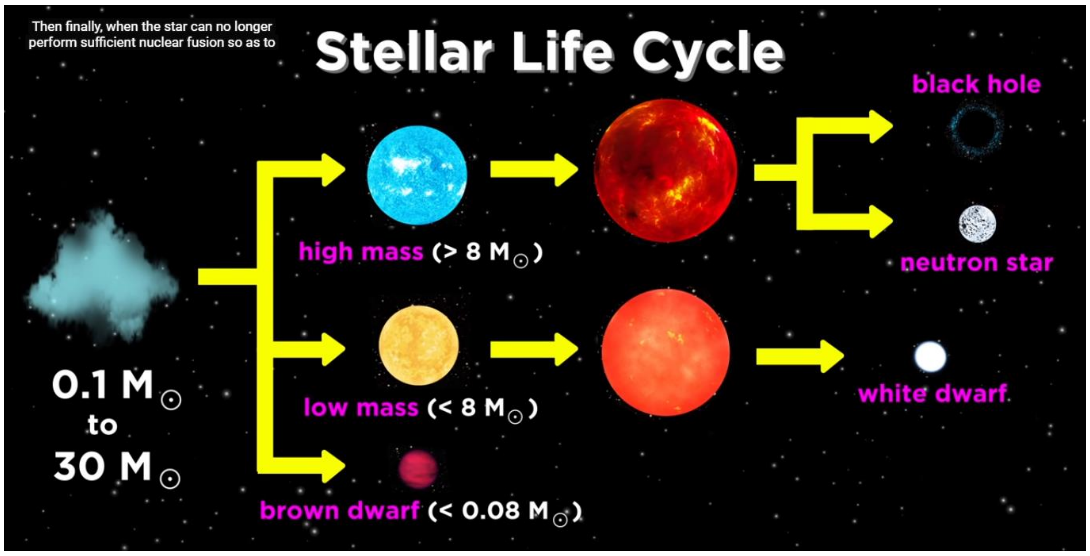
1. Introduction

A star death may be accompanied by a tremendous explosion called supernova. Being the brightest events in the entire universe, a typical supernova can outshine an entire galaxy, emitting a typical amount of 10^{24} J of energy as radiations.

The following code does not take into consideration the luminosity, Decay processes, Opacity and Blast waves.

2. Star evolution

In order for a supernova to occur, a star has to meet certain criteria, of which the most important one for mode will be its mass. If a star is too light, it will not be capable of showcasing a type-II supernova. One can illustrate this criterion by having a look at the influence of mass on the fusion processes that occur in the star. (see figure below)


Then finally, when the star can no longer perform sufficient nuclear fusion so as to
Stellar Life Cycle
black hole
high mass (> 8 M$_\odot$)
neutron star
0.1 M$_\odot$ to 30 M$_\odot$
low mass (< 8 M$_\odot$)
white dwarf
brown dwarf (< 0.08 M$_\odot$)

The Star types are:

- Main sequence:
    - Blue stars: big, hot, bright (up to 200 solar masses)
    - Yellow stars: in between (close to 1 solar mass)
    - Red stars: small, cool, dim (down to 0.1 solar masses)
- Red Giant: red and cool (0.3-8 solar masses)
- White dwarf: tiny and hot (0.2- 1.3 solar masses)

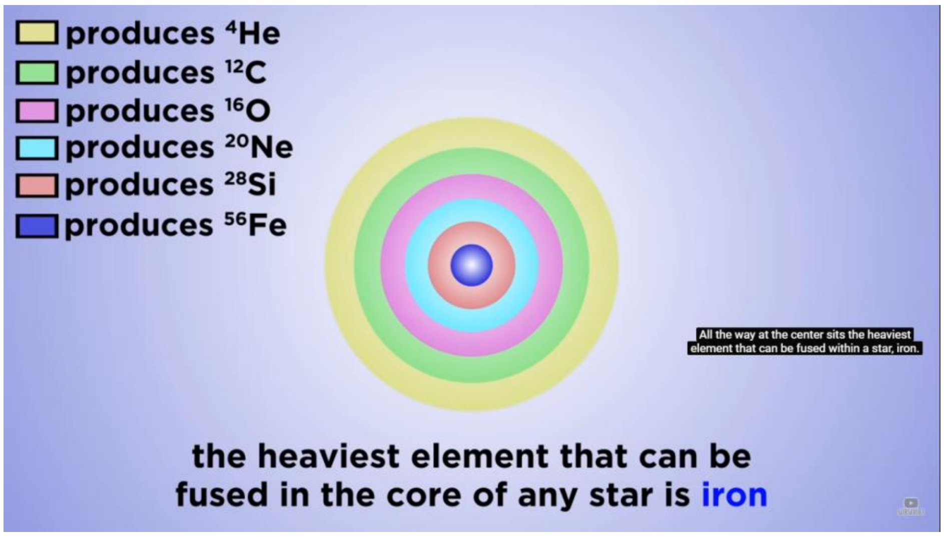
3. Stellar composition

Two different aspect of stellar composition are considered:

- A neutron star core
- Red giant star

3.1. **Neutron star core model:** The star contains only the element hydrogen, and a neutron star at its core. The fact that there is a neutron star at the center of the atmosphere, comes from the collapsing of the iron core in the pre-supernova situation. In the simulations, the model starts at the moment at which the atmosphere rebounds from the neutron star core.

3.2. **Red giant star model:** The stellar material is assumed to consist of approximately 90% hydrogen and a stellar core of iron, the stellar atmosphere is divided into layers of different elements.



🟨 produces ^{4}He
🟩 produces ^{12}C
🟪 produces ^{16}O
🟦 produces ^{20}Ne
🟥 produces ^{28}Si
🟦 produces ^{56}Fe
All the way at the center sits the heaviest element that can be fused within a star, iron.
the heaviest element that can be fused in the core of any star is iron

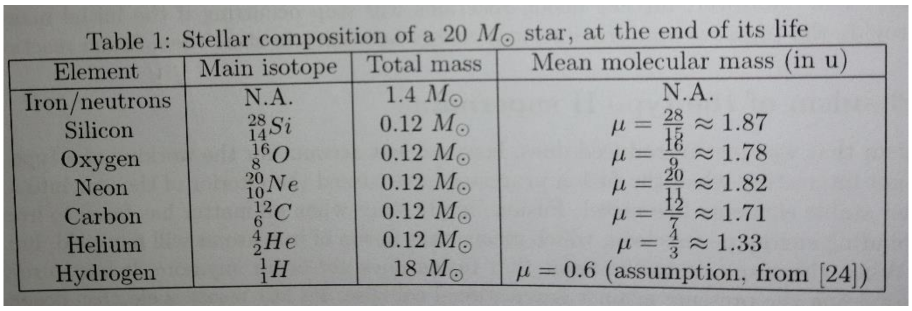


Table 1: Stellar composition of a 20 $M_\odot$ star, at the end of its life

Element	Main isotope	Total mass	Mean molecular mass (in u)
Iron/neutrons	N.A.	1.4 $M_\odot$	N.A.
Silicon	$^{28}_{14}Si$	0.12 $M_\odot$	$\mu = \frac{28}{15} \approx 1.87$
Oxygen	$^{16}_{8}O$	0.12 $M_\odot$	$\mu = \frac{16}{9} \approx 1.78$
Neon	$^{20}_{10}Ne$	0.12 $M_\odot$	$\mu = \frac{20}{11} \approx 1.82$
Carbon	$^{12}_{6}C$	0.12 $M_\odot$	$\mu = \frac{12}{7} \approx 1.71$
Helium	$^{4}_{2}He$	0.12 $M_\odot$	$\mu = \frac{4}{3} \approx 1.33$
Hydrogen	$^{1}_{1}H$	18 $M_\odot$	$\mu = 0.6$ (assumption, from [24])

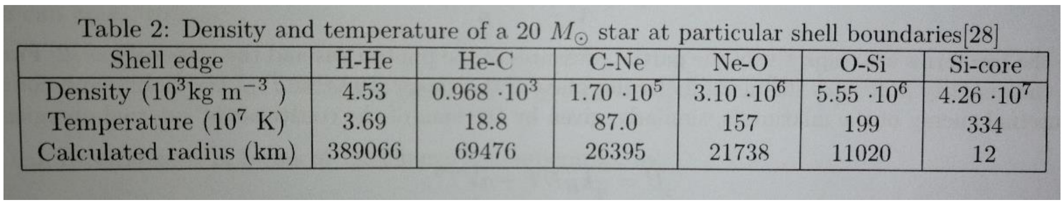


Table 2: Density and temperature of a 20 $M_\odot$ star at particular shell boundaries[28]

Shell edge	H-He	He-C	C-Ne	Ne-O	O-Si	Si-core
Density (10^3kg m^{-3})	4.53	$0.968 \cdot 10^3$	$1.70 \cdot 10^5$	$3.10 \cdot 10^6$	$5.55 \cdot 10^6$	$4.26 \cdot 10^7$
Temperature (10^7 K)	3.69	18.8	87.0	157	199	334
Calculated radius (km)	389066	69476	26395	21738	11020	12

4. Assumption

Basics Assumptions:

- Spherical symmetry (The star's properties are assumed to have no angular dependence)
- Gas composition (it is assumed that the star consists of a mixture of photon gas and ideal classical gas)
- Homogeneous distributions (the same value for the density, pressure, and temperature hold at each radial position inside a certain shell. In the model, these quantities are evaluated in the middle of shells)

5. Numerical model

In this model, the star is divided into a number of spherical shells.

