模拟太阳系上机报告

姓名：秦光辉

学号：1500011398

模拟太阳系上机报告

姓名：秦光辉

学号：1500011398

题目描述：

编制程序，描述太阳系星星随时间变化的运动轨迹，以及在各种意外情况下太阳系的稳定性分析。利用编制的程序来分析一些物理问题，例如如果太阳质量下降一半会如何？如果一个小星星击中地球，地球运动轨迹会如何变化？

题目分析：

本题要求我们通过设计数据结构和算法来实现一个模型：太阳系。太阳系由太阳，八大行星，小行星带，八大行星的卫星以及可能出现的卫星组成。太阳系之间的作用力可以认为仅有万有引力，不考虑其他作用力。

如果仅考虑八大行星，其计算量并不大，八大行星之间的作用力并不复杂。但是倘若考虑所有的卫星的话，其计算量是庞大的。我们不能单纯把所有的作用力全部考虑进去，应该有所取舍。在我实现的过程中我定义了一个结构：树，用树来表示所有的星球，这样我们就便于取舍作用力。

在计算过程中，各个行星距离太阳的轨道近似于周期性的，而每次都需要重新计算作用力，这样造成了很大的浪费，因为相同的半径，其作用力是相同的，我们可以考虑使用一个字典结构，把半径作为关键码 key，作用力作为值 value，这样我们就可以用检索获得作用力，其效率大大提高。

还有一个关键因素是观察其运动状态。我设计了两种方式获得其运动状态，一个是输出为文件，一个是可视化。可视化是用 OpenGL 实现的，输出为文件可以选择其精度。这样我们可以较为清晰获得其运动信息。

实现之后，我们可以利用这个模型模拟很多现实生活中不可能出现的事情，比如加入一颗行星，太阳的质量突然无限扩大而成为一个黑洞，一个外来的陨石撞击地球等，而且我们可以验证一些计算，比如地日之间的拉格朗日点的稳定性等。

头文件：

AVL.h

声明了 AVL 平衡二叉树的模板类。

Bucket.h

声明了一个桶结构。

control.h

声明了用于读入键盘、鼠标操作的函数，移动视角和位置。

display.h

声明了可视化的主函数，在 main 函数中调用。

element.h

声明了 element 模板类，在 hash table 字典中会用到。

globals.h

记录一些太阳系的参数，物理参数。

hashtable.h

声明了 hash table 模板类。

hierarchical_indexed.h

声明了一个三级索引字典

link.h

声明了 link 模板类，是一个链表，在 queue 中会用到。

load_planet.h

声明了读入太阳系参数，返回行星列表的函数，在 main 函数中会调用。

node.h

声明了 node 模板类， 这是一个不循环的链表，在 stack 中会用到。

objloader.h

声明了读入 obj 模型文件的函数。

planet.h

声明了 planet 类，用来存储行星的信息（恒星、行星、卫星统一使用 planet 名称）

queue.h

声明了 queue 模板类，这是一个队列结构。

shader.h

声明了 shader 函数，在可视化中会用到。

Stack.h

声明了 stack 栈模板类。

text2D.h

声明了一个可以在窗口表面打印 2D 字体的函数。

texture.h

声明了一个可以贴图的函数。

update.h

声明了一个 updater 类，可以通过万有引力更新行星的信息。

Additional Dependencies:

Freeglut

Glew

Glfw

Glm

平台和编译器：

Visual Studio 14.0

MSVC 1900

第一部分 程序和结构

1、星体的树形结构：

太阳系的行星大致可以分为三个等级：恒星，行星，卫星。

我们可以把所有的星体放入一个顺序表中，每次计算的时候把顺序表里的所有星体全部引力计算一次，但是这样费时费力。我采用了一种树形结构：

第一级：太阳

第二级：八大行星，全部是太阳这个结点的子树

第三级：卫星系统。地球有 1 颗卫星，火星有 2 颗卫星，以此类推

在计算某个行星的受力时，我们不会计算所有的星体对它的作用，而单单考虑它的所有上级星体的作用。这种方法大大减轻了计算量，而且并没有造成很大误差。树形结构定义在 planet 类中，为

```
std::vector<planet *> satellite;
```

在统计这些作用行星的时候，我采用了**队列**的结构。通过队列的**广度优先搜索**，确定了所有需要考虑作用力的行星，放在一个 vector 数组中，每次更新数据的时候调用。函数定义在 planet 类中，为

```
std::vector<planet *> &set_interaction_planet_vector();
```

计算出来的行星列表存储在

```
std::vector<planet *> interaction_planet;
```

中。具体的算法写在代码的注释中。

2、用图结构来简化万有引力的计算

在上述过程中，行星考虑了所有的上级行星对它的吸引力，但是我们又遇到另一个问题：同级行星的引力要不要都考虑呢？

我认为，我们需要考虑同级行星的引力，但是并不需要全部考虑。众所周知，宇宙的尺度非常大，有的行星离得很远，有的行星离得很近。为了减小模拟的误差，同时也为了减小计算量，我设计了一个**图结构**，定义在 graph.h 中。图结构是一个模板类，它的成员有：

std::vector<T> m_vertex;

std::vector<std::vector<double> > m_sides;

double threshold = 0;

上述成员中，m_vertex 是图的顶点，也就是行星的指针。m_sides 是图的边，也就是行星的距离。Threshold 是一个阈值，如果一个行星对另一个行星造成的加速度如果大于这个阈值，我们就需要考虑这个行星的作用力。对于三级的空间模型而言，我们有三级的 threshold，定义在 globals.h 中。初始值为下：

