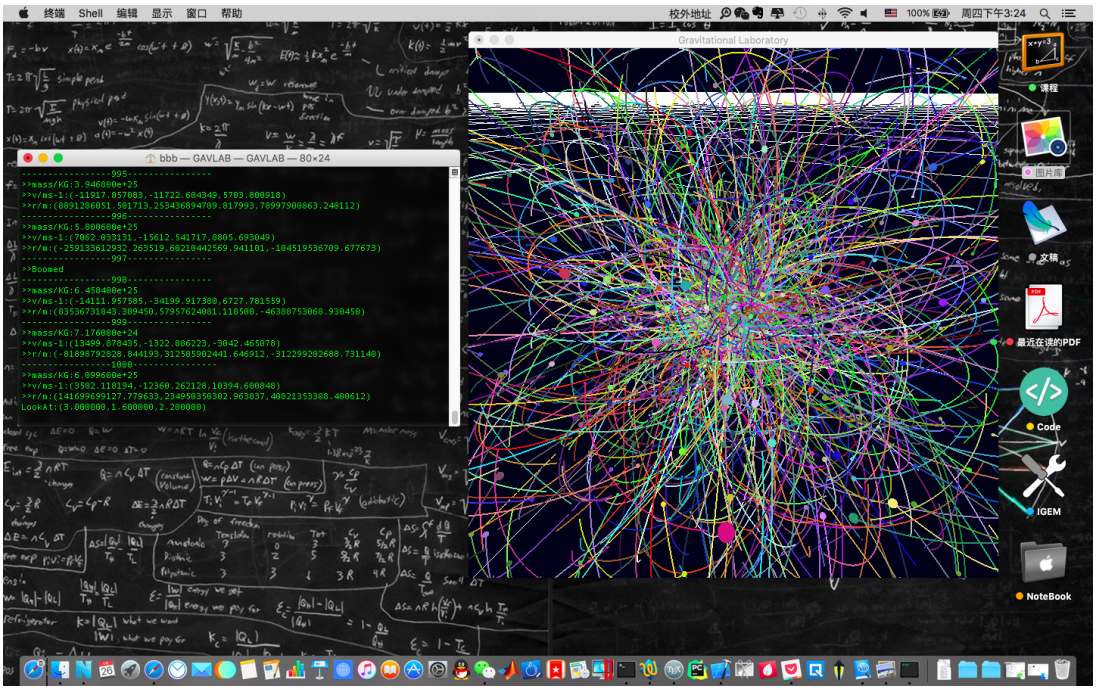
Mr.B's Gravitational Laboratory

1500011362

November 14, 2016



Contents

1 Welcome to my Gravitational laboratory

Hello,welcome to Mr.B's **G**r**AV**itational **LAB**oratory or you can call it **GAVLAB**.

1.1 What can you do in GAVLAB

Please don't be confused by its name because GAVLAB is not a real laboratory. Actually it's a virtual laboratory running on computer, a program which can simulate the physical process or a system driven by gravitational force.

You can simulate any gravitational process because GAVLAB allows the users to design their own system in the universe. In this **manual mode** users have to set up all information of the system,including the planet's name,mass,radius,position,velocity and RGB color. It's an exhausting task!

Besides manual mode there are another two automodes for the users who are too busy to set up the system manually.

The first choice is to simulate the **real solar system** with eight planets in it. All data was provided by **Professor Xue** on the website.

The other choice is a little crazy. It's **random mode**. Random mode can generate any number of planets randomly in random position with random velocity and random mass and even random color. Everything is random! It means we can simulate hundreds of planets by typing several keys on the keyboard and then enjoy the universe and drink a cup of coffee. It's convenient!

1.2 How to run GAVLAB

The first step of running GAVLAB is easy:get a computer. More specifically, a computer with Linux based system such as MAC with OSX system.

Unfortunately GAVLAB cannot run on Windows system because there are a lot of libraries missing which I will list out later.

Anyway Mac is the best choice. On a Mac computer all you have to do is double clicking the executable file named GAVLAB with the sun-shaped logo.

Figure 1: All you have to do to run GAVLAB is double clicking the sun logo

1.3 An illustration of GAVLAB

It doesn't matter if you don't have a computer with Linux based system because I will demonstrate to you what GAVLAB looks like when it is running and some results of the experiment in this lab.

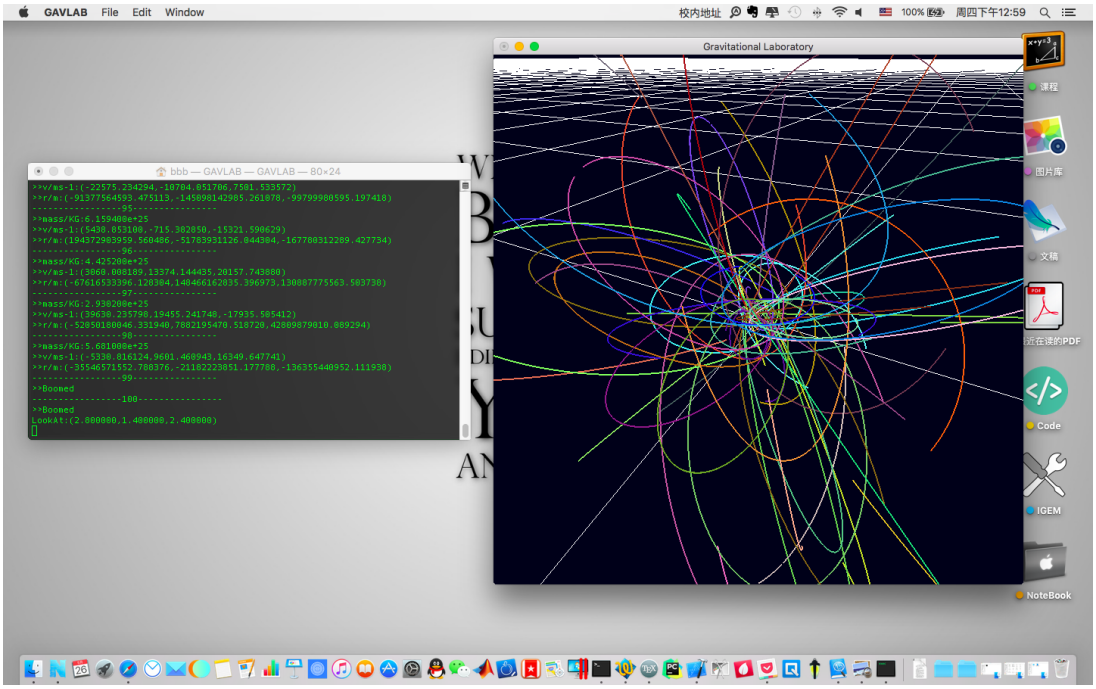


Figure 2: GAVLAB is simulating a system with 100 planet

Figure 1 is the result of simulation of 100 planet. As you can see those round balls are planets and the colorful lines are their trajectories.

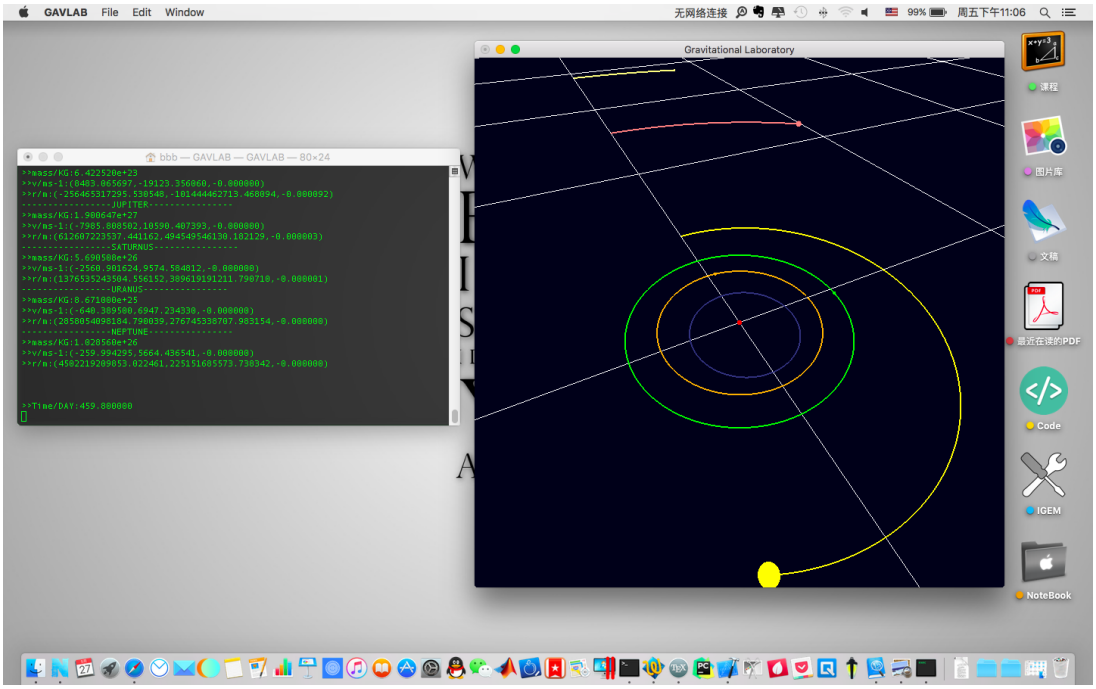


Figure 3: GAVLAB is simulating a real solar system

Figure 2 showed the real solar system to you. It's worth to notice that the trajectory can be really stable as time goes by. Especially for mercury which it is very close to sun.

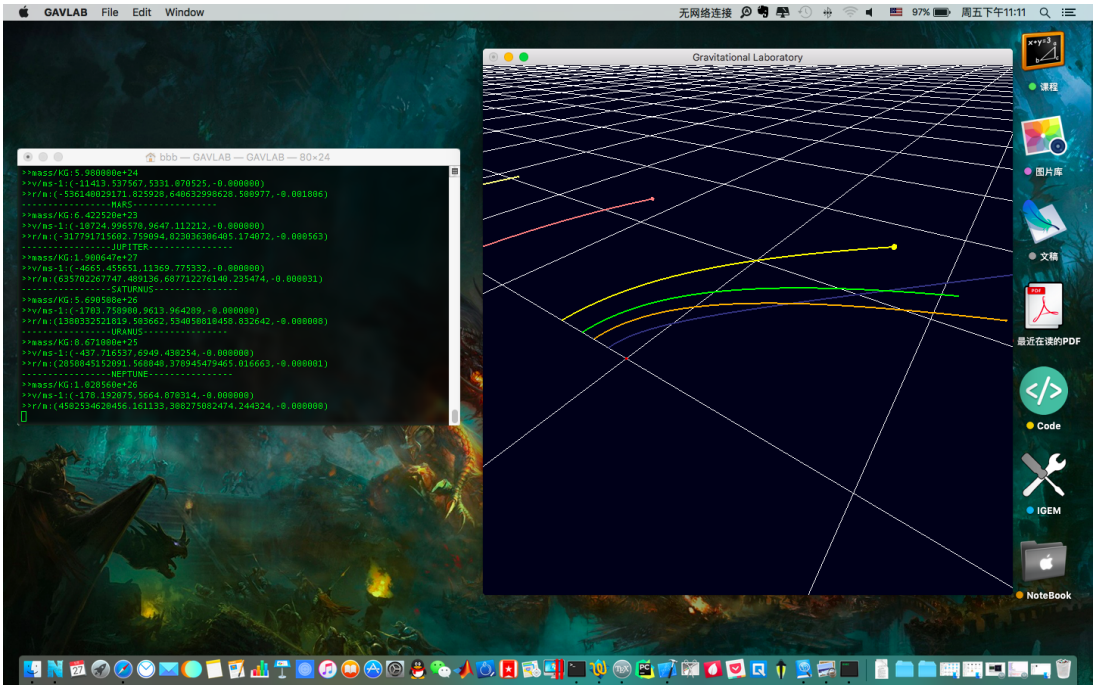


Figure 4: GAVLAB is simulating a half solar system

Figure 3 tells you what will happen if half mass of sun dispair. As you can see some planets will leave the solar system and never come back.