These shells contain stellar material with a certain density, temperature, pressure and internal energy.

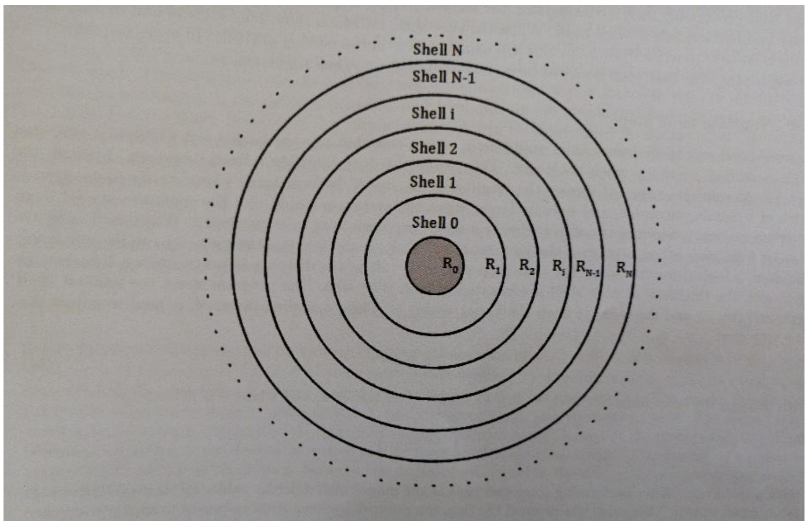


Figure 1: This diagram(not to scale) illustrates the discretization of a stellar atmosphere into spherical shells. The shaded center of the diagram denotes the neutron star core. The dashed circle above radius R_N represents the mass that is located above that shell.

The matter within a shell is regarded as being isolated from the matter within other shells. The shell may change in volume but the same mass remains inside of it.

The dynamics of the star can be simulated by studying the movement of these spherical shells. By looking at the way in which the radial position of each shell changes due to gravity and pressure, gives an understanding of the dynamics of the star as a whole.

6. Physics of supernova

6.1. Pressure

The star consists of a mixture of photon gas and ideal classical gas. The pressure of the stellar gas is then given by:

$$P = \frac{a}{3}T^4 + \frac{\rho k_B T}{\mu}$$

Where the two terms are respectively the partial pressures of the photon gas and the classical gas.

μ: The mean molecular mass of the classical gas.

k_B: The Boltzmann constant.

T: The temperature.

6.2.Energy

The temperature of a stellar shell changes due to changes in its energy. It is therefore of importance to first find the expressions for the total energy of a shell.

- The first part of the total energy is the internal energy. The internal energy of the mixture is given by the sum of the constituents' internal energies.

  $$U = \frac{2}{3}\, k_B N T + aVT^4$$

  Where the first contribution comes from the classical gas and the second from the photon gas.

  N: number of particles.

  A: radiation constant.

  For high temperatures, the photon gas will become dominant over the classical gas.

- The second energy term is the kinetic energy, which is given by:

  $$E_{kin} = \frac{1}{2}\, m_{shell} \left(\frac{v_{i+1} + v_i}{2}\right)^2$$

  Where the average velocity of the two surfaces (i.e. the two shell radii) enclosing a shell of stellar material are taken to represent the velocity of the matter inside the shell.

- The third and final term is the gravitational potential energy, and it has negative contribution to the total energy. It is given by:

  $$E_{grav} = -\frac{G M_{encl} m_{shell}}{r^2}$$

So combining the above equations gives the following expression for the total energy of a stellar shell:

$$U + E_{kin} + E_{grav} = \frac{2}{3}\, k_B N T + aVT^4 + \frac{1}{2}\, m_{shell} \left(\frac{v_{i+1} + v_i}{2}\right)^2 - \frac{G M_{encl} m_{shell}}{r^2}$$

6.3.Temperature

When one assumes that the shell expands and shrinks adiabatically, its temperature is related to its internal energy. Therefore, one can determine the change in temperature of a shell by means of the change in energy.

The internal energy of a shell can change in two ways: by exchanging heat and by performing work:

$$dU = d\omega + dq$$

The stellar gas is assumed to expand adiabatically so dq is set to zero.

$$dU = d\omega = -PdV$$

By replacing the internal energy and the pressure by their values (with $N = \frac{m_{shell}}{\mu}$) we'll have the following expression in terms of the change in temperature as:

$$dT = \frac{-\left(\frac{\rho kT}{\mu} + \frac{aT^4}{3} + aT^4\right) dV}{\frac{3}{2} k \frac{m_{shell}}{\mu} + 4aVT^3}$$

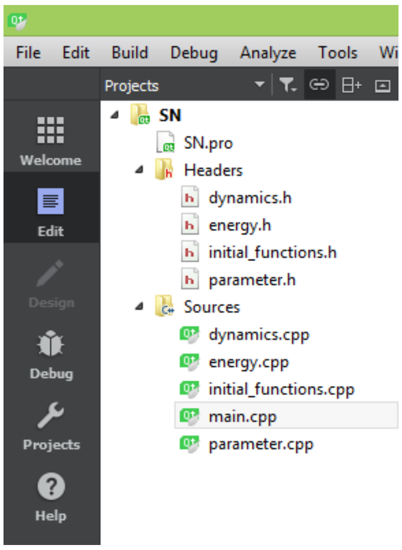
It describes how the temperature of a shell of stellar matter changes due to changes in its volume.

Chapter 2: Program

1. Classes

Our program is composed of 4 classes:

- Initial functions
- Dynamics
- Energy
- Parameter



The first class Initial functions calculate the initial values of the pressure, temperature, density and the H constant.

The second class Dynamics calculate the acceleration, pressure generation and the temperature distribution

The third class Energy calculate the kinetic energy, internal energy and the gravitational energy.

The fourth class Parameter define the number of shell desired.

The initial conditions are specified in the main program.

For the full program see Annex A.