#define THRESHOLD_0 0

#define THRESHOLD_1 4e-20

#define THRESHOLD_2 1e-19

如果我们希望牺牲一些计算速度来获得计算精度的话，可以把 threshold 调低，这样就扩大了计算时的考虑范围。

另外我们有一个 updater 的类。这个类内存有三个 graph<planet *> ，用来统计哪些行星的作用力需要考虑，哪些作用力不需要考虑。各个行星的动态是一直在更新的，所以我们需要实时更新 graph，更新 graph 的函数如下：

void updater::update_graphs();

这个函数可以根据此刻的行星位置更新 graph，来获得更加实时的距离信息。这个函数是否调用由 counter 决定。Updater 中有

Int counter;

这个是用来计数的，每计算一次行星的实时坐标，我们就递增一次 counter。当 counter 满足：

(++counter) % PERIOD_FRESH_GRAPH == 0

则更新一次 graph。可见，graph 更新的频率由 PERIOD_FRESH_GRAPH 决定。它定义在 globals.h 中。初始值为：

#define PERIOD_FRESH_GRAPH 128

换言之，我们每更新 128 次坐标数据，就更新一次 graph。

3、存储星体的 AVL 平衡二叉树结构

在可视化的操作中，有一些操作需要指定操作对象，比如把视角固定到某一个星体上。此时需要读入星体的名字，找到对应的星体。这种操作在星体数目极为庞大的时候，并不总是可以马上找到的；如果我们的操作涉及大量星体的时候，顺序表检索未免太过缓慢。

如果使用简单的字典结构或者普通二叉树，在面对大量的添加的时候，其检索效率必然会大为下降，我放弃了这一想法。

我用 **AVL 平衡二叉树**结构实现了这一功能，写在 display.cpp 中，为

```
bintree<std::string, planet *> planet_bintree;

for (auto c : all_planet)

        planet_bintree.add(c->get_name(), c);
```

这样当我们需要定位某个元素的时候，只需要输入这个星球的名字。在 AVL 平衡二叉树中定义了[ ]运算符，为

```
V operator[](const K &n_key);
```

在添加星球的时候，我们有

```
void add(K n_key, V n_value);
```

这样我们就可以很快添加并定位到这个元素。

AVL 平衡二叉树定义在 AVL.h 中，类型名为 bintree。这是一个链表树形结构，注释中有介绍各个函数的用途。AVL 二叉树不是这次代码的重点，不做赘述。

4、存储作用力的三级索引字典和哈希表结构

在计算过程中，各个行星距离太阳的轨道近似于周期性的，而每次都需要重新计算作用力，这样造成了很大的浪费，因为相同的半径，其作用力是相同的，我们可以考虑使用一个**哈希表字典结构**，把半径作为关键码 key，作用力作为值 value，这样我们就可以用检索获得作用力，其效率大大提高。

但是这种索引有弊端。如果我们对系统计算的精度要求很高，那么这个哈希表就需要很大。但我们知道，一个数组申请如此之大的内存空间是不现实的。

在三维空间中运动，参数至少需要三个：x、y、z 左边值。这很适合于做一个分级索引结构。为了实现分级索引功能，我设计了 hierarchical_index 类。这个类调用了哈希表，其成员为：

```
// 输入信息

void enter(const int &index_1, const int &index_2, const int &index_3,

        const double &value_1, const double &value_2, const double &value_3);
```

```
// 检索代码
void get_data(const int &index_1, const int &index_2, const int &index_3,
        double &value_1, double &value_2, double &value_3);
// constructor
hierarchical_index(int n_size_1, int n_size_2, int n_size_3);
// destructor
~hierarchical_index();

// 这是字典的内容
HashTable<int, HashTable<int, HashTable<int, double *> *> *> *content;
// the size
int size_1, size_2, size_3;
```

在实际的操作中，类型为 double 的坐标将会被放大之后强制转换成 int，然后在字典中寻找对应的加速度大小。

在 planet 类中，对应的成员为

```
hierarchical_index *distance_force_dictionary;
```

这个类成员会存储一个三级索引字典，这个字典的关键码是位置的三维数组，值是对应的三个作用力。因为每个星球的 GM 值不同，所以每个星球的哈希表需要单独制作，如下

```
void planet::calculate_dictionary();
```

这个函数的作用是计算出每个星球的哈希表。为了节省空间，所有的 vector 有如下操作

```
position_vector.resize(3);
force_vector.shrink_to_fit();
```

这样就可以避免多余的空间浪费。

在实际操作过程中，如果

```
#define USE_HASHTABLE
```

则会调用这个函数，否则它不起作用。这个方法会使用大量内存空间，会有很长时间的准备时间。经测试，每个星球需要使用的内存空间大约是 1G。

5、运动与碰撞处理

星体运动的基本思路是用万有引力和牛顿第二定律来处理，算法是蛙跳算法，如果

#define USE_HASHTABLE

则用哈希表处理，否则直接用万有引力计算。重点是处理碰撞。

碰撞分为两种：弹性碰撞与非弹性碰撞。我在处理碰撞的过程中，默认所有的碰撞都是弹性碰撞。碰撞处理函数定义在

void collision(planet *p1, planet *p2);

中。这个函数利用动量守恒定律处理碰撞。需要的信息有两个星球的质量，相对位置，速度等信息。在

void planet::update_position_velocity(int step);