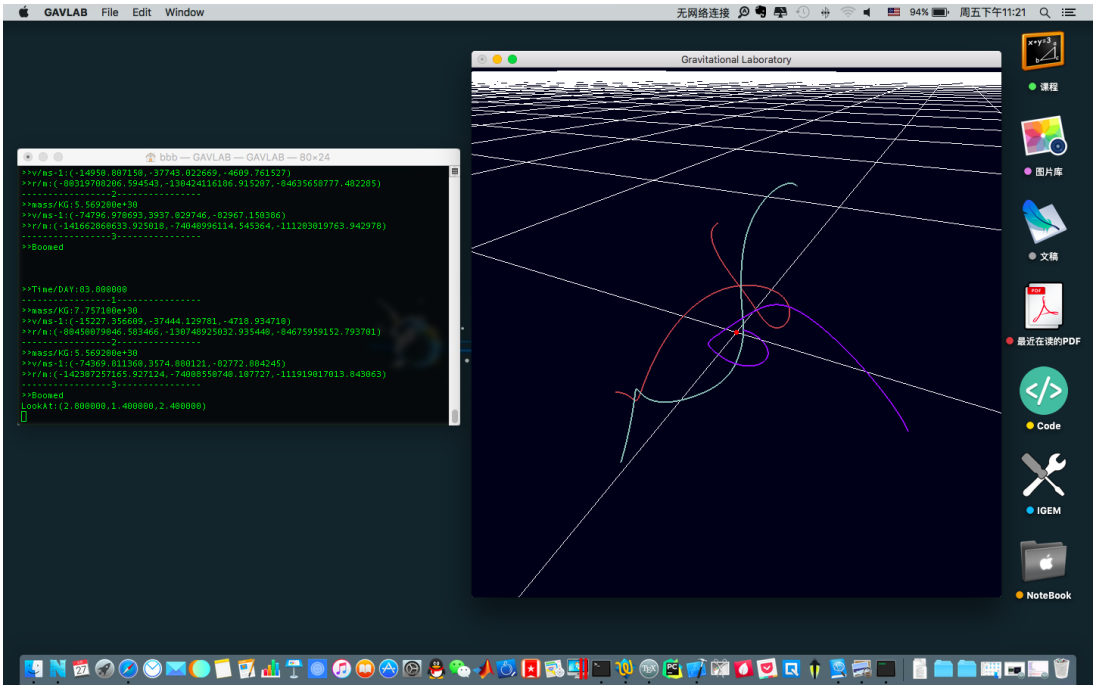


Figure 5: GAVLAB is simulating a 3-body system

Figure 4 simulated the 3-body system with equal mass. What a mess!

1.4 How was GAVLAB built

For geeks and those who love computers and physics I will give you a short summary of how I built GAVLAB.

1.4.1 Xcode project

GAVLAB is built on Xcode,an IDE for IOS developers. On Xcode I can write program in C,C++,objective-C or swift.

The complier of Xcode is Clang developed by Apple Inc.

Xcode is really convenience for programing because it contains lots of functions such as debugging,breakpoints,code auto complement and beautiful syntax highlight.

Aslo Xcode contains all kinds of libraries such as OpenGL or OpenCL which saved my time setting my developing environment.

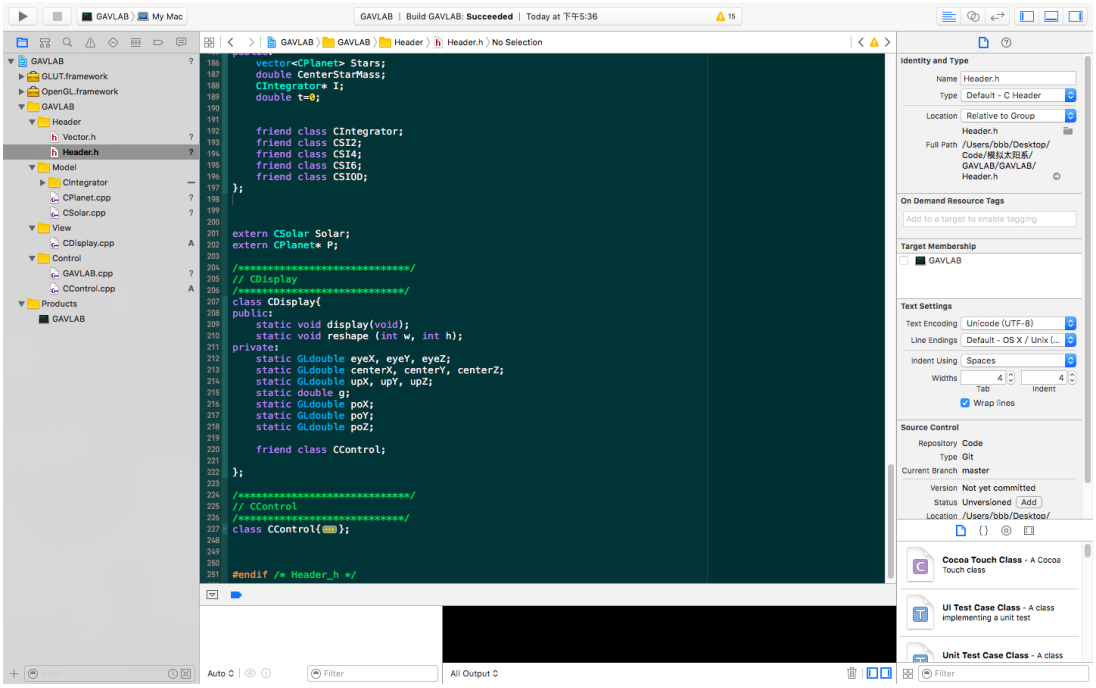


Figure 6: GAVLAB is an Xcode project

1.4.2 Summary

GAVLAB was built in C++ language with a little Object-Oriented programing technique. It contains **data abstraction**,**inheritance** and **dynamic binding**.

In GAVLAB each planet is a CPlanet object and the whole solar is a CSolar object which is **data abstraction**.

There are several kinds of integrators in GAVLAB. Each of them is an object of class derived from CIntegrator class which is **inheritance**.

Different integrators share the same method **Update** with different implementation code which is **dynamic binding**.

Also I used **class template** when I was building my math library Vector which is **generic programing**.

The structure of GAVLAB is **MVC** where I imitated from the software developing.

Control class takes in charge of the whole program and interact with users in command window.

Model is actually math. To make our trajectory stable and correct I used **Symplectic Integrator** with two order,four order and six order. They can be notated as **SI2**,**SI4** and **SI6**.

View method makes GAVLAB visual with OpenGL which is an open graphics API.

2 What do we need to build GAVLAB

In this section I will analyze the requirement of building GAVLAB.

- Data Structure
  - We need to store data information of the planet.
  - We need to divide the space into several grids
- Model
  - We need to make the integrator accurate.
  - We need to make the energy of the system conserve.
  - We need to deal with the collision circumstance.
  - We need to make GAVLAB able to simulate a large amount of planet.
- Design
  - We need to make GAVLAB able to interact with users.
  - We need to visualize the result of GAVLAB.
  - We need to make animation while simulating

3 An overview of GAVLAB's structure

3.1 MVC

MVC is Model View and Controller. Model for GAVLAB here is just simulating the gravitational process. View is to let the users see the result of the program. And Controller is the commander which takes the command of users and tells the program what to do.

In GAVLAB there will be a loop. At the beginning of each loop the Controller will get the command of the users such as starting simulation or pause. Then it parses the command and execute the command.

After that the model will simulate the physics process according the integrator and update the state.

Finally View is responsible for making the animation according the data generated by model.

3.2 Illustration

This is the illustration figure of the structure.

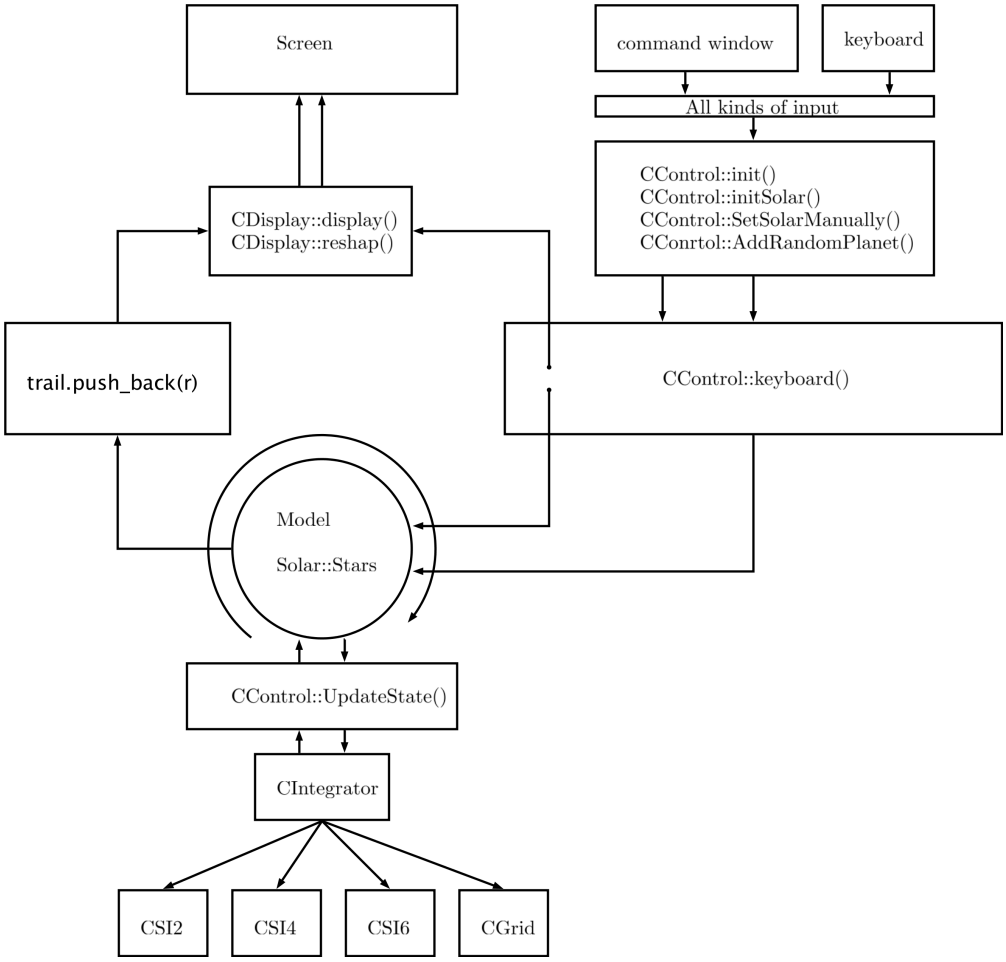


Figure 7: GAVLAB has a frame similar to MVC

3.3 Structure of program

C **Header** Folder contains two header files

- ⊞ **Header.hpp** Header file contains declarations of all the classes
  - ▶ **class CIntegrator** Base class of all integrators
- ⊞ **Vector.hpp** Math Library contains all kinds of vector operation
  - ▶ **class Vector**$< classT >$ Vector template