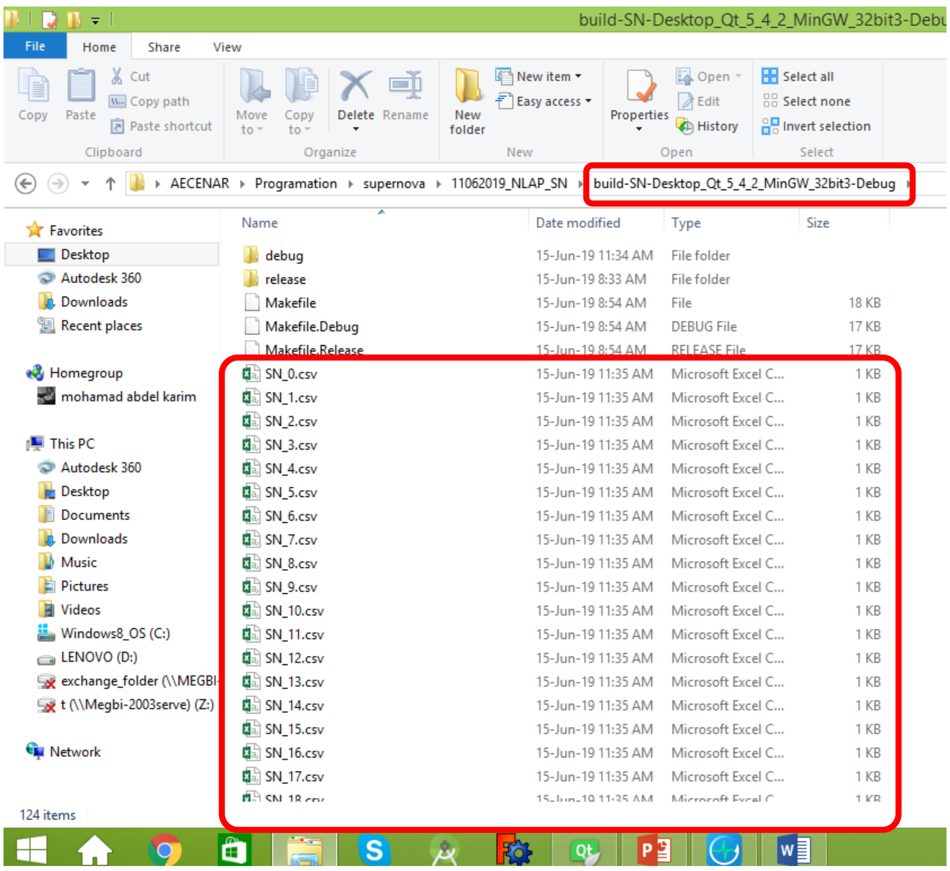
2. Results saving

A simple code has been added to the program to save the results automatically into files.csv.

```
#include "iostream"
#include "fstream"
#include "sstream"
```

```
string filename;

ofstream myfile;

stringstream b;
b<<i;//number of iteration
    filename= "SN_"+ b.str();
    filename+= ".csv";
    myfile.open(filename.c_str());

  for(int k=0;k<d;k++){
        rold[k]=r[k];//store the old radii
        accel[k]=dyn.acc(mass[k],r[k],rho[k],dp_over_dr[k],G);
        v[k]=v[k]*damping+accel[k] * dt;
        r[k]=r[k]+v[k] * dt+accel[k] * dt * dt;
        y=sqrt(r[k]*r[k]-(k+1)*(k+1));
        if(r[k]<0){
            r[k]=0;
        }

myfile<<0<<","<<r[k]<<",0,"<<r[k]<<","<<T[k]<<","<<dt<<","<<pressure[
k]<<"\n";

    }
  myfile.close();
```

This code will create a file named SN000.csv for each iteration. They will be saved in the build folder.



AECENAR › Programation › supernova › 11062019_NLAP_SN › build-SN-Desktop_Qt_5_4_2_MinGW_32bit3-Debug

Name	Date modified	Type	Size
debug	15-Jun-19 11:34 AM	File folder	
release	15-Jun-19 8:33 AM	File folder	
Makefile	15-Jun-19 8:54 AM	File	18 KB
Makefile.Debug	15-Jun-19 8:54 AM	DEBUG File	17 KB
Makefile.Release	15-Jun-19 8:54 AM	RELEASE File	17 KB
SN_0.csv	15-Jun-19 11:35 AM	Microsoft Excel C...	1 KB
SN_1.csv	15-Jun-19 11:35 AM	Microsoft Excel C...	1 KB
SN_2.csv	15-Jun-19 11:35 AM	Microsoft Excel C...	1 KB
SN_3.csv	15-Jun-19 11:35 AM	Microsoft Excel C...	1 KB
SN_4.csv	15-Jun-19 11:35 AM	Microsoft Excel C...	1 KB
SN_5.csv	15-Jun-19 11:35 AM	Microsoft Excel C...	1 KB
SN_6.csv	15-Jun-19 11:35 AM	Microsoft Excel C...	1 KB
SN_7.csv	15-Jun-19 11:35 AM	Microsoft Excel C...	1 KB
SN_8.csv	15-Jun-19 11:35 AM	Microsoft Excel C...	1 KB
SN_9.csv	15-Jun-19 11:35 AM	Microsoft Excel C...	1 KB
SN_10.csv	15-Jun-19 11:35 AM	Microsoft Excel C...	1 KB
SN_11.csv	15-Jun-19 11:35 AM	Microsoft Excel C...	1 KB
SN_12.csv	15-Jun-19 11:35 AM	Microsoft Excel C...	1 KB
SN_13.csv	15-Jun-19 11:35 AM	Microsoft Excel C...	1 KB
SN_14.csv	15-Jun-19 11:35 AM	Microsoft Excel C...	1 KB
SN_15.csv	15-Jun-19 11:35 AM	Microsoft Excel C...	1 KB
SN_16.csv	15-Jun-19 11:35 AM	Microsoft Excel C...	1 KB
SN_17.csv	15-Jun-19 11:35 AM	Microsoft Excel C...	1 KB
SN_18.csv	15-Jun-19 11:35 AM	Microsoft Excel C	1 KB

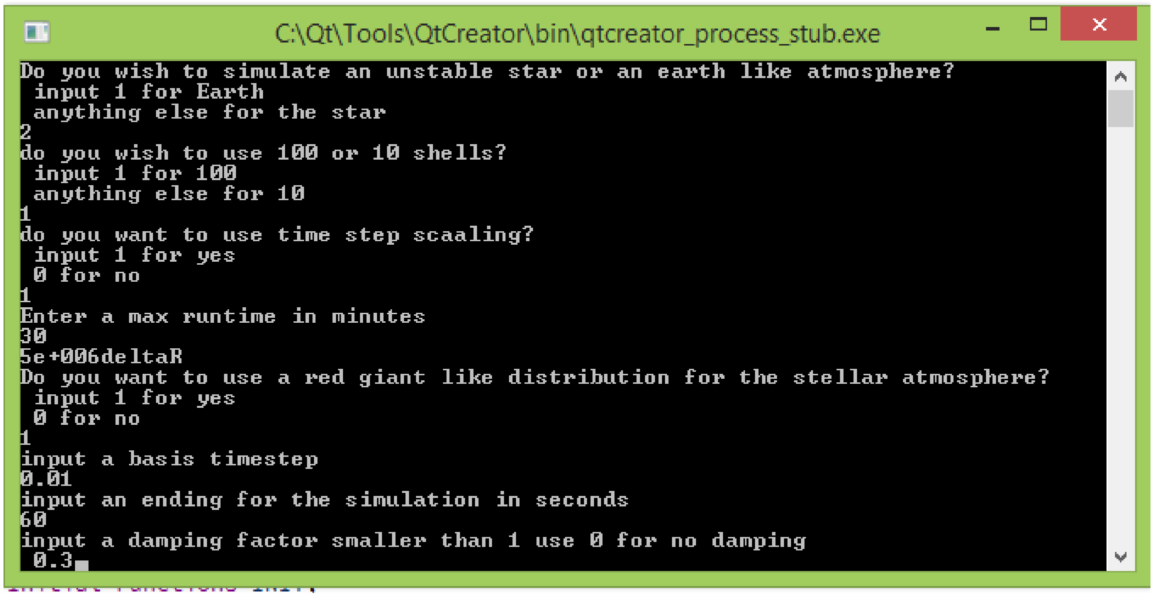
Chapter 3: Results

1. Input

The user first specifies if he wants to simulate an unstable star or an earth like atmosphere, then the number of shells desired (10 or 100 shells).

The program presents the option to have a time step scaling. The user should present a max runtime in minutes, a basis time step, an ending for the simulation and finally the damping factor.

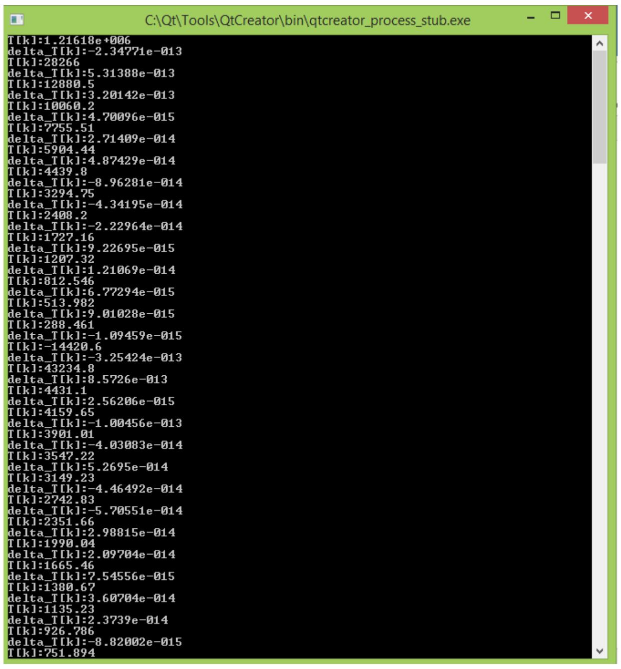
The user should also choose whether he wants a neutron core or a red giant distribution.



```
Do you wish to simulate an unstable star or an earth like atmosphere?
 input 1 for Earth
 anything else for the star
2
do you wish to use 100 or 10 shells?
 input 1 for 100
 anything else for 10
1
do you want to use time step scaaling?
 input 1 for yes
 0 for no
1
Enter a max runtime in minutes
30
5e+006deltaR
Do you want to use a red giant like distribution for the stellar atmosphere?
 input 1 for yes
 0 for no
1
input a basis timestep
0.01
input an ending for the simulation in seconds
60
input a damping factor smaller than 1 use 0 for no damping
 0.3
```