中，程序需要判断两个星球的半径之和是否大于距离，如果没有大于距离，则会发生碰撞。如果没有大于，则不碰撞。碰撞发生的时候，两个行星距离非常近，此时用万有引力公式计算会产生一个很大的数字，这个是不合理的，所以这个时候不会计算万有引力。

6、桶结构部分

为了克服内存占用太多、每次载入都需要预处理很多数据的问题，我做了一个桶结构。桶结构可以把字典内容全部放到硬盘，从而节约内存和处理时间。我把 bucket 定义成了一个类，其成员有：

```
// 读入信息
void enter(const int &index_1, const int &index_2, const int &index_3,
        const double &value_1, const double &value_2, const double &value_3);
// 读出信息
void get_data(const int &index_1, const int &index_2, const int &index_3,
        double &value_1, double &value_2, double &value_3);
// constructor
bucket();
```

在创建 bucket 元素的时候，系统会在根目录下生成一个 bat 批处理文件，这个文件负责创建文件夹。在实际运行中，程序会用三条检索指令在文件夹中生成相应的目录。在读取数据的时候，程序会根据三条检索指令到达相应的目录并读取信息。

7、可视化部分

这部分并非题目要求。但是我认为这个 project 非常适合做可视化，做出 3D 模型可以让程序非常直观。我用了 OpenGL 做可视化，介绍一下可视化提供的功能：