C **Model** Folder contains model related files

- C **CIntegrator** Folder contains all the integrators
  - ⊞ **CSI2.cpp** 2-order Symplectic Integrator
    - ▶ **class CSI2**
  - ⊞ **CSI4.cpp** 4-order Symplectic Integrator
    - ▶ **class CSI4**
  - ⊞ **CSI6.cpp** 6-order Symplectic Integrator
    - ▶ **class CSI6**
  - ⊞ **CGrid.cpp** Grid method
    - ▶ **class CGrid**
- ⊞ **CPlanet.cpp** File contains definition of CPlanet class
  - ▶ **class CPlanet** class to store Planet's information
- ⊞ **CSolar.cpp** Folder contains definition of CSolar class
  - ▶ **class CSolar** class to represent the whole solar system

C **View** Folder contains View related files

- ⊞ **CDisplay.cpp** File contains definition of CDisplay class
  - ▶ **class CDisplay** Contains OpenGL code to make animation

C **Control** Folder contains Controller related files

- ⊞ **CControl.cpp** File contains definition of CControl class
  - ▶ **class CControl** class control the input and parse command
- ⊞ **GAVLAB.cpp** File contains main function
  - ▶ **int main()** main function runs the Loop

4 MODEL is actually math

4.1 Math Library Vector

We are physicists and we prefer writing our equation in concise form. So there will be a lot of vector and matrix operation in our equations and formulas. Especially in Newton's Law and the formula of gravity force there are vectors.

For that reason it will save us a lot of time if we define our own **Vector** operation library [1].Here's my code of class template Vector

```
template<class T>
class Vector{
public:
    explicit Vector(T a=0,T b=0,T c=0):x{a},y{b},z{c
       }{};
    Vector(const Vector<T>& u)=default;
    Vector(Vector<T> && u)=default;
    Vector<T>& operator=(const Vector<T>&u)=default;
    Vector<T>& operator=(Vector<T>&&u)=default;
    T Magnitude(void);
    Vector<T>  Normalize(void);
    Vector<T>  Reverse(void);
    // vector operation
    Vector<T>& operator+=(const Vector<T> &u);
    Vector<T>& operator-=(const Vector<T> &u);
    // scaler operation
    Vector<T>& operator*=(T s);
    Vector<T>& operator/=(T s);
    bool operator ==(const Vector<T> &u);
    bool operator !=(const Vector<T> &u);
    inline T X()const{return x;}
    inline T Y()const{return y;}
    inline T Z()const{return z;}
```

CODE 1 : Vector Library I

```
friend Vector<T> operator+<>(const Vector<T>& u,
   const Vector<T> &v);
friend Vector<T> operator-<>(const Vector<T>& u,
   const Vector<T> &v);
friend Vector<T> operator^<>(const Vector<T>& u,
   const Vector<T> &v);
friend T operator*<>(const Vector<T>& u,const Vector
   <T> &v);
friend Vector<T> operator*<>(T s, const Vector<T>& u
   );
friend Vector<T> operator*<>(const Vector<T>& u, T s
   );
friend Vector<T> operator/<>(const Vector<T>& u, T s
   );
friend ostream& operator<< <>(ostream &out,const
   Vector<T>& u);
private:
    T x,y,z;
};
```

CODE 2 : Vector Library II

Those methods are just ordinary math formula for basic vector operation which is very trivial. So I will not show the implementations of those methods. Instead I will show my type definition of template class Vector

```
// Vector with three float type variables
typedef Vector<float> vector3f;
// Vector with three double type variables
typedef Vector<double> vector3d;
```

CODE 3 : typedef

4.2 CPlanet class

To store and manage the data of each planet we can abstract the planet to be an object. So I defined CPlanet class.

```
class CPlanet{
public:// Constructor
    CPlanet(string name,
            double mass,
            double radius,
            double x,double y,double z,
            double vx,double vy,double vz,
            double Red,double Green,double Blue):
Name(name), Mass(mass),Radius(radius),r(vector3d(x,y
   ,z)),v(vector3d(vx,vy,vz))
,Color(vector3d(Red,Green,Blue)){exist=true;}
    // Get the attribute
    inline const vector3d& GetR()const{return r;}
    inline const vector3d& GetV()const{return v;}
    inline const double X()const{return r.X();}
    inline const double Y()const{return r.Y();}
    inline const double Z()const{return r.Z();}
    double GetMass()const{return Mass;}
    inline string GetName()const{return Name;}
    inline const double Red()const{return Color.X()
       ;}
    inline const double Green()const{return Color.Y
       ();}
    inline const double Blue()const{return Color.Z()
       ;}
    inline const bool Exist()const{return exist;}
    inline const double GetRadius()const{return
       Radius;}
```

CODE 4 : CPlanet class I

```
    // Set the attribute
    void SetR(double a,double b,double c){r=vector3d
       (a,b,c);}
    void SetV(double a,double b,double c){v=vector3d
       (a,b,c);}
    // update the state
    void Updatetr(double DT){trail.push_back(r);}
    // list storing trajectory
    list<vector3d> trail;
private:
    double Mass;
    string Name;
    double Radius;
    vector3d r;
    vector3d v;
    vector3d Color;
    bool exist; // false if it explode

    friend class CSolar;
    friend class CIntegrator;
    friend class CSI2;
    friend class CSI4;
    friend class CSI6;
    friend class CGrid;
};
```

CODE 5 : CPlanet class II

Not too much complicated method for CPlanet because it's just a data structure to store the data. It contains the name,mass,radius,position,velocity,color and a bool type variable representing if it exists.

In GAVLAB we don't really delete the CPlanet object if it has exploded because we want to plot the trajectory of every planet and research how it exploded or which planet it collided with. So we just use a bool type flag called exist to mark that information.

4.3 CSolar class

We have a lot of operation aimed at the whole solar system so it's good to abstract that system as an object too. We defined CSolar system to store all CPlanet and the complicated method or algorithm interface.

```
class CSolar{
public:// Constructor
    CSolar(double mass):CenterStarMass(mass){ }
    CSolar(){}
    // Set the attribute
    void SetMass(double mass){CenterStarMass=mass;}
    void SetI(int command);
    void SetStep(double DT_){I->SetDT(DT_);}
    // Get the attribute
    double GetMass()const {return CenterStarMass;}
    double GetT()const{return t;}
    bool AddPlanet(CPlanet &P);// Method
    void GenerateRandomPlanet();// Method
    void UpdateSolar();// Method
public:
    vector<CPlanet> Stars;
    double CenterStarMass;
    CIntegrator* I;
    double t=0;
    friend class CIntegrator;
    friend class CSI2;
    friend class CSI4;
    friend class CSI6;
    friend class CGrid;
};
```

CODE 6 : CSolar class

The CSolar class has a vector member containing all the planet object in it. Besides that we need to store the center mass of the solar for our convenience of adjusting that.

The most important member I is the integrator which I'll explain later. For now we just know each solar can have one integrator to update the state and they are all inherited from the base class CIntegrator.

```
void CSolar::SetI(int command){
    switch (command) {
        case SI2:
            I = new CSI2;
            break;
        case SI4:
            I = new CSI4;
            break;
        case SI6:
            I = new CSI6;
            break;
        case Grid:
            I = new CGrid;
            break;
        default:
            break;
    }
}
```

CODE 7 : SetI()

here SI2 SI4 ... are macro defined int type variables in Header.h

Also adding a planet into the solar is an important method.

```
bool CSolar::AddPlanet(CPlanet &P){
    Stars.push_back(P);
    return true;
}
```

CODE 8 : AddPlanet()

To generate random solar system we need to generate random planet to the solar.

```
int Number=1;
void CSolar::GenerateRandomPlanet(){
    srand((unsigned)time(0));
    char S[10];
    sprintf(S, "%d",Number++);
    string name=S;
    double mass=double(arc4random()%100+10);
    mass*=(ME/10);
    double radius=double(arc4random()%10+1);
    radius/=10.0;
    double x=double(arc4random()%20+1)*flag();
    double y=double(arc4random()%20+1)*flag();
    double z=double(arc4random()%20+1)*flag();
    x*=(0.1*RA);y*=(0.1*RA);z*=(0.1*RA);
    double vx=double(arc4random()%100+10)*flag();
    vx*=100;
    double vy=double(arc4random()%100+10)*flag();
    vy*=100;
    double vz=double(arc4random()%100+10)*flag();
    vz*=100;
    double r=double(arc4random()%100+1);
    r/=100.0;
    double b=double(arc4random()%100+1);
    b/=100.0;
    double g=double(arc4random()%100+1);
    g/=100.0;
    P = new CPlanet(name,mass,radius,x,y,z,vx,vy,vz,
       r,g,b);
    AddPlanet(*P);
}
```

CODE 9 : GenerateRandomPlanet()

Further more we have to Update the solar system.

```
void CSolar::UpdateSolar(){
    I->Update(*this);// call method of integrator
    // check for collision
    vector<CPlanet>::iterator itri,itrj;
    for (itri=Stars.begin(); itri != Stars.end();
       itri++) {
        if (itri->exist == false) {continue;}
        double R0=(itri->r^(itri->v/(itri->v.
           Magnitude()))).Magnitude();
        if (R0<E) {// if it passed the sun
            cout<<itri->Name<<" BOOM"<<endl;
            itri->exist=false;
        }
        for (itrj=Stars.begin(); itrj!=Stars.end();
           itrj++) {
            if (itrj == itri || itrj->exist==false)
               {
                continue;
            }
            vector3d rij=itri->r-itrj->r;
            double R=rij.Magnitude();
            if(R < E){// if the planets are close
                cout<<itri->Name<<" and "<<itrj->
                   Name<<" BOOM!"<<endl;
                itri->v=(itri->Mass*itri->v+itrj->
                   Mass*itrj->v)/(itri->Mass+itrj->
                   Mass);
                itri->Mass+=itrj->Mass;
                itrj->exist=false;
            }
        }
    }
}
```

CODE 10 : UpdateSolar()

4.4 Integrator is important

As we know almost all GAVLAB has to do is simulating. Because the real physics world is driven by differential equation such as Newton's Law or Hamiltonian equation, GAVLAB has to integrate those equations to get the solution. So we need a good integrator.

There's something different between computer simulation and real world's physical process. In computer we can only solve **difference equations** rather than **differential equations** because computers can only deal with decrete math. So instead of calculating infinitesimal step time we have to calculate the new state of the system after some pretty small but finite time step τ.Then if τ is too large it will lose accuracy.

There're some good integrators such as Euler method or Runge-Kutta method. However those methods are not suitable in GAVLAB cause we want our trajectory stable. It's easy to know that the deviation of energy will increase as time goes through using Euler or Runge-Kutta method which means the radius of planet's trajectory will decrease slowly and finally move to sun.