2. Results

The results start to appear on the console and result files are created for each time step in the directory

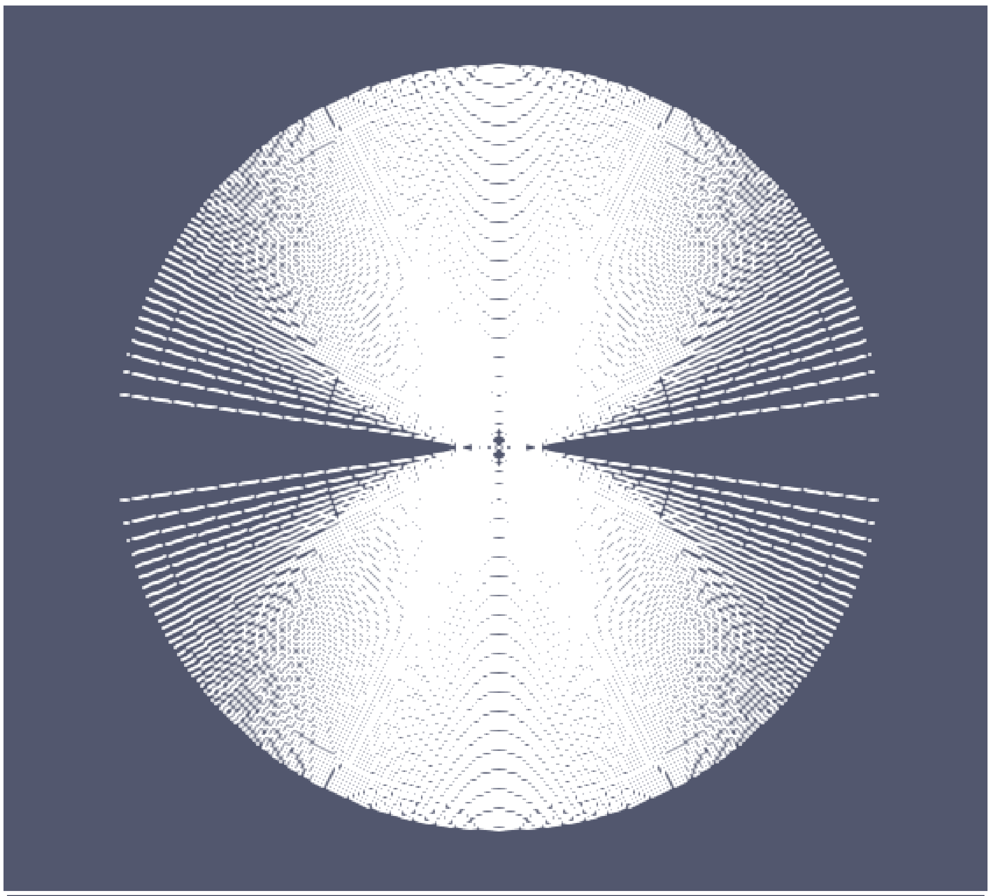


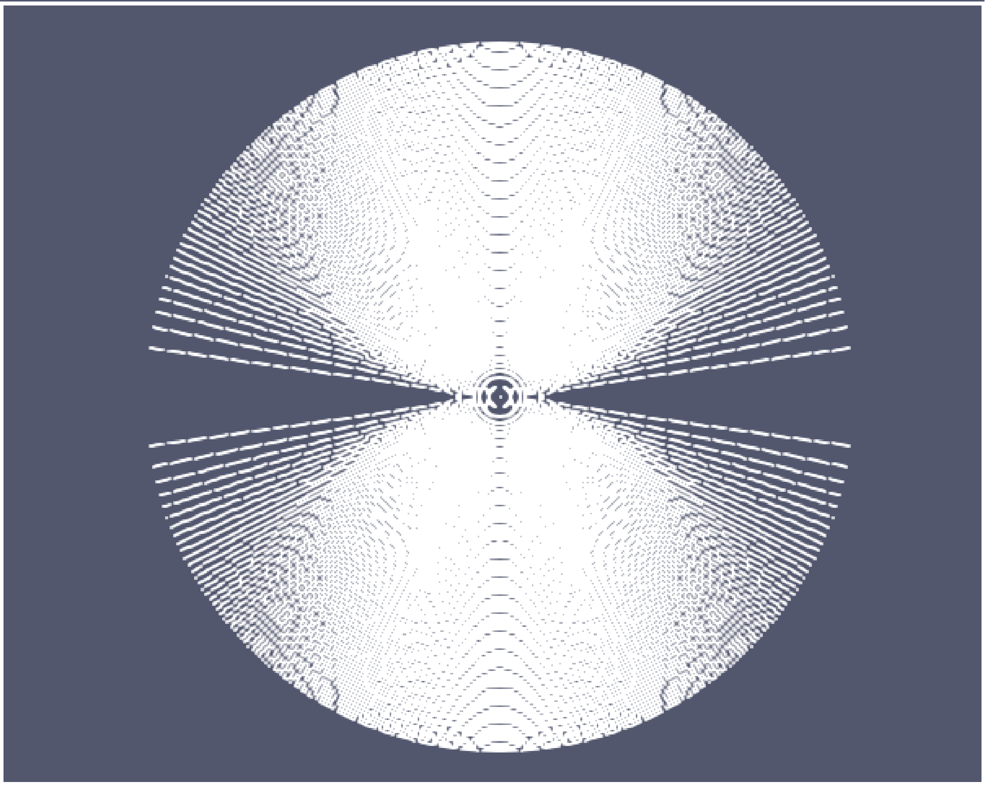
```
T[k]:1.21618e+006
delta_T[k]:-2.34771e-013
T[k]:28266
delta_T[k]:5.31388e-013
T[k]:12880.5
delta_T[k]:3.20142e-013
T[k]:10060.2
delta_T[k]:4.70096e-015
T[k]:7755.51
delta_T[k]:2.71409e-014
T[k]:5904.44
delta_T[k]:4.87429e-014
T[k]:4439.8
delta_T[k]:-8.96281e-014
T[k]:3294.75
delta_T[k]:-4.34195e-014
T[k]:2408.2
delta_T[k]:-2.22964e-014
T[k]:1727.16
delta_T[k]:9.22695e-015
T[k]:1207.32
delta_T[k]:1.21069e-014
T[k]:812.546
delta_T[k]:6.77294e-015
T[k]:513.982
delta_T[k]:9.01028e-015
T[k]:288.461
delta_T[k]:-1.09459e-015
T[k]:-14420.6
delta_T[k]:-3.25424e-013
T[k]:43234.8
delta_T[k]:8.5726e-013
T[k]:4431.1
delta_T[k]:2.56206e-015
T[k]:4159.65
delta_T[k]:-1.00456e-013
T[k]:3901.01
delta_T[k]:-4.03083e-014
T[k]:3547.22
delta_T[k]:5.2695e-014
T[k]:3149.23
delta_T[k]:-4.46492e-014
T[k]:2742.83
delta_T[k]:-5.70551e-014
T[k]:2351.66
delta_T[k]:2.98815e-014
T[k]:1990.04
delta_T[k]:2.09704e-014
T[k]:1665.46
delta_T[k]:7.54556e-015
T[k]:1380.67
delta_T[k]:3.60704e-014
T[k]:1135.23
delta_T[k]:2.3739e-014
T[k]:926.786
delta_T[k]:-8.82002e-015
T[k]:751.894
```