方向键 or WASD：

可以移动摄像机的 position。

HJKL or 鼠标的移动：

可以移动视角。

Q：

加速摄像机移动的速度，摁住不放可以一直增加。

E：

减缓摄像机的移动速度，摁住不放则一直减小。

Z：

增大 step。功能在下面介绍。

X：

减小 step。功能在下面介绍。

N：

减小 period。功能在下面介绍。

M：

增大 period。功能在下面介绍。

，：

减小太阳质量。

。：

增大太阳质量。

C：

创建一个新的星体。可以选择创建的行星的名字、质量、半径、轨道半径、轨道类型、轨道倾角等等参数，这些会在 console 上提供选择。

V：

立即把当前所有行星的位置信息打印在 console 上。

F：

立刻把摄像机的 position 固定到某一星球上，此时 WASD 和方向键失去作用，摄像机随着某一星球运动而运动。可以运动视角。

R：

接触锁定，此时摄像机的 position 不再随某一星球运动。用户重新获得 WASD 和方向键的使用权。

P：

暂停计算。所有星球暂时静止，用户可以移动位置和视角。再按一次恢复。

这是效果图：

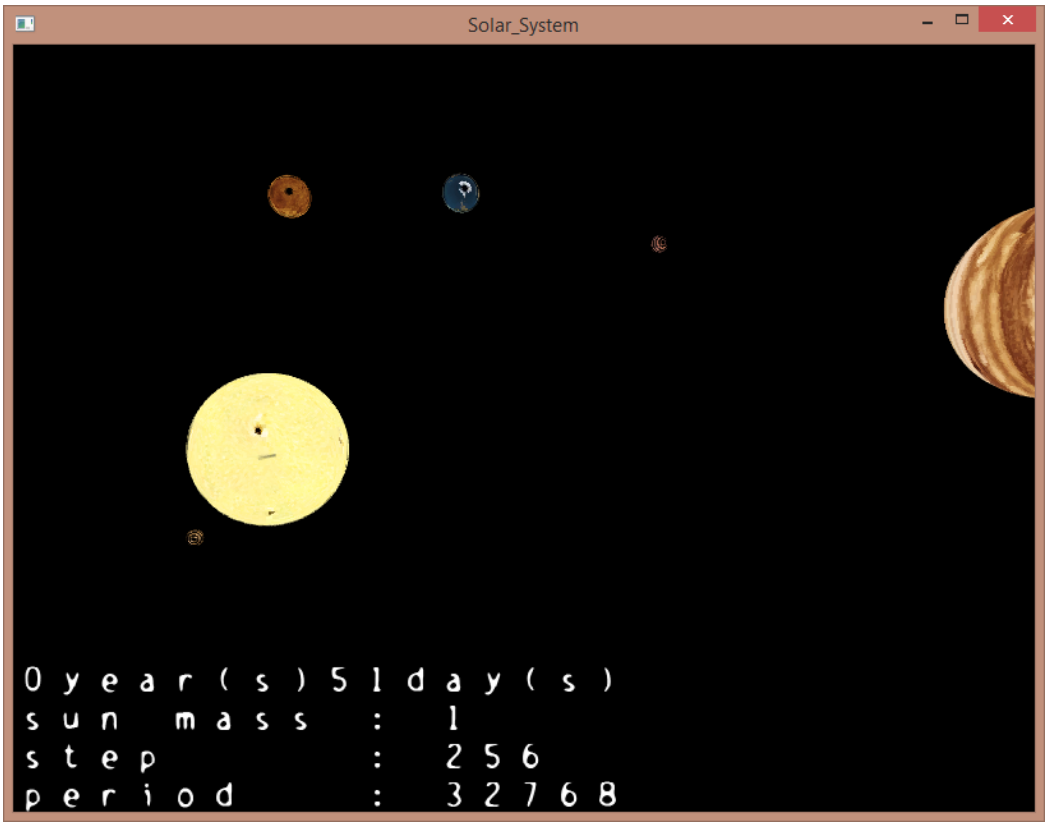

图 1 显示效果图

屏幕下方可以显示星系运行的时间，太阳的质量（以 1 为单位），step 值和 period 值。这四行字会实时更新。

Step 是计算的单位，是计算的步长，单位为 s。

Period 是刷新的单位，即每次画面刷新的步长，单位为 s。

Step 的提高会使计算加速，但是精度会大大下降。Step 降低可以使得计算更加细腻，但是同时也会使画面卡顿，耗费更多时间资源。

Period 的提高使得画面更新率降低，但是可以提高运行的速度。降低 period 使得画面流畅，但是行星的运行变得极为缓慢。

将视角固定在地球上，效果如下

图 2 固定视角的效果图甲

固定视角之后会在 console 中输出选中星体的实时坐标。如下

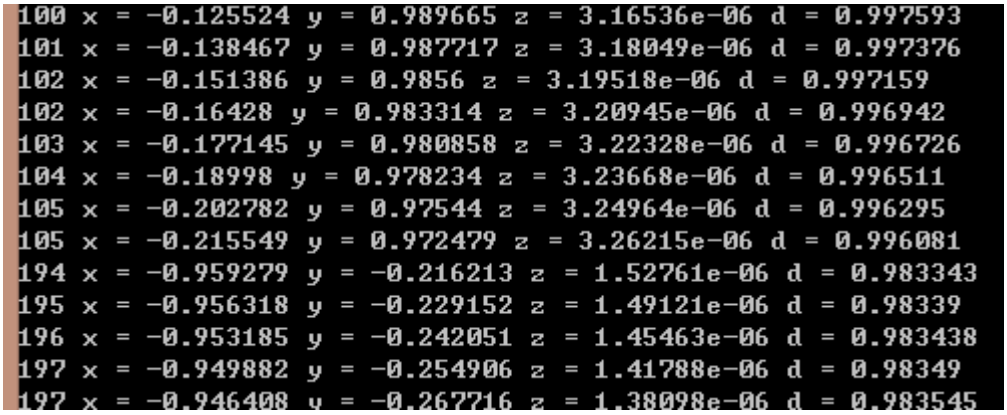

图 3 固定视角的效果图乙

创建新星球的效果图如下

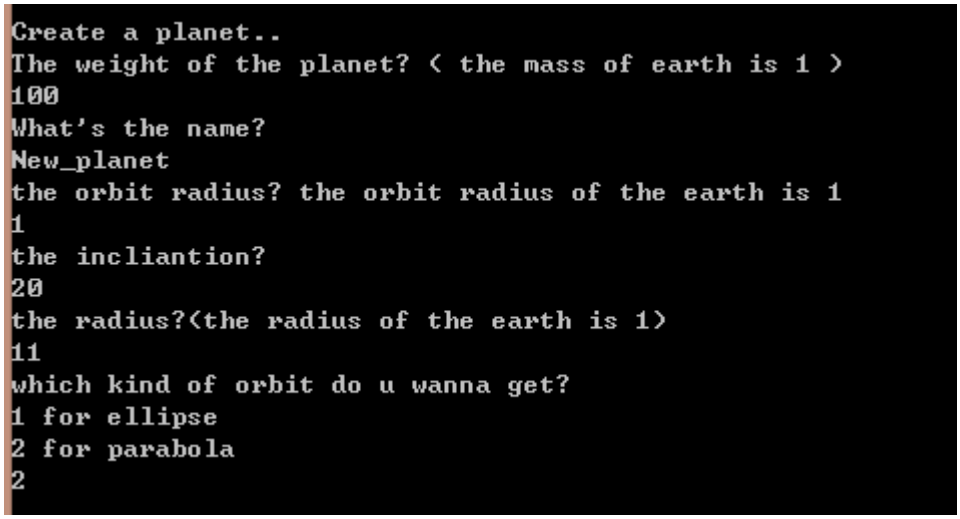

图 4 创建新星体的效果图

新星体会随机分配一个贴图。

8、其他一些数据结构和算法

单链表、双链表、栈、顺序表等结构，在队列、AVL 平衡树中有所应用。

第二部分 模拟实验

1、系统稳定性测试

我做的太阳系是否稳定？轨道闭合性如何？我做了如下测试。

我把计算的精度 step 调整为 1，把轨道数据输出为 xls 文件，在 1048576 天（共计 2872.8 年）中，每天收集一组数据，把地球的轨道通过 excel 打印出来，如下图：