So in GAVLAB I used **Symplectic Integrating** [2] method which is able to maintain the **symplectic proporty** of the planet's canonical coordinate and canonical momentum (q, p).

4.5 CIntegrator

Here is the code of CIntegrator,base class of all the integrators.

```
class CIntegrator{
protected:
    double DT;//time step
public:
    virtual void Update(CSolar &S)=0;
    void SetDT(double dt){DT = dt;}
    double GetDT()const{return DT;}
};
```

CODE 11 : CIntegrator

In CIntegrator class we have pure virtual function Update which means CIntegrator is abstract class.

4.6 Quick description of Symplectic Integrator

It is easy to know that the gravitational system is separable Hamilton system which means Hamiltonian $H(p,q)$ can be written as

$$H(p,q) = T(p) + V(q) \tag{1}$$

And we have Hamilton equation of motion:

$$\begin{aligned} \dot{p} &= -\frac{\partial V}{\partial q} = \varphi(q) \\ \dot{q} &= \frac{\partial T}{\partial p} = \phi(p) \end{aligned} \tag{2}$$

So according to Hamilton equation's difference form we give the difference equation of Symplectic Integrator here:

$$\begin{aligned} q(t_k) &= q(t_{k-1}) + \tau \sum_{i=1}^{m} c_i \phi_{i-1} \\ p(t_k) &= p(t_{k-1}) + \tau \sum_{i=1}^{m} d_i \varphi_i \end{aligned} \tag{3}$$

Where m is the order of integration and c_i d_i are some specific coefficients.

The form of ϕ_{i-1} and φ_i is

$$\begin{aligned} \phi_{i-1} &= \phi(p^{i-1}) \\ \varphi_i &= \varphi(q^i) \end{aligned} \tag{4}$$

And we used m assist points (p^i, q^i) in the difference form equation. Here is how they are represent

$$\begin{aligned} q^i &= q^{i-1} + \tau c_i \phi(p^{i-1}) \\ p^i &= p^{i-1} + \tau d_i \varphi(q^i) \quad (i = 1, 2 \ldots m) \end{aligned} \tag{5}$$

particularly here we have

$$\begin{aligned} p^0 &= p(t_{k-1}) \quad p^m = p(t_k) \\ q^0 &= q(t_{k-1}) \quad q^m = q(t_k) \end{aligned} \tag{6}$$

Now I'll show you the equations and code of CSI2,CSI4 and CSI6.

4.6.1 CSI2

This is the definition of class CSI2

```
class CSI2:public CIntegrator{
public:
    void Update(CSolar &S);
};
```

CODE 12 : CSI2 class

Actually SI2 is just Leap Frog algorithm.

$$
\begin{aligned}
q^* &= q(t_{k-1}) + \frac{\tau}{2}\left(\frac{\partial T}{\partial p}\right)_{p=p(t_{k-1})} \\
p(t_k) &= p(t_{k-1}) - \frac{\tau}{2}\left(\frac{\partial V}{\partial q}\right)_{q=q^*} \\
q(t_k) &= q(t_{k-1}) + \frac{\tau}{2}\left(\frac{\partial T}{\partial p}\right)_{p=p(t_k)}
\end{aligned}
\tag{7}
$$

here $c_1 = c_2 = \frac{1}{2} \quad d_1 = 1 \quad d_2 = 0$

```
void CSI2::Update(CSolar &S){
    vector<CPlanet>::iterator itri,itrj;
    for (itri = S.Stars.begin(); itri != S.Stars.end
       (); itri++) {
        if (itri->Exist() == false) {continue;}
        vector3d r_;
        r_ = itri->r+0.5*DT*(itri->v);
        vector3d v_=itri->v;
        double R_ = r_.Magnitude();
        v_ -=DT*G*S.GetMass()*r_/(R_*R_*R_);
        for (itrj = S.Stars.begin(); itrj != S.Stars
           .end(); itrj++) {
            if(itri == itrj || itrj->Exist() ==
               false) continue;
```

CODE 13 : CSI2 Part I

```
                vector3d rij = r_ - itrj->r;
                double R = rij.Magnitude();
                v_ -= DT*G*itrj->GetMass()*rij/(R*R*R);
            }
            itri->v = v_;
            itri->r +=0.5*DT*v_;
            itri->Updatetr(DT);
        }
        S.t+=DT;
}
```

CODE 14 : CSI2 Part II

And here's the how SI2 works

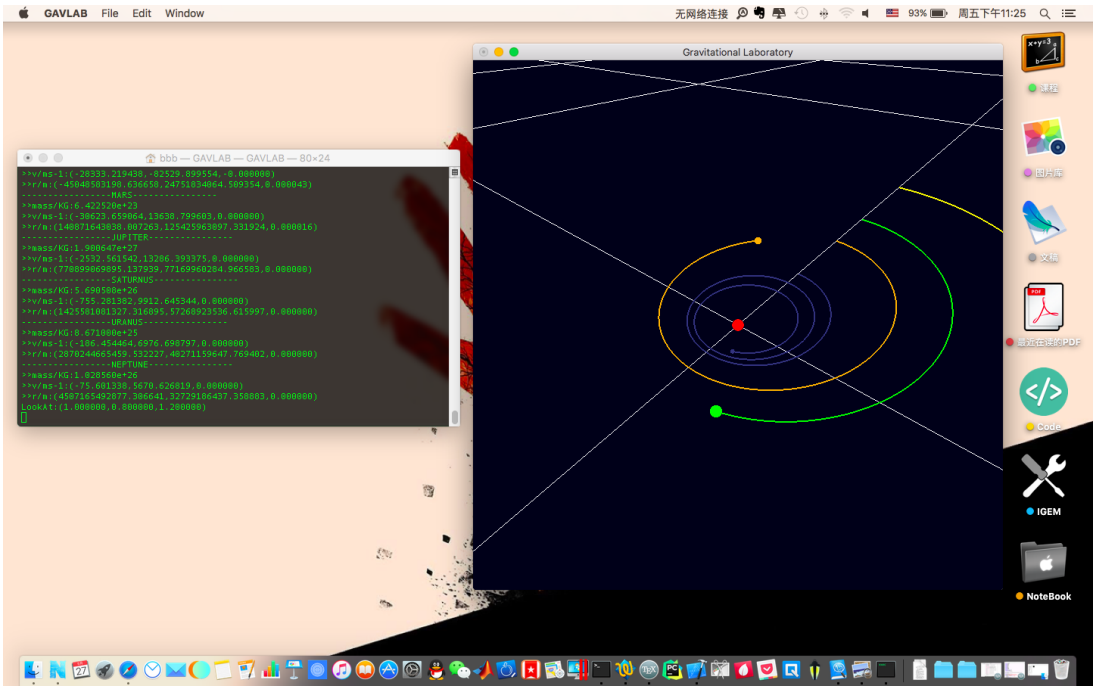


Figure 8: GAVLAB is simulating solar system with SI2

As you can see SI2(LeapFrog method) is not pretty stable for trajectory. The energy of planet will decrease as time goes by and finally fall into sun. So we have to increase the order of integration to get more precise solution.

4.6.2 CSI4

This is the definition of class CSI4

```
class CSI4:public CIntegrator{
public:
    void Update(CSolar &S);
};
```

CODE 15 : CSI4 class

The formula of SI4 was derived by Neri [3] in 1987 and the difference equation is

$$\begin{aligned} q^i &= q^{i-1} + \tau c_i \frac{\partial T}{\partial p}\left(p^{i-1}\right) \\ p^i &= p^{i-1} - \tau d_i \frac{\partial V}{\partial q}\left(q^i\right) \quad i = 1, 2 \ldots 4 \end{aligned} \tag{8}$$

here the coefficient is

$$c_1 = c_4 = \frac{1}{2(2-\sqrt[3]{2})}, \quad c_2 = c_3 = \frac{1-\sqrt[3]{2}}{2(2-\sqrt[3]{2})}$$
$$d_1 = d_3 = \frac{1}{2-\sqrt[3]{2}}, \quad d_2 = c_3 = -\frac{\sqrt[3]{2}}{2-\sqrt[3]{2}}, \quad d_4 = 0$$

Here's code

```
void CSI4::Update(CSolar &S){

    double c[5]={0,0.67560359597,-0.1756035979,
    -0.1756035979,0.67560359597};
    double d[5]={0,1.3512071919596577,
    -1.70241438391932,1.3512071919596577,0};
    vector<CPlanet>::iterator itri,itrj;
    for (itri = S.Stars.begin(); itri != S.Stars.end
       (); itri++) {
        if (itri->exist==false) {continue;}
```

CODE 16 : CSI4 Part I

```
        vector3d r_[5],v_[5];
        r_[0]=itri->r;
        v_[0]=itri->v;
        for (int i=1; i<=4; i++) {
            r_[i]=r_[i-1] + DT*c[i]*v_[i-1];
            // Center gravity
            double r=r_[i].Magnitude();
            v_[i] = v_[i-1]-G*DT*d[i]*S.GetMass()*r_
               [i]/(r*r*r);
            if(i == 4) continue;
            // gravity between stars
            for (itrj = S.Stars.begin();itrj!= S.
               Stars.end(); itrj++) {
                if (itri == itrj) continue;
                if (itrj->exist==false) {
                    continue;
                }
                vector3d rij = r_[i]-itrj->r;
                double R = rij.Magnitude();
                v_[i]-= DT*d[i]*G*itrj->GetMass()*
                   rij/(R*R*R);
            }
        }
        itri->r = r_[4];
        itri->v = v_[4];
        itri->Updatetr(DT);
    }
    S.t+=DT;
}
```

CODE 17 : CSI4 Part II

Let's see if SI4 can get better solution than SI2 did

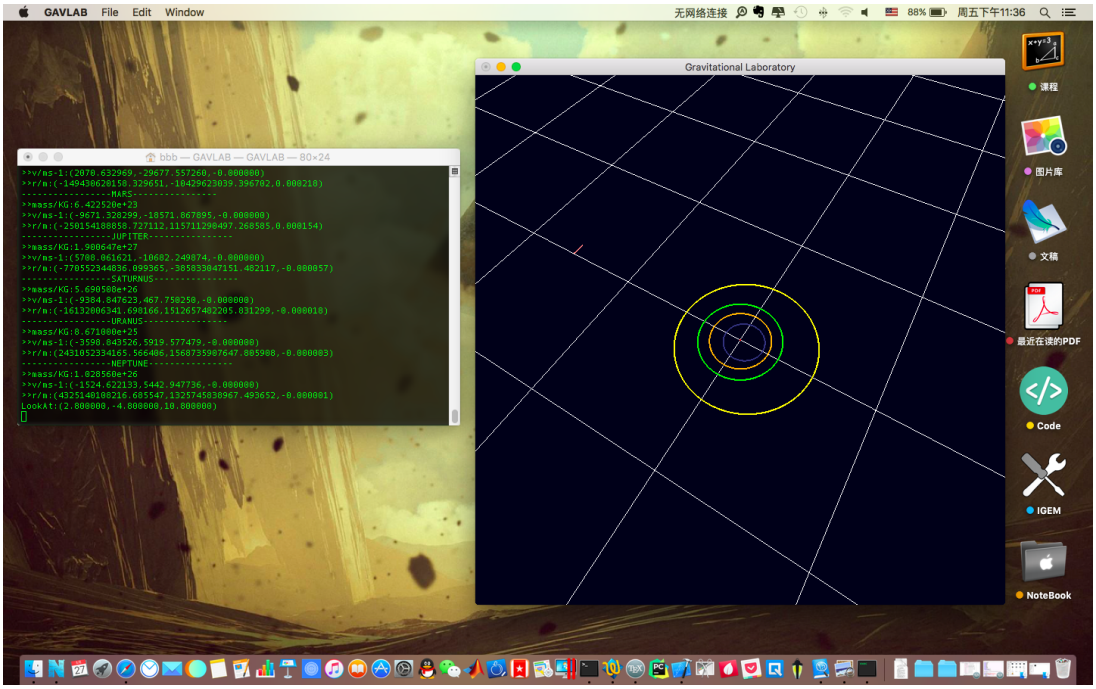


Figure 9: GAVLAB is simulating solar system with SI4

So it's clearly that SI4 can handle the gravitational system simulation. The trajectory of mercury venus and earth can maintain perfect ellipse after long time.