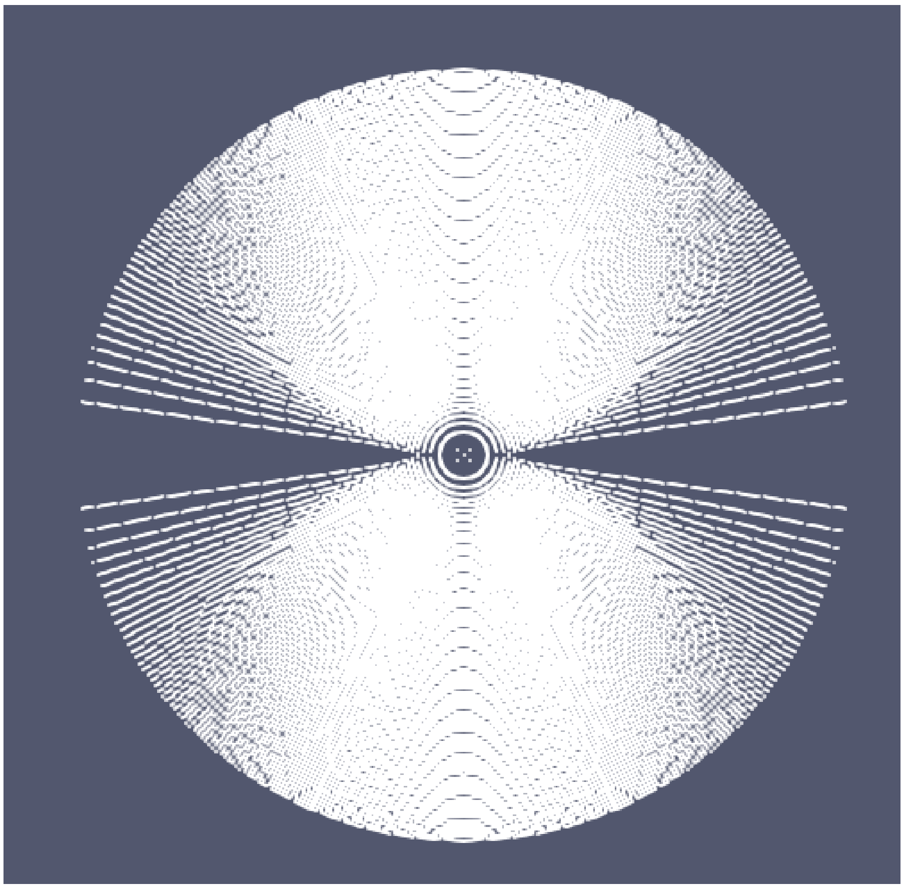
3. <u>Results viewed on Paraview</u>

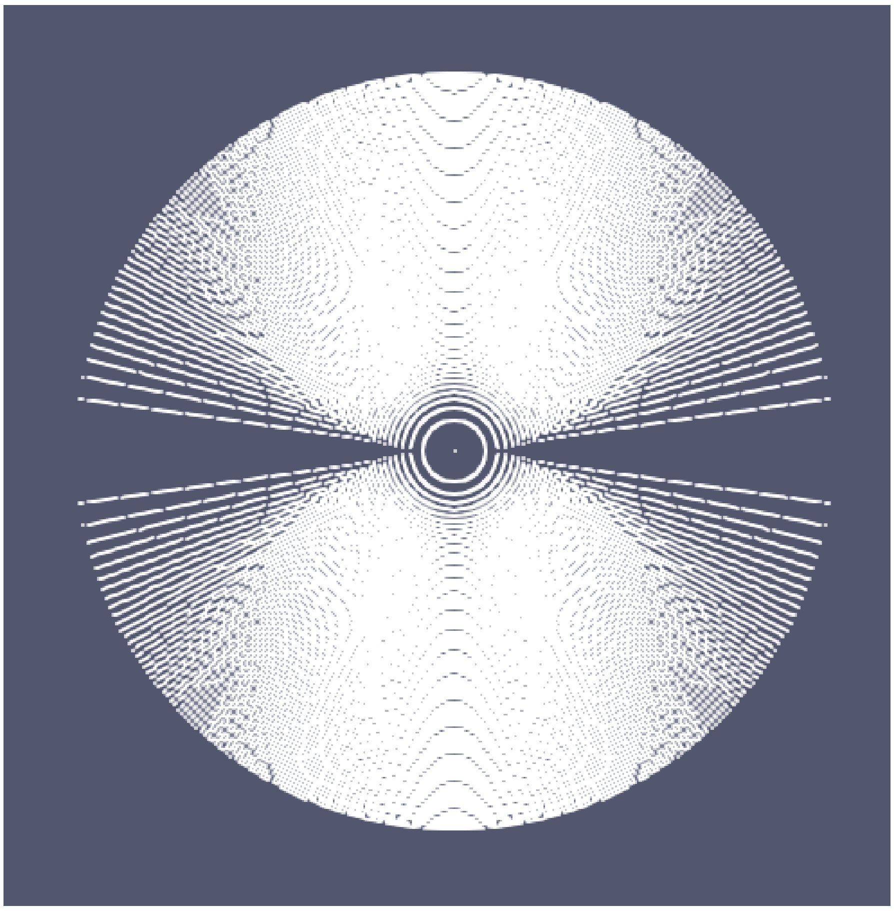
The results shown in this section are for a mesh composed of 100 shells.

<u>For 100 shells</u>





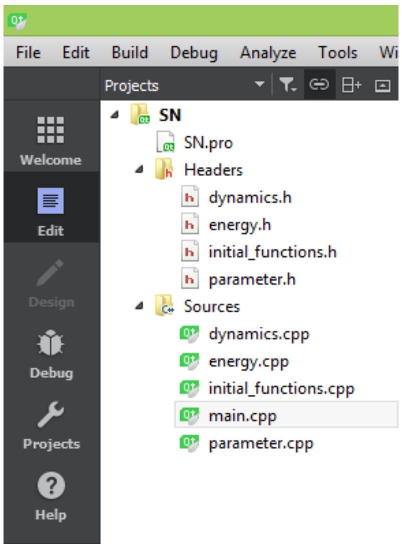




Annex A

Our program is composed of 4 classes:

- Initial functions
- Dynamics
- Energy
- Parameter



```
SN
    SN.pro
    Headers
        dynamics.h
        energy.h
        initial_functions.h
        parameter.h
    Sources
        dynamics.cpp
        energy.cpp
        initial_functions.cpp
        main.cpp
        parameter.cpp
```

A.1. Class Initial functions

Initial functions.h

```
#ifndef INITIAL_FUNCTIONS_H
#define INITIAL_FUNCTIONS_H


class Initial_Functions
{
public:
    Initial_Functions(){

    }

    double rhof(double beginrho,double exponentfactrho,double
r,double Rearth,double deltaR, double H_constant);
    double temp(double begintemp);
    double pressure(double beginpres,double exponentfactpres,double
r,double Rearth,double deltaR, double H_constant);
    double H_constant(double molecular_mass, double grav_acc,double
avogadro,double T, double gascons);
    double der(double ai,double ai_1,double bi,double bi_1);

};

#endif // INITIAL_FUNCTIONS_H
```

Initial functions.cpp

```
#include "initial_functions.h"
#include "math.h"
```

```
//initial density, temperature, and pressure functions
// density distribution at t=0

double Initial_Functions::rhof(double beginrho,double
exponentfactrho,double r,double Rearth,double deltaR, double
H_constant){
double rhof;
    rhof=beginrho*exp(-exponentfactrho*(r-Rearth-deltaR)/H_constant);


return rhof;
}

// temp distribution at t=0
double Initial_Functions::temp(double begintemp){

    return begintemp;
}

// pressure distribution at t=0
double Initial_Functions::pressure(double beginpres,double
exponentfactpres,double r,double Rearth,double deltaR,double
H_constant){
double P;
    P=beginpres*exp(-exponentfactpres*(r-Rearth-deltaR)/H_constant);

    return P;
}

// function for H_constant
double Initial_Functions::H_constant( double molecular_mass, double
grav_acc,double avogadro,double T,double gascon){
 double H;

 H=gascon*T/(grav_acc*molecular_mass*avogadro);

    return H;
}

//derivative function

double Initial_Functions::der(double ai,double ai_1,double bi,double
bi_1){

    double k;

        k=(ai-ai_1)/(bi-bi_1);

return k;
}
```

A.2. Class dynamics

Dynamics.h

```
#ifndef DYNAMICS_H
#define DYNAMICS_H


class dynamics
{
```

```
public:
    dynamics(){

    }

double acc(double m_encl, double r, double rho, double dp_over_dr,
double G);
double pressure_gen(double T, double rho, double
molecular_mass,double a,double boltz);
double tempdistr(double r, double T1, double T2,double R1,double R2);

};

#endif // DYNAMICS_H
```

Dynamics.cpp

```
#include "dynamics.h"
#include "math.h"

//acceleration
double dynamics::acc(double m_encl, double r, double rho, double
dp_over_dr, double G){
double acc;
if(r<=0){
    return 0;
}
else{
acc=-G*m_encl/(r*r)-(1/rho)*dp_over_dr;
        return acc;
}
}

//Pressure
double dynamics::pressure_gen(double T, double rho, double
molecular_mass,double a,double boltz){
    double press;
    press=(a/3)*pow(T,4)+rho*boltz*T/molecular_mass;
return press;
}

//this function is used for interpolating the temperature between
element layers
//in the situation of stellar distribution with elements
double dynamics::tempdistr(double r, double T1, double T2,double
R1,double R2){
  double tempdist;
  tempdist=T1*exp(-(1/(R2-R1))*log(T1/T2)*r);
    return tempdist;
}
```