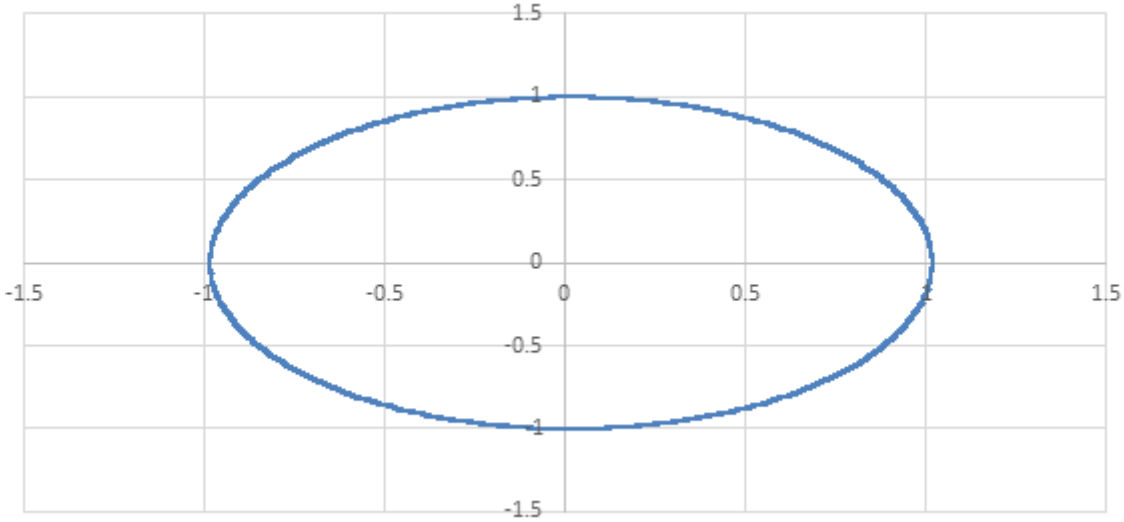

图 5 系统稳定性测试 XOY 平面投影图

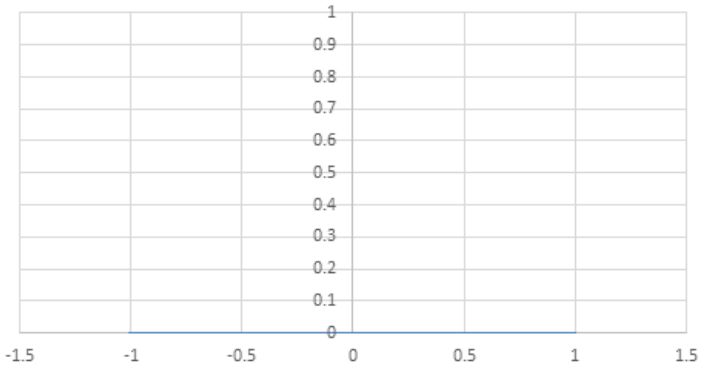

图 6 系统稳定性测试 YOZ 平面投影图

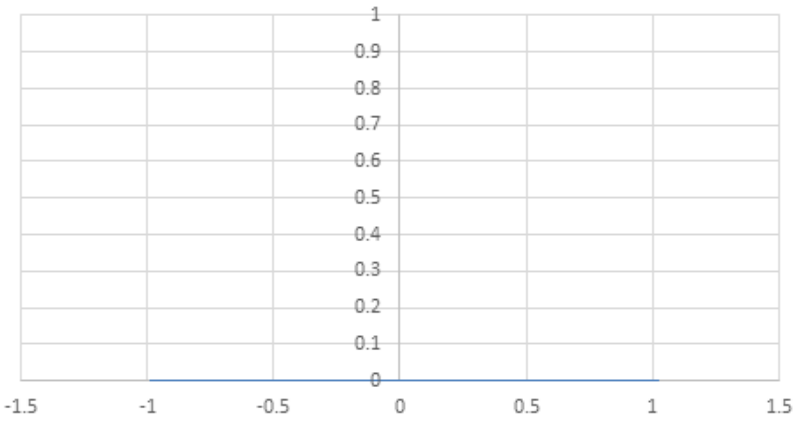

图 7 系统稳定性测试 XOZ 平面投影图

从测试图看，系统运行过程中在 z 方向非常稳定，但是在 x、y 方向上有较为明显的误差。在实际运行中，精度不会达到这个高度，所以运行时间不宜太长，否则累计误差太高。