4.6.3 CSI6

This is the definition of class CSI6

```
class CSI4:public CIntegrator{
public:
    void Update(CSolar &S);
};
```

CODE 18 : CSI6 class

The formula of SI6 was derived by Yoshida [4] in 1990 and the difference equation is

$$\begin{aligned} q^i &= q^{i-1} + \tau c_i \frac{\partial T}{\partial p}\left(p^{i-1}\right) \\ p^i &= p^{i-1} - \tau d_i \frac{\partial V}{\partial q}\left(q^i\right) \quad i = 1, 2 \ldots 10 \end{aligned} \tag{9}$$

here the coefficient is

$$\begin{aligned} x_0 &= -\frac{\sqrt[3]{2}}{2-\sqrt[3]{2}}, \quad x_1 = \frac{1}{2-\sqrt[3]{2}} \\ y_0 &= -\frac{\sqrt[5]{2}}{2-\sqrt[5]{2}}, \quad y_1 = \frac{1}{2-\sqrt[5]{2}} \\ d_1 &= d_3 = d_7 = d_9 = x_1 y_1, \quad d_2 = d_9 = x_0 y_1 \\ d_4 &= d_6 = x_1 y_0, \quad d_5 = x_0 y_0 \\ c_1 &= \frac{1}{2} d_1, \quad c_{10} = \frac{1}{2} d_9 \\ c_i &= \frac{1}{2}\left(d_{i-1} + d_i\right) \quad i = 2, 3, \ldots, 9 \end{aligned}$$

```
void CSI6::Update(CSolar &S){
    double c[11];
    double d[11];
    d[1]=d[3]=d[7]=d[9]=-0.293667939522341;
    d[2]=d[8]=-1.99977809735512;
    d[4]=d[6]=0.337335879044682;
    d[5]=2.29714181079093;
    d[0]=d[10]=0;
```

CODE 19 : CSI6 Part I

```
    c[0]=0;
    c[1]=0.5*d[1];
    c[10]=0.5*d[9];
    for (int i=2; i<10; i++) {
        c[i]=(d[i-1]+d[i])/2.0;
    }
    vector<CPlanet>::iterator itri,itrj;
    for (itri = S.Stars.begin(); itri != S.Stars.end
       (); itri++) {
        vector3d r_[11],v_[11];
        r_[0]=itri->r;
        v_[0]=itri->v;
        for (int i=1; i<=10; i++) {
            r_[i]=r_[i-1] + DT*c[i]*v_[i-1];
            // Center gravity
            double r=r_[i].Magnitude();
            v_[i] = v_[i-1]-G*DT*d[i]*S.GetMass()*r_
               [i]/(r*r*r);
            if(i == 10) continue;
            // gravity between stars
            for (itrj = S.Stars.begin();itrj!= S.
               Stars.end(); itrj++) {
                if (itri == itrj) continue;
                vector3d rij = r_[i]-itrj->r;
                double R = rij.Magnitude();
                v_[i]-= DT*d[i]*G*itrj->GetMass()*
                   rij/(R*R*R);
            }
        }
        itri->r = r_[10];
        itri->v = v_[10];
        itri->Updatetr(DT);
    }
    S.t+=DT;
}
```

CODE 20 : CSI6 Part II

4.7 How to simulate thousands of planets or more

The integrating method introduced above do a really good job when simulating hundreds of planets. However when the number of planets grows to thousands or more,the animation will be extremely slow.

The reason is quite simple. When we integrating we have to update the velocity of the planets,thus we have to calculate the gravitational force on it. Since each two planets has the gravitational interaction,we have to calculate the gravitational force between each two planets. So as the result if there are N planets in the system,the running time for each updating step will be $O(N^2)$. That running time will grow fast when N become large.

Some better techniques must occur to solve this problem. And the **Particle mesh technique and FFT** is one of the most used method to simulate the cluster system. Here FFT means Fast Fourier Transformation. So GAVLAB will use that method to simulate solar when it contains too many planets.

4.7.1 Particle mesh technique

To reduce the running time we must abandon some forces between some planets. One way to do this is to divide the system into several meshes or girds with some cells and we treat the particles in the same cell equally.

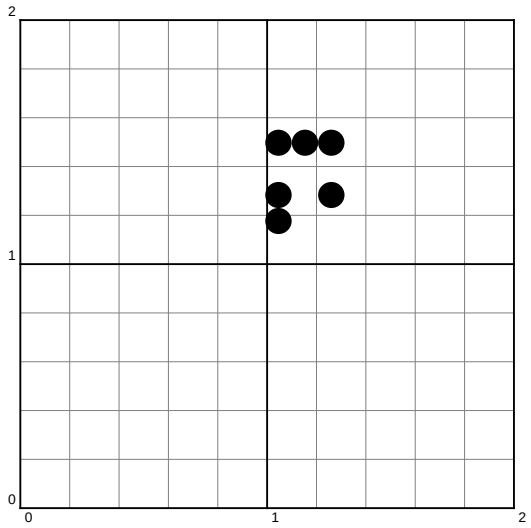


Figure 10: Divide the system into some cells

Then what we consider in the cell is the gravitational potential Φ_α,assuming there are J cells in the grid and the potential of cell α is

$$\Phi_\alpha = \sum_{\beta=1}^{J} g_{\alpha,\beta} M_\beta \tag{10}$$

here M_β is the total mass of the cell β and $g_{\alpha,\beta}$ is the gravitational potentail between cell β and α for unit mass.So we can get

$$g_{\alpha,\beta} = -\frac{G}{\sqrt{\left(\vec{r_\alpha} - \vec{r_\beta}\right)^2}} \tag{11}$$

After we get our potentail of each cell in the grid we can get the difference equation on each planet in each cell

$$\begin{aligned}
V_x\left(t+\frac{1}{2}\tau\right) &= V_x\left(t-\frac{1}{2}\tau\right) + \frac{1}{2}\left(\Phi_{p+1,q,r} - \Phi_{p-1,q,r}\right) \\
V_y\left(t+\frac{1}{2}\tau\right) &= V_y\left(t-\frac{1}{2}\tau\right) + \frac{1}{2}\left(\Phi_{p,q+1,r} - \Phi_{p,q-1,r}\right) \\
V_z\left(t+\frac{1}{2}\tau\right) &= V_z\left(t-\frac{1}{2}\tau\right) + \frac{1}{2}\left(\Phi_{p,q,r+1} - \Phi_{p,q,r-1}\right)
\end{aligned} \tag{12}$$

and

$$\begin{aligned}
x\left(t+\tau\right) &= x\left(t\right) + V_x\left(t+\frac{1}{2}\tau\right)\tau \\
y\left(t+\tau\right) &= y\left(t\right) + V_y\left(t+\frac{1}{2}\tau\right)\tau \\
z\left(t+\tau\right) &= z\left(t\right) + V_z\left(t+\frac{1}{2}\tau\right)\tau
\end{aligned} \tag{13}$$

It's worth to realize that GAVLAB used the simplest Euler's method to integrate which is extremely un stable. However when there are so many planets in the system probably there will not be stable trajectory here and accuracy is not as worth as time.

4.7.2 Using FFT to calculate potential

We can directly calculate g but we have to calculate a lot of square roots which is slow. So GAVLAB used FFT to calculate that.

The mechanism of FFT is a liite similar to the the mechanism of calculating the sum of logarithm to get the product of bases.

If the solar system was divided by the $(n+1)\times(n+1)\times(n+1)$ grid,then we need $(2n\times 2n\times 2n)$ grid to do FFT. For convenience we write Φ in position(α,β,γ) as

$$\Phi(\alpha,\beta,\gamma)=\sum_{p=0}^{2n-1}\sum_{q=0}^{2n-1}\sum_{r=0}^{2n-1}M(p,q,r)g(\alpha-p,\beta-q,\gamma-r) \tag{14}$$

here

$$\begin{aligned} g(i,j,k)&=-\frac{G}{\sqrt{(i\Delta_x)^2+(j\Delta_y)^2+(k\Delta_z)^2}} \\ g(0,0,0)&=1 \end{aligned} \tag{15}$$

g is Green function and Δ is the interval of the grid.

So the Fourier Transformation of the function g is

$$\hat{g}(k,l,m)=\sum_{a=0}^{2n-1}\sum_{b=0}^{2n-1}\sum_{c=0}^{2n-1}g(a,b,c)e^{-\frac{2\pi i}{2n}(ak+bl+cm)} \tag{16}$$

$$\Phi(\alpha,\beta,\gamma)=\sum_{p=0}^{2n-1}\sum_{q=0}^{2n-1}\sum_{r=0}^{2n-1}M(p,q,r)(\frac{1}{2n})^3\sum_{k=0}^{2n-1}\sum_{l=0}^{2n-1}\sum_{m=0}^{2n-1}\hat{g}(k,l,m)e^{-\frac{2\pi i}{2n}[(\alpha-p)k+(\beta-q)l+(\gamma-r)m]} \tag{17}$$

If we change the position of the terms in the equation above we can get

$$\Phi(\alpha,\beta,\gamma)=(\frac{1}{2n})^3\sum_{k=0}^{2n-1}\sum_{l=0}^{2n-1}\sum_{m=0}^{2n-1}\hat{g}(k,l,m)\left[\sum_{p=0}^{2n-1}\sum_{q=0}^{2n-1}\sum_{r=0}^{2n-1}M(p,q,r)e^{-\frac{2\pi i}{2n}(pk+ql+rm)}\right]e^{\frac{2\pi i}{2n}(\alpha k+\beta l+\gamma m)} \tag{18}$$

We may call the expression inside the square bracket the transformation of the mass distribution $M(p,q,r)$,let

$$\hat{M}(k,l,m)=\sum_{p=0}^{2n-1}\sum_{q=0}^{2n-1}\sum_{r=0}^{2n-1}M(p,q,r)e^{-\frac{2\pi i}{2n}(pk+ql+rm)} \tag{19}$$

Then Phi can be expressed as

$$\Phi(\alpha,\beta,\gamma)=(\frac{1}{2n})^3\sum_{k=0}^{2n-1}\sum_{l=0}^{2n-1}\sum_{m=0}^{2n-1}\hat{g}(k,l,m)\hat{M}(k,l,m)e^{\frac{2\pi i}{2n}(\alpha k+\beta l+\gamma m)} \tag{20}$$