A.3. Class energy

Energy.h

```
#ifndef ENERGY_H
#define ENERGY_H


class Energy
{
```

```
public:
    Energy(){

    }

    double gravenergy(double mass, double massencl,double r, double
G);
    double internalenergy(double volume,double temperature,double
mass, double molecularmass,double enercon,double boltz);
    double kineticenergy(double mass, double velocity);
};

#endif // ENERGY_H
```

Energy.cpp

```
#include "energy.h"
#include "math.h"

//gravitational energy
double Energy::gravenergy(double mass, double massencl,double r,
double G){
    return mass*massencl*G/r;

}

//internal energy
double Energy::internalenergy(double volume,double temperature,double
mass, double molecularmass,double enercon,double boltz){

    return
enercon*volume*pow(temperature,4)+(3/2)*boltz*mass/molecularmass*temp
erature;
}

//kinetic energy
double Energy::kineticenergy(double mass, double velocity){

    return 0.5*mass*pow(velocity,2);

}
```

A.4. Class parameter

Parameter.h

```
#ifndef PARAMETER_H
#define PARAMETER_H


class parameter
{
public:
    parameter(){

    }
int shellnumber(int wanthundred);
};

#endif // PARAMETER_H
```

Parameter.cpp

```
#include "parameter.h"


int parameter::shellnumber(int wanthundred){
    int d;
    if(wanthundred==1){
        d=100;
    }
    if(wanthundred!=1){
        d=10;
    }
    return d;
}
```