2、太阳质量改变对于太阳系的影响

如果太阳质量改变，太阳系是否稳定？我做了如下实验。

图 8 太阳质量变化实验效果图

首先我测试了太阳质量减小的几组实验，我将输出的数据做成 excel 表格，制成散点图，观察轨迹。下面的所有实验都以地球为例。

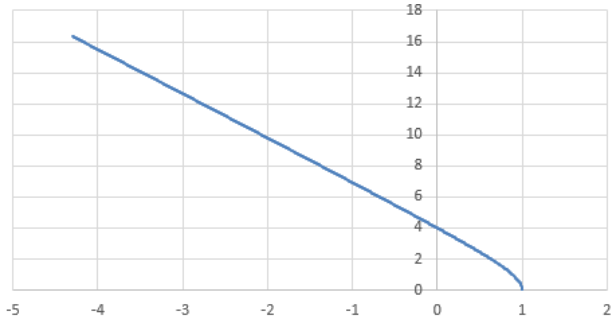

图 9 太阳质量减小为 0.25M 时地球的轨迹

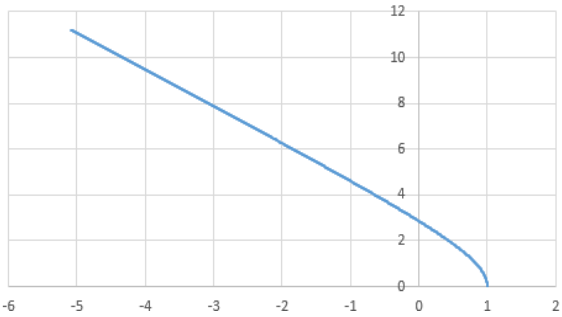

图 10 太阳质量减小为 0.35M 时地球的轨迹

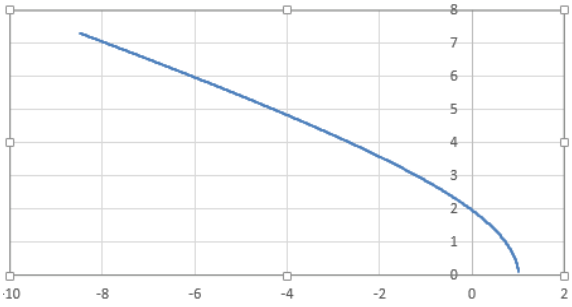

图 11 太阳质量减小为 0.45M 时地球的轨迹

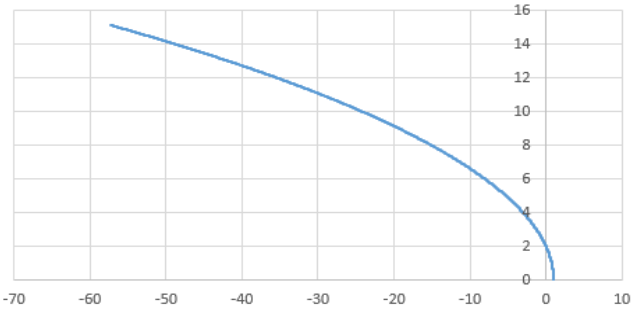

图 12 太阳质量减小为 0. 5M 时地球的轨迹

由图 9 到图 12 可以看出，当太阳质量变为 0.5M 之下的时候，星球的运动轨迹趋向于一条直线，轨道类型为双曲线。当太阳质量逐渐接近 0.5M 的时候，离心率不断下降。当太阳质量为 0.5M 的时候，轨道变成抛物线。