The whole FFT here can be written in a more concise form

$$\hat{\Phi}(k,l,m) = \hat{g}(k,l,m)\hat{M}(k,l,m)$$
$$\Phi(\alpha,\beta,\gamma) = (\frac{1}{2n})^3 \sum_{k=0}^{2n-1}\sum_{l=0}^{2n-1}\sum_{m=0}^{2n-1} \hat{\Phi}(k,l,m)e^{\frac{2\pi i}{2n}(\alpha k+\beta l+\gamma m)} \quad (21)$$

This is the algorithm of FFT above

Input: The grid
Output: $\Phi(i,j,k)$
1 calculate Green function $g(a,b,c)$ of the grid;
2 Fourier Transformation $g(a,c,b) \Longrightarrow \hat{g}(k,l,m)$;
3 Fourier Transformation $M(p,q,r) \Longrightarrow \hat{M}(k,l,m)$ and store that information in $(2n \times 2n \times 2n)$ grid.;
4 calculate the multiplication $\hat{\Phi}(k,l,m) \leftarrow \hat{g}(k,l,m)\hat{M}(k,l,m)$;
5 final Fourier Transformation $\Phi(\alpha,\beta,\gamma) \leftarrow (\frac{1}{2n})^3 \sum_{k=0}^{2n-1}\sum_{l=0}^{2n-1}\sum_{m=0}^{2n-1} \hat{\Phi}(k,l,m)e^{\frac{2\pi i}{2n}(\alpha k+\beta l+\gamma m)}$;
6 return Φ;

Algorithm 1: FFT

4.7.3 CGrid class

This is the definition of CGrid class

```
class CGrid:public CIntegrator{
public:
    CGrid();
    virtual void Update(CSolar &S);
private:
    list<CPlanet> Cell[N][N][N];
    double Phi[N+2][N+2][N+2];
    void FFT();
    void UpdateCells();

};
```

CODE 21 : CGrid class

here we have list array **Cells** to represent the cells in the grids. **Phi** array is to store the potential array. In each loop of simulation,firstly we call the function **UpdateCells** to update the planets in each cell because planets will move around. Then we call function **FFT** to update the potential array using FFT technique.

First we have to initialize all the array and lists of CGrid so we defined the constructor

```
CGrid::CGrid(){
    for (int i=0; i<N; i++) {
        for (int j=0; j<N; j++) {
            for (int k=0; k<N; k++) {
                Cell[i][j][k].clear();
                Phi[i][j][k]=0;
            }
        }
    }
}
```

CODE 22 : CGrid()

Then we update the cells by list operations

```
void CGrid::UpdateCells(){
    for (int i=0; i<N; i++) {
        for (int j=0; j<N; j++) {
            for (int k=0; k<N; k++) {
                list<CPlanet>::iterator itr;
                for (itr = Cell[i][j][k].begin();
                   itr != Cell[i][j][k].end(); itr
                   ++) {
                  // calculate the index in the
                     grid
                  int x=itr->X()/DELDA,y=itr->Y()/
                     DELDA,z=itr->Z()/DELDA;
                  if (x != i || y != j || z != k)
                     {
```

CODE 23 : UpdateCells() I

```
                        // if the planet is out of
                           the cell,erase it
                        Cell[i][j][k].erase(itr);
                        // and put it into right
                           place
                        Cell[x][y][z].push_back(*itr
                           );
                    }
                }
            }
        }
    }
}
```

CODE 24 : UpdateCells() II

This is the Update() function

```
void CGrid::Update(CSolar &S){
    // Update the cells according to the position of
        planets
    UpdateCells();
    // using FFT to calculate potential distribution
    FFT();

    for (int i=0; i<64; i++) {
        for (int j=0; j<64; j++) {
            for (int k=0; k<64; k++) {
                list<CPlanet>::iterator itr;
                for (itr = Cell[i][j][k].begin();
                   itr != Cell[i][j][k].end(); itr
                   ++){
```

CODE 25 : Update() I

```
                        double vx=(Phi[i+1][j][k]-Phi[i
                           ][j][k])
                        ,vy=(Phi[i][j+1][k]-Phi[i][j][k
                           ]),
                        vz=(Phi[i][j][k+1]-Phi[i][j][k])
                           ;
                        itr->v+=vector3d(vx,vy,vz);
                        itr->r+=DT*itr->v;
                        itr->Updatetr(DT);

                    }
                }
            }
        }
        S.t+=DT;
}
```

CODE 26 : Update() II

Unfortunately Particle mesh method doesn't work well either,because it used too much cells in 3-dimensional space. Here in GABLAB I divided the grid into 64^3 cells. It's huge,but still not huge enough. Usually grid method was used to research the 2-dimensional cluster system. Grids with 256^2 cells do a good job.

4.8 What if collision happens

Well I have to say there will be a lot of collisions in the gravitational system since the planets tend to attract each other. So we have to decide what to do when collision happens.

In GAVLAB collisions are divided into two different circumstances.

The first case is when planet collides with sun. Wow this is extremely dangerous and horrible so we let that planet destroyed and we add that planet's mass onto sun's mass.

```
vector<CPlanet>::iterator itri,itrj;
for (itri=Stars.begin(); itri != Stars.end();
   itri++) {
    if (itri->exist == false) {
        continue;
    }
    double R0=(itri->r^(itri->v/(itri->v.
       Magnitude()))).Magnitude();

    if (R0<E) { // E is marco defined
        cout<<itri->Name<<" BOOM"<<endl;
        itri->exist=false;
        centerStarMass+=itri->GetMass();
    }
}
```

CODE 27 : collision with sun

The second case is when two close planets collides together. When that happens we choose to merge them which means they sticked to a new planet. According to the conservation of momentum we have

$$\begin{aligned} M' &= M_1 + M_2 \\ \vec{v'} &= \frac{M_1\vec{v_1} + M_2\vec{v_2}}{M_1 + M_2} \end{aligned} \tag{22}$$

and then we mark one of those two planet as not exist.

```
vector<CPlanet>::iterator itri,itrj;
for (itri=Stars.begin(); itri != Stars.end(); itri
   ++) {
        for (itrj=Stars.begin(); itrj!=Stars.end();
           itrj++) {
        if (itrj == itri || itrj->exist==false) {
           continue;}
        vector3d rij=itri->r-itrj->r;
        double R=rij.Magnitude();
        if(R < E){// if they are too close
            cout<<itri->Name<<" and "<<itrj->Name<<"
                BOOM!"<<endl;
            // update v
            itri->v=(itri->Mass*itri->v+itrj->Mass*
               itrj->v)/(itri->Mass+itrj->Mass);
            itri->Mass+=itrj->Mass;// merge mass
            itrj->exist=false;// delete one of them
        }
    }
}
```

CODE 28 : collision between planets

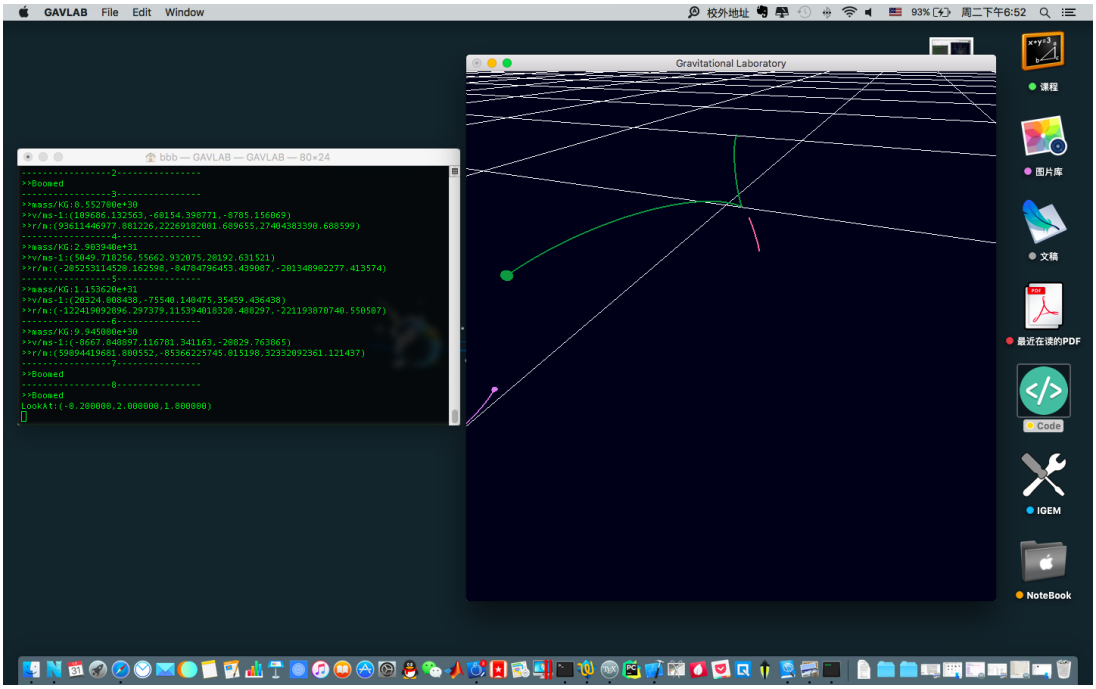


Figure 11: Collision happened between the green planet and the pink planet

5 VIEW the GAVLAB with OpenGL

5.1 OpenGL Library

OpenGL is Open Graphics Library,a powerful cross platform graphic library. It's easy to make high quality animation or graph in a short time with OpenGL.

5.2 CDisplay class

CDisplay class integrated almost all OpenGL related code and contains two functions **display** and **reshape** which is used to pass to the OpenGL commands

```
class CDisplay{
public:
    // draw everything
    static void display(void);
    // reshape the window when it changes size
    static void reshape (int w, int h);
private:
    // parameters for LookAt
    static GLdouble eyeX, eyeY, eyeZ;
    static GLdouble centerX, centerY, centerZ;
    static GLdouble upX, upY, upZ;
    // scale factor of LookAt parameters
    static double g;
    // position to draw the point
    static GLdouble poX;
    static GLdouble poY;
    static GLdouble poZ;
    friend class CControl;
};
```

CODE 29 : CDisplay

eyeX,eyeY,eyeZ are the position of the camera and centerX,centerY,centerZ are the position that the camera is looking at and upX,upY,upZ is the direction of the up of the camera to determine the orientation of camera. g is

the scale factor to adjust the offset. poX,poY,poZ is the point to be drawn

Function reshape() is called when users want to change the windos size.

```
void CDisplay::reshape (int w, int h){
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    glFrustum (-1.0, 1.0, -1.0, 1.0, 1.5, 200.0);
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0f,-20.0f,-150.0f);
}
```

CODE 30 : reshape()

Function display() is very long because it is responsible for graphing the whole system including balls represent planet,lines represent trajectory and grids represent the coordinate. PartI shows how to initialize gl before drawing.

```
void CDisplay::display(void){
    glClear(GL_COLOR_BUFFER_BIT  |
       GL_DEPTH_BUFFER_BIT);
    // set the background color deep blue
    glClearColor(0, 0, 0.1, 1);
    glShadeModel (GL_SMOOTH);
    glPushMatrix();
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(75.0f, 1.0f, 1.0f, 40000000);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(eyeX, eyeY,eyeZ, centerX, centerY,
       centerZ, upX, upY, upZ);
    // ... the rest parts
    glPopMatrix();
    glFlush ();
}
```