A.5. Main program

```
#include <QCoreApplication>
#include "math.h"
#include "dynamics.h"
#include "energy.h"
#include "parameter.h"
#include "initial_functions.h"
#include "stdio.h"
#include "ctime"
#include "algorithm"
#include "iostream"
#include "fstream"
#include "sstream"
#include <QtCore/QString>
#include <QtCore/QFile>
#include <QtCore/QDebug>
#include <QtCore/QTextStream>

using namespace std;

int main(int argc, char **argv)
{
int atmossim;
int wanthundred;
int wantdtscaling;
int wantdensity;
double tmax;
double shellnumber;
double deltaR;
double Rearth;
double beginrho;
double beginpress;
double begintemp;
double exponentfactorrho;
double exponentfactorpress;
double Ratm;
double Mstar;
double mol;
double dt;
double t_final;
double dampfactor;
double pi;
double boltz;
double gascon;
double avogadro;
double c;
double sig;
double a;
double hbar;
double enercon;
double G;
double grav_acc;


/*/////////////////////////////////////
//      Initial conditions          //
/////////////////////////////////////*/

    double Msolar=1.99e30;//mass of the sun in kg
    double Munit=1.660538921e-27;//atomic mass unit in kg
    double Rsolar=6.955e8;//Radius of the sun in m


//user input
cout<< "Do you wish to simulate an unstable star or an earth like
atmosphere?\n input 1 for Earth\n anything else for the star\n";
cin >> atmossim;
cout<< "do you wish to use 100 or 10 shells?\n input 1 for 100\n
anything else for 10\n";
cin>> wanthundred;
cout<<"do you want to use time step scaaling?\n input 1 for yes\n 0
for no\n";
cin>> wantdtscaling;
cout<<"Enter a max runtime in minutes\n";
cin>> tmax;

if(atmossim==1){//use earth conditions
    //set the shell number
    shellnumber=1;
    if(wanthundred==1){
        shellnumber=0.1;
    }
    //set the shell thickness
    deltaR=5000 * shellnumber;
    //Radius of Earth
    Rearth=6400 * pow(10,3) - deltaR;
    //set parameters for use in the initial distribution functions
    beginrho=1.251;
    beginpress=1e5/beginrho;
    begintemp=293;//293000*5.55 stability point
    //set a factor that is used to adjust the density distr.
    //it is 1 here because here no adjustment is necessary
    exponentfactorrho=1;
    exponentfactorpress=exponentfactorrho;
    //set the title of the output graph
    //graphtitle='numerical simulation of the atmosphere of the
earth';
    //radius of the atmosphere
    Ratm=50000+Rearth;
    //set the mass of the earth (the name Mstar is still used for
this variable
    Mstar=5.9721986 * pow(10,24);
    //set that the division in element layers as used in the star
simulation is not used
    wantdensity=0;
    //this variable is the molecular mass of nitrogen
    mol=4.65173 * pow(10,-26);

}

else{// use star conditions
    //initially set that the element didtribution is not used
    wantdensity=0;
    //adjust the number of shells
    shellnumber=1;
    if(wanthundred==1){
        shellnumber=0.1;
    }
    deltaR=50000000 * shellnumber;
    cout<<deltaR<<"deltaR \n";
    //set the radius of the star core (the term Rearth is a remnant
of earlier version of the model
    //and has been kept in. The variable does in fact denote the
neutron star core's radius
    Rearth=12000-deltaR;
    //set the radius of the stellar atmosphere
    Ratm=500000000+Rearth;
    //set the parameters for initial distribution
    exponentfactorrho=0.0001;
    exponentfactorpress=5;
    beginrho=1.251e5;
    begintemp=237000;
    beginpress=1e33;
    //set the mass of the stellar core
    Mstar=Msolar * 1.4;
    //set the title of the output graph
   // graphtitle='numerical simulation of a supernova explosion';
    if(wanthundred==1){//check wether the user has chosen 100 shells
        //Ask wether the user wants to use the stellar atmosphere
distribution
        //where it is divided in a number of layers containing
certain elements
        cout<<"Do you want to use a red giant like distribution for
the stellar atmosphere?\n input 1 for yes\n 0 for no\n";
        cin>>wantdensity;

    }
    //this variable is the molecular mass of hydrogen
    mol=3.34745e-27;
}

//ask the user to input some time variables
cout<<"input a basis timestep\n";
cin>> dt;
cout<<"input an ending for the simulation in seconds\n";
cin>> t_final;
cout<< "input a damping factor smaller than 1 use 0 for no damping\n
";
cin>> dampfactor;

//define a number of constants. All physical constants are in SI
units
//Pi
```

```
pi=3.141592653;
//the boltzmann constant
boltz=1.38064852e-23;
//the gas constant
gascon=8.3144;
//Avogadro's number
avogadro=6.02214129e23;
//speed of light
c=299792458;
//radiation constants
sig=5.67e-8;
a=(4 * sig)/c;
//reduced Planck's constant
hbar=6.62607015e-34;

//define an energy constant.
// This combination of constants occurs multiple times in the program
and is therefore defined for convenience.

enercon=(pow(pi,2) * pow(boltz,4))/(15 * pow(c,3) * pow(hbar,3));
//gravitation constant
G=6.67408e-11;
//gravitational acceleration on earth
grav_acc=9.807;


/*///////////////////////////////////////
// Calculation of the initial Arrays //
//////////////////////////////////////*/

//set the array containing the shell radii
parameter par;
int d;//array lenght

d=par.shellnumber(wanthundred);
double r[d];

for(int k=0;k<d;k++){
   r[0]=Rearth+deltaR;
   r[k+1]=r[k]+deltaR;

}
//initialize an array to be used later
double rold[d];
//create an array giving the mean molecular mass of the material
inside shells
double molecularmass[d];
for(int i=0;i<d;i++){
    rold[i]=0;
    molecularmass[i]=mol;
}

//calculate the density between radii (loop_rho) and radii themselves
(rho)
double loop_rho[d];
double rho[d];
double H_constant=0;
double temp;
Initial_Functions INIT;

for(int i=0;i<d;i++){
    temp=INIT.temp(begintemp);

H_constant=INIT.H_constant(molecularmass[i],grav_acc,avogadro,temp,ga
scon);

loop_rho[i]=INIT.rhof(beginrho,exponentfactorrho,r[i]+0.5*deltaR,Rear
th,deltaR,H_constant);

rho[i]=INIT.rhof(beginrho,exponentfactorrho,r[i],Rearth,deltaR,H_cons
tant);
}

//initial some arrays to be used later
double mass[d];
double mass_shell[d-1];
for(int i=0;i<d;i++){
    mass[i]=Mstar;
}
for(int i=1;i<d-1;i++){
mass_shell[i]=0;
}

//calculation of the temperature array
double T[d];
double delta_T[d];
double delta_L[d];
for(int i=0;i<d;i++){
    T[i]=INIT.temp(begintemp);
    delta_T[i]=0;
    delta_L[i]=0;
}

dynamics dyn;
if(wantdensity==1){//check if the user wants to use the red giant-
like element layer division
    //set the boundary radii of the elements layers
    r[0]=12e3;
    r[10]=11020e3;
    r[20]=21738e3;
    r[25]=26395e3;
    r[40]=69476e3;
    r[60]=389066e3;
    r[99]=1000*Rsolar;
    //set the temperatures at the boundaries
    T[0]=334e7;
    T[10]=199e7;
    T[20]=157e7;
    T[25]=87e7;
    T[40]=18.8e7;
    T[60]=3.69e7;
    T[99]=3500;

    //define the radii of shells in between layers and interpolate
the temperature

    for(int k=1;k<10;k++){
        r[k]=r[0]+(r[10]-r[0])/10 * k;
        T[k]=dyn.tempdistr(r[k],T[0],T[10],r[0],r[10]);
        molecularmass[k]=28/15 * Munit;

    }
    for(int k=11;k<20;k++){
        r[k]=r[10]+(r[20]-r[10])/10 * (k-10);
        T[k]=dyn.tempdistr(r[k],T[10],T[20],r[10],r[20]);
        molecularmass[k]=16/9 * Munit;

    }
    for(int k=21;k<25;k++){
        r[k]=r[20]+(r[25]-r[20])/5 * (k-20);
        T[k]=dyn.tempdistr(r[k],T[20],T[25],r[20],r[25]);
        molecularmass[k]=20/11 * Munit;

    }
    for(int k=26;k<40;k++){
        r[k]=r[25]+(r[40]-r[25])/15 * (k-25);
        T[k]=dyn.tempdistr(r[k],T[25],T[40],r[25],r[40]);
        molecularmass[k]=12/7 * Munit;

    }
    for(int k=41;k<60;k++){
        r[k]=r[40]+(r[60]-r[40])/20*(k-40);
        T[k]=dyn.tempdistr(r[k],T[40],T[60],r[40],r[60]);
        molecularmass[k]=4/3 * Munit;

    }
    for(int k=61;k<100;k++){
        r[k]=r[60]+(r[99]-r[60])/40*(k-60);
        T[k]=dyn.tempdistr(r[k],T[60],T[99],r[60],r[99]);
        molecularmass[k]=0.6 * Munit;

    }


    //define the mass of the matter inside each shell
    for(int k=0;k<21;k++){
        mass_shell[k]=0.12 * Msolar/10;
    }
    for(int k=21;k<26;k++){
        mass_shell[k]=0.12 * Msolar/5;
    }
    for(int k=26;k<41;k++){
        mass_shell[k]=0.12 * Msolar/15;
    }
    for(int k=41;k<61;k++){
        mass_shell[k]=0.12 * Msolar/20;
    }
    for(int k=61;k<99;k++){
        mass_shell[k]=18 * Msolar/40;
    }
    //create the array containing enclosed masses
    for(int i=0;i<d-1;i++){
        mass[i+1]=mass[i]+mass_shell[i];

    }

}

//define the mass arrays when the red giant-like element layer
division is not used
if(wantdensity!=1){
    for(int i=0;i<d-1;i++){
        mass_shell[i]=4 * pi/3 * (pow(r[i+1],3)-pow(r[i],3)) *
INIT.rhof(beginrho,exponentfactorrho,r[i]+0.5 *
deltaR,Rearth,deltaR,H_constant);
        mass[i+1]=mass[i]+mass_shell[i];
        //akkk=mass_shell[35]*9
        //mass_shell[35]  10*mass_shell[35]
        //for i in range(35,r_length-1):
        //mass[i+1]=mass[i+1]+akkk
    }
}
//calculate the densities of the stellar matter in case of the red
giant-like element layer division
if(wantdensity==1){
    for(int k=0;k<d-1;k++){
        loop_rho[k]=mass_shell[k]/(4 * pi/3 * (pow(r[k+1],3)-
pow(r[k],3)));
    }
    for(int k=1;k<d-1;k++){
    rho[k]=(loop_rho[k-1]+loop_rho[k])/2;
    rho[d-1]=rho[d-2]/10;
    cout<<"rho "<<rho[k]<<"\n";
    }

}
//initialise the energy array
double energybegin[d];
for(int i=0;i<d;i++){
    energybegin[i]=0;
}
//calculate the pressure inside each shell
double pressure[d];
for(int i=0;i<d;i++){

pressure[i]=dyn.pressure_gen(T[i],rho[i],molecularmass[i],a,boltz);
}
//calculation of the dp over dr array
//rvooder contains the positions of the center of each shell
double rvooder[d];
for(int i=1;i<d;i++){
    rvooder[i]=1;
}
double dp_over_dr[d];
for (int k = 0; k <d-1; ++k) {