当太阳质量继续上升的时候，轨道的离心率小于 1，呈一个椭圆状。地球的公转周期变长，轨道越来越接近一个圆形。当太阳质量不变的时候，地球的离心率为 0.0167。

上述实验结果和理论计算完全相符。

下面是太阳质量大于 0.5M 且小于等于 M 时的数据。

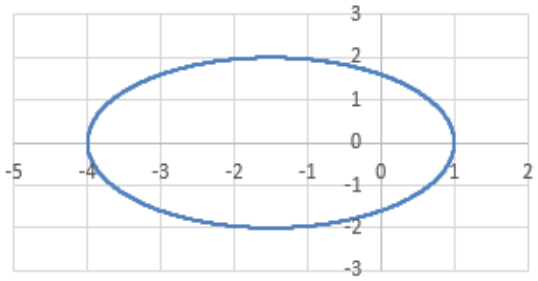

图 13 太阳质量减小为 0. 625M 时地球的轨迹

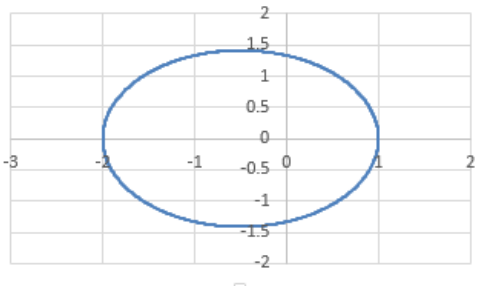

图 14 太阳质量减小为 0. 75M 时地球的轨迹

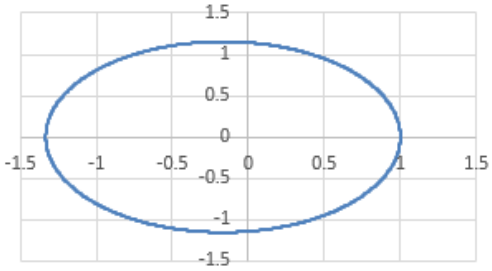

图 15 太阳质量减小为 0. 875M 时地球的轨迹

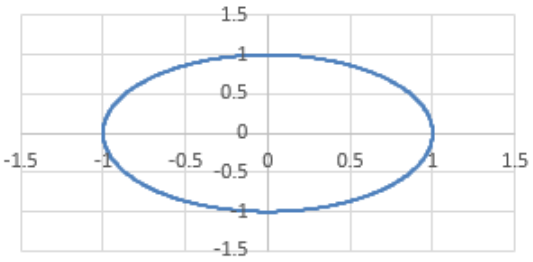

图 16 太阳质量不变时地球的轨迹

从图 13 到图 16 可以看出，随着太阳质量的增大，地球轨道的离心率逐渐下降，在 M=1 的时候，地球的轨道退化为一个圆。

上述实验结果和理论计算完全相符。

下面是几组太阳质量增大的实验图。

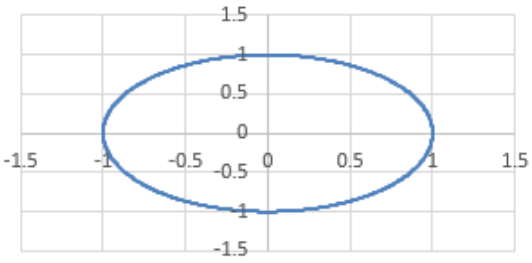

图 17 太阳质量不变时地球的轨迹

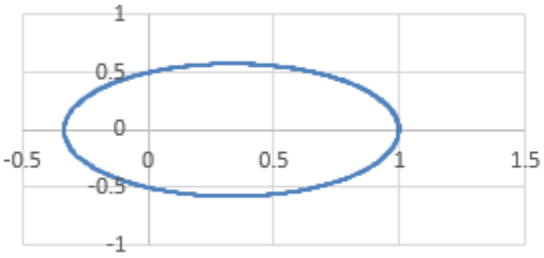

图 18 太阳质量增大到 2M 时地球的运动轨迹

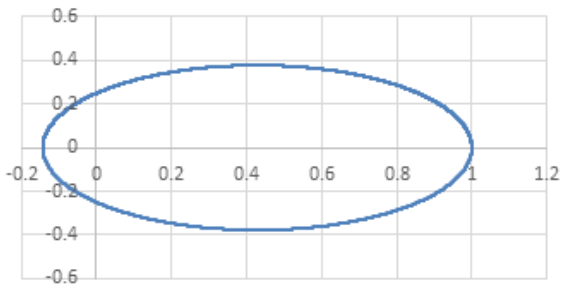

图 19 太阳质量增大到 4M 时地球的运动轨迹

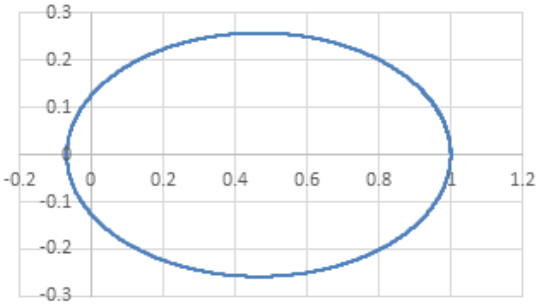

图 20 太阳质量增大到 8M 时地球的运动轨迹

由图 17 到图 20 可以看出，当太阳质量不断增大的时候，地球轨道运动的近日距离不断缩小，最后趋于 0。

上述实验结果和理论计算完全相符。

3、拉格朗日点稳定性实验

拉格朗日点（Lagrangian point）又称平动点（libration points），在天体力学中是限制性三体问题的五个特解。就平面圆型三体问题，1767 年数学家欧拉根据旋转的二体引力场推算出其中三个点（特解）为 L1、L2、L3，1772 年数学家拉格朗日推算出另外两个点（特解）为 L4、L5。例如，两个天体环绕运行，在空间中有五个位置可以放入第三个物体（质量忽略不计），并使其保持在两个天体的相应位置上。理想状态下，两个同轨道物体以相同的周期旋转，两个天体的万有引力与离心力在拉格朗日点平衡，使得第三个物体与前两个物体相对静止。

从理论力学上我们可以证明，二维平面上存在 5 个拉格朗日点。位置如下：

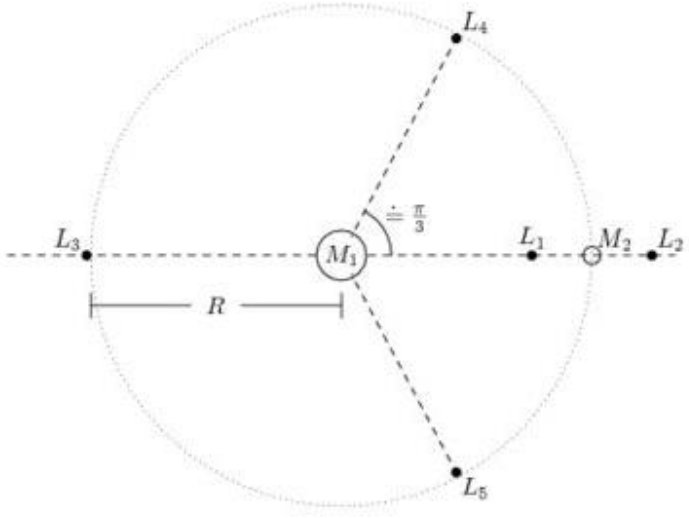

图 21 五个拉格朗日点示意图

对于位于两星连线上的拉格朗日点，设星体的坐标为

$$(x,0)$$

列出平衡方程如下：

$$-\frac{GM_{sun}x}{|x^3|}-\frac{GM_{earth}(x-R_{earth})}{|x-R_{earth}|^3}=\frac{G(M_{earth}+M_{sun})}{R_{earth}^3}\left(x-\frac{M_{earth}}{M_{earth}+M_{sun}}\right)$$

由上式可以解出：

$$x_1=R_{earth}\left(1-\sqrt[3]{\frac{a}{3}}\right)$$

$$x_2=R_{earth}\left(1+\sqrt[3]{\frac{a}{3}}\right)$$

$$x_3=-R_{earth}\left(1+\frac{5a}{12}\right)$$

上式中

$$\mathrm{a}=\frac{M_{earth}}{M_{earth}+M_{sun}}$$

线外有两个平衡点，设其坐标为

$$(x,y)$$

可以列出平衡方程如下

$$r_{sun}=\sqrt[2]{x^2+y^2}$$

$$r_{sun}=\sqrt[2]{(x-R_{earth})^2+y^2}$$

$$r=\sqrt[2]{[(x-aR_{earth})]^2+y^2}$$

$$r^2+{r_{sun}}^2-2r_{sun}r\cos\theta_{sun}=(aR_{earth})^2$$

$${r^2+r_{earth}}^2-2r_{earth}r\cos\theta_{earth}=(R_{earth}-aR_{earth})^2$$

$$F_{sun}=\frac{GM_{sun}}{r_{sun}^2}$$

$$F_{earth}=\frac{GM_{earth}}{r_{earth}^2}$$

$$F_{sun}\sin\theta_{sun}=F_{earth}\sin\theta_{earth}$$

$$\frac{G(M_{earth}+M_{sun})}{R_{earth}^3}r=F_{sun}\cos\theta_{sun}-F_{earth}\cos\theta_{earth}$$

解上述方程得

$$\left(\frac{R_{earth}}{2} * \frac{M_{sun} - M_{earth}}{M_{sun} + M_{earth}}, \frac{\sqrt{3}}{2} R_{earth}\right)$$

$$\left(\frac{R_{earth}}{2} * \frac{M_{sun} - M_{earth}}{M_{sun} + M_{earth}}, -\frac{\sqrt{3}}{2} R_{earth}\right)$$