CODE 31 : display() PartI Set up the graph

Then we have to draw the backgrounds. First we want to draw the 2-dimensional grids represent the reference coordinate. Then we have to draw the sun which is fixed on origin position (0,0,0).

```
void CDisplay::display(void){
    // Part I...
    //coordinate lines
    glColor3f( 1.0f, 1.0f, 1.0f );
    glLineWidth(0.1);
    // x-axis
    for(int y=-300;y<=300;y+=5){
        glBegin( GL_LINE_STRIP );
        {
            glVertex3f( -300.0f, y, 0.0f);
            glVertex3f( 300.0f, y, 0.0f);
        }
        glEnd();
    }
    // y-axis
    for(int x=-300;x<=300;x+=5){
        glBegin( GL_LINE_STRIP );
        {
            glVertex3f( x, -300.0f, 0.0f);
            glVertex3f( x, 300.0f, 0.0f);
        }
        glEnd();
    }
    // the red sun
    glColor3f(1.0, 0.0, 0.0);
    glutSolidSphere(.03, 100, 100);
    //...
}
```

CODE 32 : display() PartII drawing background

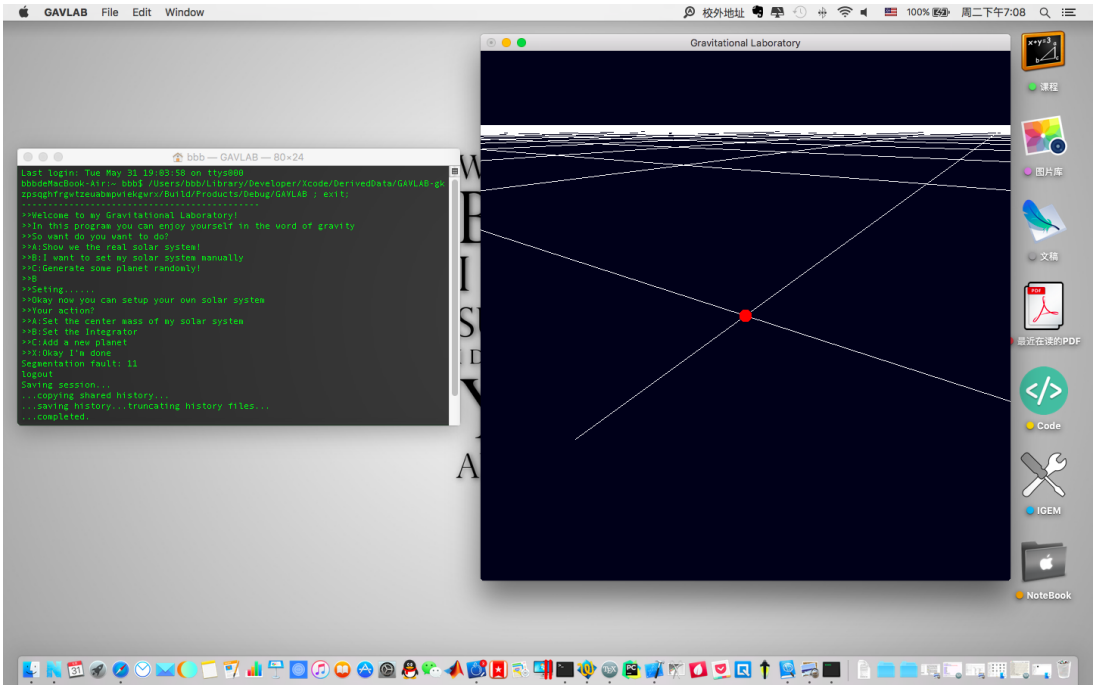


Figure 12: Draw the background first

After that we want to draw the trajectory of all the planets cause that's what we really care about.

```
void CDisplay::display(void){
    // Part I & Part II...
    vector<CPlanet>::iterator itr;
    // draw the tragetory here
    for (itr=Solar.Stars.begin(); itr != Solar.Stars
       .end(); itr++) {
        GLfloat r=itr->Red(),g=itr->Green(),b=itr->
           Blue();
        glColor3f( r, g, b );//setup the color of
           line
        glLineWidth(2);// make the trajectory thick
        // this mode allows users draw stripes
        glBegin(GL_LINE_STRIP);
        {
            list<vector3d>::iterator p;
            for (p=itr->trail.begin(); p != itr->
               trail.end(); p++) {
                poX = p->X()/SCALE;
                poY = p->Y()/SCALE;
                poZ = p->Z()/SCALE;
                glVertex3d(-poX,poY,poZ);
            }
        }
        glEnd();
    }
    //...
}
```

CODE 33 : display() Part III drawing the trajectory

Some times we need not only the animation but also the real data. For this reason GAVLAB has the option of exporting data on the command window.

```
void display(void){//Part I II III...
    if (CControl::ShowRecord == true && CControl::
       play == true) {
        printf("\n\n\n>>Time/DAY:%lf\n",Solar.GetT()
           /DAY);
    }
    for (itr=Solar.Stars.begin(); itr != Solar.Stars
       .end(); itr++) {
        if (itr->Exist() == false) {
            if (CControl::ShowRecord == true &&
               CControl::play == true) {
                printf("-----------------%s
                   ----------------\n",itr->GetName
                   ().c_str());printf(">>Boomed\n");
            }continue;
        }
        if(CControl::ShowRecord == true && CControl
           ::play == true){
            printf("-----------------%s
               ----------------\n",itr->GetName().
               c_str());
            printf(">>mass/KG:");
            printf("%e\n",itr->GetMass()*1e18);
            printf(">>v/ms-1:");
            printf("(%lf,%lf,%lf)\n",itr->GetV().X()
               ,itr->GetV().Y(),itr->GetV().Z());
            printf(">>r/m:");
            printf("(%lf,%lf,%lf)\n",itr->X(),itr->Y
               (),itr->Z());
        }
    }
//...}
```

CODE 34 : display() PartIV exporting data

The reason why we used C styled function printf() here instead of cout is that printf is fast while ostream object operation cause a lot of time. Time is everything in GAVLAB!

Finally We can draw a ball representing the planet to make GAVLAB beautiful.

```
void display(){
    // Part I-IV ...
    vector<CPlanet>::iterator itr;
    for (itr=Solar.Stars.begin(); itr != Solar.Stars
       .end(); itr++){
        GLfloat r=itr->Red(),g=itr->Green(),b=itr->
           Blue();
        double radius=0.03*itr->GetRadius();
        glColor3f(r, g, b);
        poX = itr->X()/SCALE;
        poY = itr->Y()/SCALE;
        poZ = itr->Z()/SCALE;
        // translate the position of sphere
        glTranslated(-poX,poY,poZ);
        glutSolidSphere(radius, 100, 100);
        // translate the position back to origin
        glTranslated(poX,-poY,-poZ);
    }
}
```

CODE 35 : display() Part V drawing planets

Because **glutSolidShpere()** can only draw sphere in position (0,0,0). So we have to translate the origin position to the position of the planet then draw the sphere. After that we have to translate it back in avoid of drawing the next planet in a wrong position.

6 CONTROL class is the commander

6.1 CControl class

CControl class has two jobs.

The first job is to interact with users through command window. It means CControl has to be able to get the command entered by users through commmand window.

The second job is to control the rest classes,functions,variables in the program such as setup the solar system or readjust the camera.

```
class CControl{
public:
    static void UpdateState();
    // readjust lookAt parameter
    static void keyboard(unsigned char key, int x,
       int y);
    // initialize the program
    static void init();
    // initialize the real solar
    static void initSolar();
    static void SetSolarManually();
    static void AddNewPlanet();
    static void AddRandomPlanet();
    static void Terminate();
private:
    static bool ShowRecord;// if export data
    static bool play;// is is playing or paused

    friend class CDisplay;

};

bool CControl::ShowRecord=false;
bool CControl::play=true;
```

CODE 36 : CControl class

We need a ultimate update command function. So we defined UpadteState() function in CControl to be the interface of all simulation related functions.

```
void CControl::UpdateState(){
    if(play)
        Solar.UpdateSolar();
    glutPostRedisplay();
}
```

CODE 37 : UpdateState()

We may want to change the perspective when running GAVLAB to get a better view of the trajectories or want to pause simulation to find out some secret. So want CControl does here is getting the command from the keyboard and parse the command into operations.

```
void CControl::keyboard(unsigned char key, int x,
   int y){
    switch (key)    {
        case '+': CDisplay::g*=10; break;
        case '-': CDisplay::g/=10.0; break;
        case 'w': CDisplay::eyeY += CDisplay::g*
           OFFSET; break;
        case 'x': CDisplay::eyeY -= CDisplay::g*
           OFFSET; break;
        case 'a': CDisplay::eyeX -= CDisplay::g*
           OFFSET; break;
        case 'd': CDisplay::eyeX += CDisplay::g*
           OFFSET; break;
        case 's': CDisplay::eyeZ -= CDisplay::g*
           OFFSET; break;
        case 'S': CDisplay::eyeZ += CDisplay::g*
           OFFSET; break;
```

CODE 38 : keyboard() Part I

```
        case 'l': CDisplay::centerX -= 0.5*CDisplay
           ::g*OFFSET; break;
        case 'j': CDisplay::centerX += 0.5*CDisplay
           ::g*OFFSET; break;
        case 'i': CDisplay::centerY += CDisplay::g*
           OFFSET; break;
        case 'k': CDisplay::centerY -= CDisplay::g*
           OFFSET; break;
        case 'r':
            CDisplay::eyeX = 2.8; CDisplay::eyeY =
               1.4; CDisplay::eyeZ= 2.4;
            CDisplay::centerX= 0; CDisplay::centerY=
                0; CDisplay::centerZ= 0;
            CDisplay::upX= 0; CDisplay::upY= 0;
               CDisplay::upZ= 1;
            break;
        case 27: Terminate(); break;
        case ' ':play = !play;break;
        default: break;
    }
    if(ShowRecord == true){
        printf("LookAt:(%lf,%lf,%lf)\n",CDisplay::
           eyeX,CDisplay::eyeY,CDisplay::eyeZ);
    }

}
```

CODE 39 : keyboard() Part II

When users run GAVLAB the function **init()** will be called to set up all the initial variables such as solar system and integrators and so on. So want init() do is getting command from the commmand window and call the interfaces of initializing functions.