    rvooder[k]=r[k]+(r[k+1]-r[k])/2;
    rvooder[d-1]=r[d-1]+0.5*deltaR;
 dp_over_dr[k]=INIT.der(pressure[k],pressure[k-
1],rvooder[k],rvooder[k-1]);
}
//creating the intial speed array, we assume ths is 1m/s at all
radial coordinates
double v[d];
for(int i=0;i<d;i++){
    v[i]=1;
}
//initialise the acceleration array
double accel[d];
for(int i=0;i<d;i++){
    accel[i]=0;
}
//Data storage
double v_total[100][d];
double r_total[100][d];
double rho_total[100][d];
double T_total[100][d];
double dp_over_dr_total[100][d];
double accel_total[100][d];
double P_total[100][d];
double rho_new[d-1];

for(int i=0;i<d;i++){
    v_total[0][i]=v[i];
    r_total[0][i]=r[i];
    rho_total[0][i]=rho[i];
    T_total[0][i]=T[i];
    dp_over_dr_total[0][i]=dp_over_dr[i];
    accel_total[0][i]=accel[i];
    P_total[0][i]=pressure[i];
}
for(int i=0;i<d-1;i++){
    rho_new[i]=0;
}


//define error as 0 , meaning no shell overlap has occured yet
int error;
error=0;

//initialise time storage
vector<double> totime;
totime.push_back(0);

//initialise the number of iterations of the program
int iterations;
iterations=0;
//save the old time step for use in adjusting it later on
double dtold;
dtold=dt;
//set the first element of the array containing the values of the
radii at a previous time step
rold[0]=r[0];
//initialise some other time variables
double timevar;
time_t t1;
timevar=dt;
time(&t1);
tmax=tmax * 60;

//set the initial two shell distances for the red giant like element
layer division situation
double deltaR1=0;
double deltaR2=0;
if(wantdensity==1){
    deltaR1=r[15]-r[14];
    deltaR2=r[35]-r[34];
}
Energy Energ;
//calculate the initial total energy
for(int k=0;k<d-1;k++){
    energybegin[k]=Energ.internalenergy(((4/3) * pi * (pow(r[k+1],3)-
pow(r[k],3))),T[k],mass_shell[k],molecularmass[k],enercon,boltz)\
            -Energ.gravenergy(mass_shell[k],mass[k],r[k],G)\
            +Energ.kineticenergy(mass_shell[k],(v[k+1]+v[k])/2);
```

```
}

//initialise the total energy array that will be changed over the
course of the runtime
double energytotal[100][d];
for(int k=0;k<d-1;k++){
    energytotal[0][k]=energybegin[k];
}
if(atmossim!=1 && wanthundred!=1){
    T[0]=1e9;
    T[1]=8.5e8;
    T[2]=7e8;
    T[3]=6e8;
    T[4]=5e8;
    T[5]=3e8;
    T[6]=1e8;
    T[7]=1e7;
    T[8]=1e6;
    T[9]=1e5;
    rho[0]=10 * rho[0];
    rho[1]=9 * rho[1];
    rho[2]=8 * rho[2];
    rho[3]=7.5 * rho[3];
    rho[4]=8 * rho[4];
    rho[5]=8 * rho[5];
    rho[6]=8 * rho[6];
    rho[7]=8 * rho[7];
    rho[8]=2 * rho[8];

    for(int k=0;k<10;k++){
        cout<<"T :"<<T[k]<<" et rho:" << rho[k]<<"\n";
    }
}

//start the time loop in which all quantities are evaluated step by
step
int i=0;
int l=1;
double damping;
double x;
double DeltaV,A1,B1,C1,D1,E1;
double energy[d-1];
double y;
string filename;
while(timevar<=t_final && error==0){//keep running while max. time
has not been exceeded and no overlap error has occured yet
   totime.push_back(timevar); //update the time array
   dtold=dt;
   //calculation of the new r and v array
   cout<<"dt "<<dt<<"\n";
   damping =1-dampfactor * dt;//calculate the damping
   cout<<"damp "<<damping<<"\n";
    ofstream myfile;

    stringstream b;
    b<<i;
    filename= "SN_"+ b.str();
    filename+= ".csv";
    myfile.open(filename.c_str());
    myfile<<"x,y,z,T\n";
    double stp,x1,y1;
    // cout<<"x,y,z,r,v,dt,accel\n";
    for(int k=0;k<d;k++){
       x1=r[k];
       rold[k]=r[k];//store the old radii
       accel[k]=dyn.acc(mass[k],r[k],rho[k],dp_over_dr[k],G);
       v[k]=v[k]*damping+accel[k] * dt;
       r[k]=r[k]+v[k] * dt+accel[k] * dt * dt;
       if(r[k]<0){
           r[k]=0;
       }
       stp=r[k]/100;
       for(int i=0;i<100;i++){
           x1=x1-stp;
           y1=sqrt(abs(pow(r[k],2)-pow(x1,2)));
       myfile<<x1<<","<<y1<<",0,"<<T[k]<<"\n";
       myfile<<x1<<","<<-y1<<",0,"<<T[k]<<"\n";
       myfile<<-x1<<","<<y1<<",0,"<<T[k]<<"\n";
       myfile<<-x1<<","<<-y1<<",0,"<<T[k]<<"\n";
       }

    }
    myfile.close();
    //check whether shells overlap
    for(int k=1;k<d;k++){
    if(r[k]<r[k-1]){
        cout<<"overlap error\n";
        error=1;
        cout<< "overlap shell"<<k<<"\n";
        break;
        }
    }
    time_t t2;
    time(&t2);
    if((t2-t1)>=tmax){
        error=1; //here error is used to end the loop when max time
has been exceeded
        //if does not mean that there is an error in the program
        cout<< "max run time exceeded\n";
    }

    //storage of v,r,T, rho and dp_over-dr
    for(int j=0;j<d;j++){
        v_total[l][j]=v[j];
        r_total[l][j]=r[j];
        rho_total[l][j]=rho[j];
        T_total[l][j]=T[j];
        dp_over_dr_total[l][j]=dp_over_dr[j];
        accel_total[l][j]=accel[j];
        P_total[l][j]=pressure[j];
    }

    //update the temperature and pressure
    for(int k=0;k<d-1;k++){
        x=mass_shell[k]/(4 * pi/3 * (pow(r[k+1],3)-
pow(r[k],3)));//temporarily store the density in the variable x

        //calculate the terms used in the deltaT equation
        DeltaV=(4/3) * pi * ((pow(r[k+1],3)-pow(r[k],3))-
(pow(rold[k+1],3)-pow(rold[k],3)));
        A1=x * boltz * T[k]/(molecularmass[k]);
        B1=a * pow(T[k],4);
        C1=a/3 * pow(T[k],4);
        D1=(3/2) * boltz * mass_shell[k]/molecularmass[k];
        E1=4 * a * pi * (4/3) * (pow(r[k+1],3)-pow(r[k],3)) *
pow(T[k],3);
        delta_T[k]=(-1 * (A1+B1+C1) * DeltaV)/(D1+E1);  //caculate
delta T
        loop_rho[k]=x; //set the density equal to the temporary
variable x

    }

    for(int k=1;k<d-1;k++){
        delta_L[k]=4 * sig * 4 * pi * (pow(T[k-1],4) *
pow(r[k],2)+T[k+1] * pow(r[k+1],2)-T[k] *
(pow(r[k],2)+pow(r[k+1],2)));
    }
    //update the temperature array

    for(int k=1;k<=d;k++){
        T[k]=T[k]+delta_T[k];
    cout<<"T[k]:"<<T[k]<<"\n";
    cout<<"delta_T[k]:"<<delta_T[k]<<"\n";

    }

    for(int k=0;k<d-1;k++){//update the pressure

pressure[k]=dyn.pressure_gen(T[k],loop_rho[k],molecularmass[k],a,bolt
z);
        if(k==1 && i%200 ==0 ){
        cout<<"timevar"<<timevar<<"\n";//print the time step to show
the user how far the program is
        }
    }

    //initialise the energy array
    //calculate the energy of each shell
    for(int k=0;k<d-1;k++){
        energy[k]=Energ.internalenergy(((4/3) * pi * (pow(r[k+1],3)-
pow(r[k],3))),T[k],mass_shell[k],molecularmass[k],enercon,boltz)-
Energ.gravenergy(mass_shell[k],mass[k],r[k],G)+Energ.kineticenergy(ma
ss_shell[k],(v[k+1]+v[k])/2);

    }
    //update the energy storage
    for(int k=0;k<d;k++){
        energytotal[l][k]=energy[k];
    }

    //calculate the density at shell radii
    for(int k=1;k<d-1;k++){
        rho[k]=(loop_rho[k-1]+loop_rho[k])/2;

    }
    //create the array containing the positions of the centers of the
shells
    for(int k=0;k<d-1;k++){
        rvooder[k]=r[k]+(r[k+1]-r[k])/2;
        rvooder[d-1]=r[d-1]+0.5 * deltaR;
    }
    //calculate the pressure gradients
    for(int k=1;k<d;k++){
    dp_over_dr[k]=INIT.der(pressure[k],pressure[k-
1],rvooder[k],rvooder[k-1]);
    }
    //adjust the time step size
   double rshift[d];
   double vshift[d];
   double newdeltaR[d];
   double rcheck;
   double vcheck;
   double vavg;
   double deltaRavg;
   double dtfactor;

   if(wantdtscaling==1 && wantdensity!=1){

        for(int k=0;k<d;k++){
        rshift[k]=r[k+1];
        vshift[k]=v[k+1];
        }

        for(int k=0;k<d;k++){
        newdeltaR[k]=rshift[k]-r[k];
        }
        double min1,max1;
        min1=newdeltaR[0];
        max1=v[0];
        for(int i=0 ; i<d ; i++){
            if(min1>newdeltaR[i]){
                min1=newdeltaR[i];
          }

            if(max1<v[i]){
                max1=v[i];
          }
        }

        rcheck=min1;
        vcheck=abs(max1);
        double sum1=0,sum2=0;
        for(int i=0;i<d;i++){
            sum1+=v[i];
            sum2+=newdeltaR[i];
        }
        vavg=abs(sum1/d);
        deltaRavg=sum2/d;
        dtfactor=pow((rcheck/deltaR),2);

        if( dtfactor>0.000001){//check if the time step size is
larger than the allowed minimum
        dt=dtold * dtfactor;
        }
        else{
          cout<<"dt reached minimum\n";//tell the user that the time
step has reached its lowest value
            //indicating that the program is highly unstable and
liable to have shell overlap

        }

    }
    if(wantdensity==1 && wantdtscaling==1){//adjust the time step
size when the red giant-like element layer is used
        for(int k=0;k<d;k++){
        rshift[k]=r[k+1];
        newdeltaR[k]=rshift[k]-r[k];

        }
        double rcheck1=0,rcheck2=0,min1,min2,dtfactor1=0,dtfactor2=0;
        min1=newdeltaR[0];
        min2=newdeltaR[25];
        for(int i=0;i<25;i++){

            if(min1>newdeltaR[i]){
                min1=newdeltaR[i];
            }
        }
        for(int i=25;i<99;i++){

            if(min2>newdeltaR[i]){
                min2=newdeltaR[i];
            }
        }
        rcheck1=min1;
        rcheck2=min2;
        for(int k=1;k<26;k++){
            dtfactor1=pow((rcheck1/deltaR1),2);
        }
        for(int k=26;k<99;k++){
            dtfactor2=pow((rcheck2/deltaR2),2);
        }
       if(dtfactor1<dtfactor2){
       dtfactor=dtfactor1;
       }
       else{
       dtfactor=dtfactor2;
       }
       cout<<"dt factor "<<dtfactor<<"\n";
        if(dtfactor>0.00001){
            dt=dtold * dtfactor;
        }
        else{
            cout<<"dt reached minimum1:\n";
        }
    }
    //increase the total time by dt
    timevar=timevar+dt;
    iterations=iterations+1;
    i+=1;
    l+=1;
cout<<i<<"\n";
}

//print the initial and final energies so the user can compare them
 cout<<"Energybegin,EnergyEnd\n ";
 for(int i=0;i<d;i++){
   cout<<energybegin[i]<<","<<energy[i]<<"\n";
}

cout<<"number of iterations: "<<iterations<<" \n";

}
```