Attention there used to be a lot of dialogs in printf() functions. I just omitted those dialogs and marked them as comments. Also I used function **usleep()** to pause between the sentences in order to make time for users to read the word exported from command window.

```
void CControl::init(){
    Solar.Stars.clear();
    // dialogs and ask users want to do
    string command;
    cin>>command;
    switch (command[0]) {
        case 'A':
            CControl::initSolar();
            break;
        case 'B':
            CControl::SetSolarManually();
            break;
        case 'C':
            CControl::AddRandomPlanet();
            break;
        default:
            break;
    }

    printf(">>Do you want the data exported in
       command window?\n");
    printf(">>A:Yes\n>>B:No\n>>");
    cin>>command;
    switch (command[0]) {
        case 'A':
            CControl::ShowRecord=true;
            break;
        case 'B':
            CControl::ShowRecord=false;
            break;
        default:
            break;
    }
    // dialogs and tell users how to operate GAVLAB
}
```

CODE 40 : init()

The first choice for users is to simulate the real solar system! So we have to initial the solar system before simulation,add real planets's data into the Solar object. When simulating real solar,we just use SI4 integrator.

```
void CControl::initSolar(){
    printf(">>Please wait...\n");
    double dt=DAY*0.1;
    Solar.SetMass(MS);
    Solar.SetI(SI4);
    Solar.SetStep(dt);
    CPlanet mercuy("MERCURY",0.0553*ME,0.35,0.3871*
       RA,0,344,0,52716.3,0,.2, .2, .5);
    CPlanet venus("VENUS",0.815*ME,0.8,0.7233*RA
       ,0,0,0,35200,0, 1, .7, 0);
    CPlanet earth("EARTH",ME,0.8,RA,0,0,0,29789,0,
       0, 1, 0);
    CPlanet mars("MARS",0.1074*ME,3,1.523*RA
       ,0,0,0,25310,0, 1, 1, 0);
    CPlanet jupiter("JUPITER",317.834*ME,2.5,5.202*
       RA,0,0,0,13413,0, 1, .5, .5);
    CPlanet saturnus("SATURNUS",95.159*ME,1.5,9.544*
       RA,0,0,0,9927.7,0, 1, 1, .5);
    CPlanet uranus("URANUS",14.5*ME,1.5,19.19*RA
       ,0,0,0,6978,0, .5, 1, .5);
    CPlanet neptune("NEPTUNE",17.2*ME,0.35,30.13*RA
       ,0,0,0,5670.9,0, .4, .4, .4);
    Solar.AddPlanet(mercuy);
    Solar.AddPlanet(venus);
    Solar.AddPlanet(earth);
    Solar.AddPlanet(mars);
    Solar.AddPlanet(jupiter);
    Solar.AddPlanet(saturnus);
    Solar.AddPlanet(uranus);
    Solar.AddPlanet(neptune);
    printf(">>done\n");
}
```

CODE 41 : initSolar()

Also GAVLAB is a program with freedom which means the users can add any planet in the solar. Then function **SetSolarManually()** will be called and ask users to enter the data through command window.

```
void CControl::SetSolarManually(){

    printf(">>Seting......\n");
    usleep(3000000);
    printf(">>Okay now you can setup your own solar
       system\n");
    usleep(1000000);
    printf(">>Your action?\n");
    printf(">>A:Set the center mass of my solar
       system\n");
    printf(">>B:Set the Integrator\n");
    printf(">>C:Add a new planet\n");
    printf(">>X:Okay I'm done\n");
    string command;
    string IntrgratorType;
    bool Ismass=false,IsIntegrator=false;
    bool isbreak=false;
    while (cin>>command) {
        switch (command[0]) {
       // switch the command showed in CODE 37-38
        }
        if (isbreak) {
            break;
        }
        printf(">>Your action?\n");
        printf(">>A:Add a new planet\n");
        printf(">>B:Okay I'm done\n");

    }
}
```

CODE 42 : SetSolarManually()

```
        switch (command[0]) {
            case 'A':
                double Mass;
                printf(">>Mass/kg:");
                cin>>Mass;
                Solar.SetMass(Mass);
                Ismass = true;
                break;
            case 'B':
                printf(">>You can choose the
                   following Integrators\n");
                printf(">>A:SI2\n>>B:SI4\n>>C:SI6\n
                   >>D:Grid\n>>");
                cin>>IntrgratorType;
                switch (IntrgratorType[0]) {
                    case 'A':
                        Solar.SetI(SI2);
                        IsIntegrator=true;
                        break;
                    case 'B':
                        Solar.SetI(SI4);
                        IsIntegrator=true;
                        break;
                    case 'C':
                        Solar.SetI(SI6);
                        IsIntegrator=true;
                        break;
                    case 'D':
                        Solar.SetI(Grid);
                        IsIntegrator=true;
                        break;
                    default:
                        break;
                }
                break;
```

CODE 43 : switch part I

```
            case 'C':
                CControl::AddNewPlanet();
                break;
            case 'X':
                if (IsIntegrator && Ismass) {
                    isbreak=true;
                }
                else{
                    if (!IsIntegrator) {
                        printf(">>You didn't set
                           your solar's integrator\n
                           ");
                    }
                    if (!Ismass) {
                        printf(">>You didn't set
                           your solar's center mass\
                           n");
                    }
                }
                break;
            default:
                break;
        }
```

CODE 44 : switch part II

If we chose to add a planet in the solar system manually in function **SetSolarManually()** it will call function **AddNewPlanet()** which will get the information from the commmand window entered by user and then it will generate the corresponding planet and add it into solar system.

```
void CControl::AddNewPlanet(){
    string name;
    double mass;
    double radius;
    double x,y,z,vx,vy,vz;
    double r,g,b;
    printf(">>name:");cin>>name;
    printf(">>mass/kg:");cin>>mass;mass/=1e18;
    printf(">>radius/km:");
    cin>>radius;radius/=1.4e6;
    printf(">>Assuming the sun is in (0,0,0)\n");
    printf(">>Position X/AU:");cin>>x;x*=RA;
    printf(">>Position Y/AU:");cin>>y;y*=RA;
    printf(">>Position Z/AU:");cin>>z;z*=RA;
    printf(">>Velocity Vx/ms-1:");cin>>vx;
    printf(">>Velocity Vy/ms-1:");cin>>vy;
    printf(">>Velocity Vz/ms-1:");cin>>vz;
    printf(">>Set the Color now\n");
    printf(">>Red:");cin>>r;
    printf(">>Green:");cin>>g;
    printf(">>Blue:");cin>>b;
    P = new CPlanet(name,mass,radius,x,y,z,vx,vy,vz,
       r,g,b);
    Solar.AddPlanet(*P);
}
```

CODE 45 : AddNewPlanet()

Finally if the users choose to press ESC to terminate GAVLAB,function **Terminate()** will be called.

In fact Terminate() controls what happen when the program exits.

```
void CControl::Terminate(){
    printf(">>The simuation is over\n");
    printf(">>Thanks for using\n");
    exit(0);
}
```

CODE 46 : Terminate()

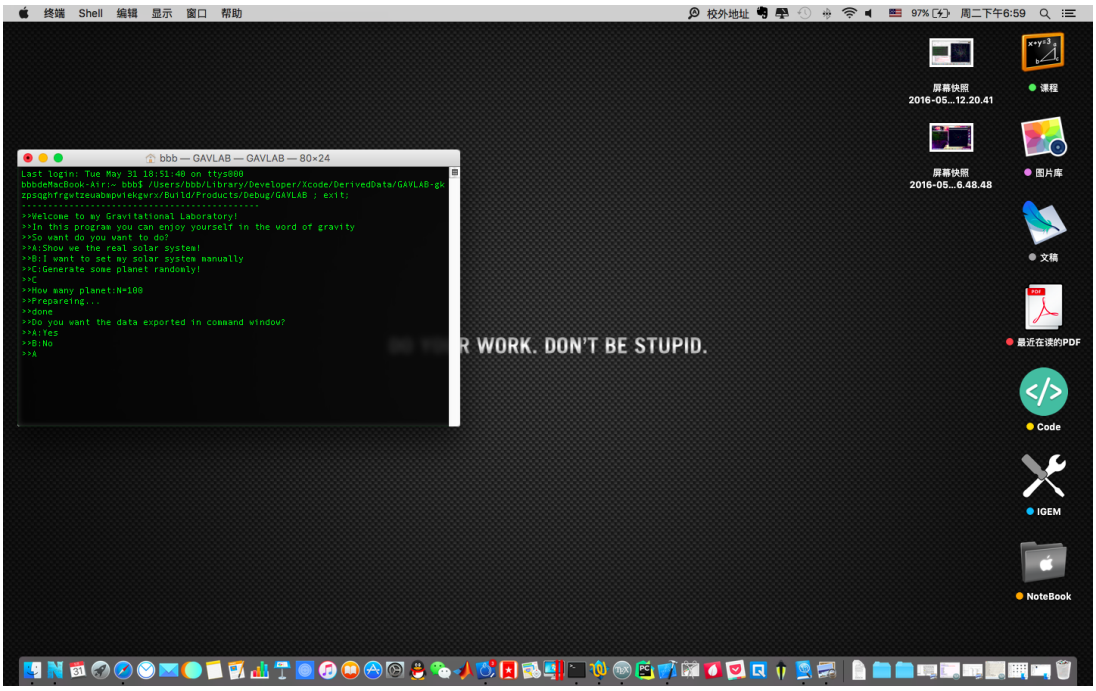


Figure 13: CControl is asking for order

6.2 main function

Finally Let's see what's happening in main function.

```
CSolar Solar;
CPlanet* P;
int main(int argc, char** argv) {
    CControl Control;
    Control.init();
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_RGBA |  GLUT_SINGLE);
    glutInitWindowSize (700, 700);
    glutInitWindowPosition (700, 100);
    glutCreateWindow ("Gravitational Laboratory");
    glutDisplayFunc(CDisplay::display);
    glutReshapeFunc(CDisplay::reshape);
    glutKeyboardFunc(CControl::keyboard);
    glutIdleFunc(CControl::UpdateState);
    glutMainLoop();
    return 0;
}
```

CODE 47 : main()

I will explain those glut-prefixed functions here

glutInit initialize GLUT

glutInitDisplayMode Set up the display mode.Here we chose RGBA color mode and single buffer window mode.

glutInitWindowSize Set the size of the window.

glutInitWindowPosition Set the position of the window.

glutCreateWindow Create a window.

glutDisplayFunc Time to display!Our CDisplay::display is the argument of this function which means when it's time to display in a loop GAVLAB will call display function.

glutReshapeFunc Time to reshape the window and it will call our CControl::reshape function.

glutKeyboardFunc Time to parse the command from the keyboard.When users type on keyboard GAVLAB will call our CControl::keyboard function to parse the command.

glutIdleFunc When there's no extra command GAVLAB will call function CControl::UpdateState in the loop.

glutMainLoop The whole animation is a big loop in which the functions above will be called and executed one by one.

And we defined a global variable Solar because there's only one solar system in GAVLAB. The CPlanet pointer is pointing those planet objects who are dynamically allocated durring the program.

7 Epilogue

7.1 I paid a lot of time building GAVLAB

I've been building GAVLAB since the beginning of this semester after professor Xue assigned this project. For days and days I cost a lot of time thinking how to build it. At first the design and blue print was finished at April. Then it took me half a month to program and code on my computer. For the whole May I kept optimizing the model, restructure my program to be perfect and test the result of GAVLAB. At last I used three days to make this 50 pages report by LaTeX.

All the work was done by myself,including thousands lines of code and 50 pages report.

7.2 I found a lot of reference

I found a lot of resources,including three books **Gravitational N-Body Simulations**, **The Computer Simulation of The Gravitational N-Body Problem** , **The Art of Molecular Dynamic Simulation** and tens of papers from Comp Phys.

I also read **OpenGL SuperBible** to make GAVLAB visual.

7.3 I really love GAVLAB

GAVLAB is the third big C++ program I've ever built after **Pokemon** and **The Game of Brick**. It's my own child,I really love it. It's so beautiful,so powerful to me.

7.4 I appreciate Professor Xue

Thank you for giving me such a chance to experience the importance of data structure and the pleasure of programing,especially the happiness of making my own program.

References

[1] Physics.for.Game.Developers. David.M.Bourg. 2nd 2013 111

[2] The computer Simulation of The Gravitational N-Body Problem LiuBulin

[3] Celestial Mechanics and Dynamical Astronomy Haruo Yoshida 1993 56 27

[4] Phys Letters Yoshida H A 1990,150:262