理论还可以证明，太阳与地球的五个拉格朗日点只有线外的两个点是稳定的。线上的三个点，虽然不稳定，但是却可以在旋转过程中做一种特殊的振动，称为 halo orbit。为了证明上述理论，我做了如下实验。

为了保证计算的精度，太阳和地球的质量比被重新设定为 100：1，这样可以保证我们在计算过程中不会因为过高的引力而丧失精度。我用上述设定测试了 5 个拉格朗日点的稳定性，把数据输出，用 excel 绘制出轨道图。

第一次实验我的精度设置是 step=0.1，换言之，每 0.1s 计算一次位置。这个精度下我绘制了卫星第一圈的运行情况，下面是三个不稳定平衡点的运动图像。

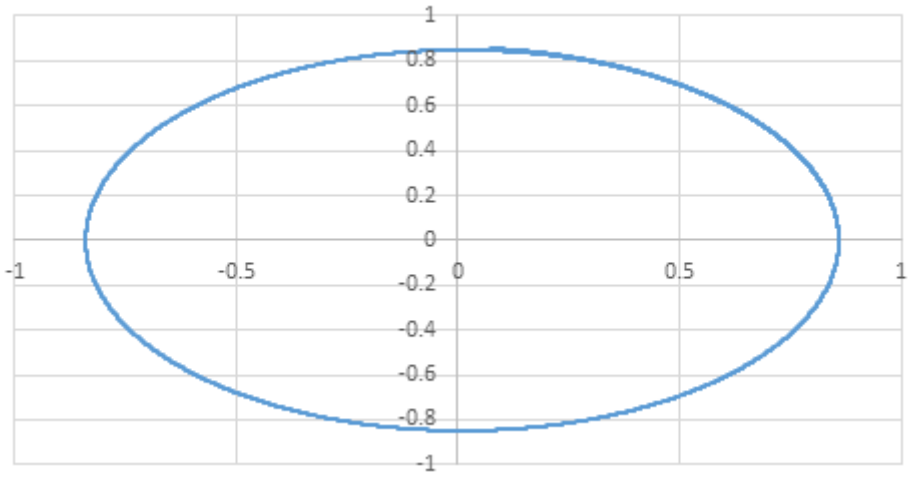

图 22 第一拉格朗日点卫星运行情况(10^4 h)

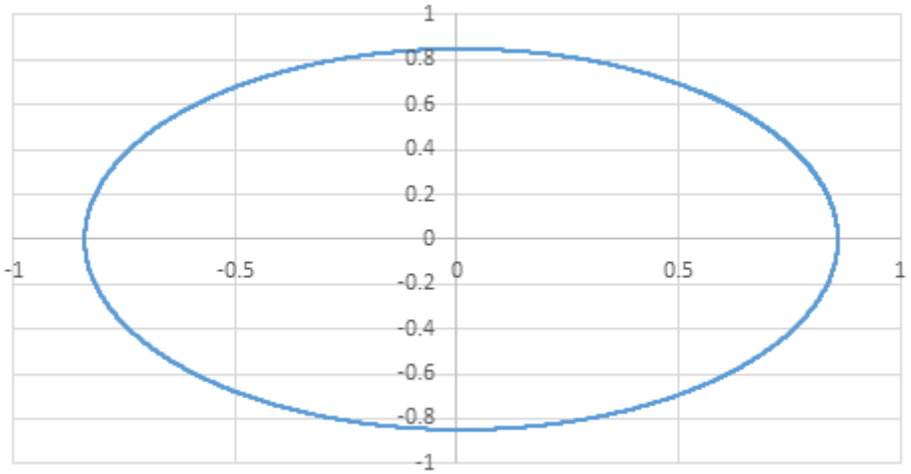

图 23 第二拉格朗日点卫星运行情况(10^4 h)

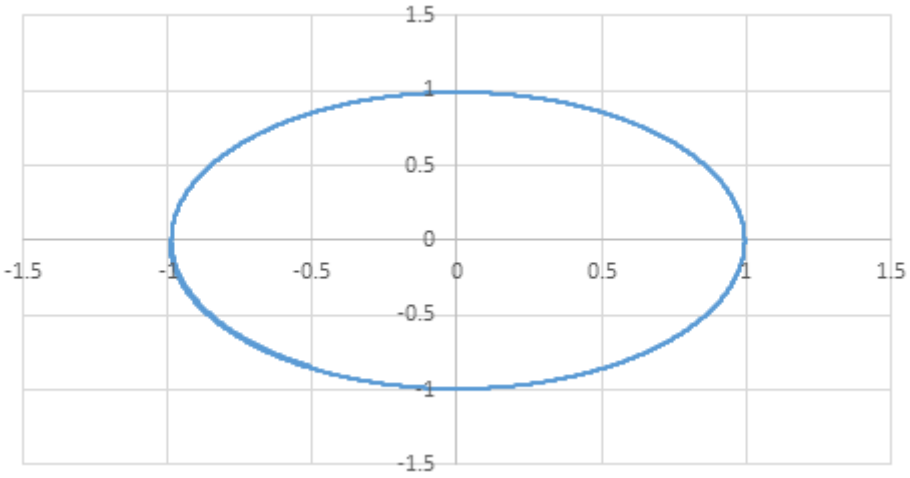

图 24 第三拉格朗日点卫星运行情况(10^4 h)

可以看出，上图虽然有一些小的误差，总体是可以接受的。但是接下来就有非常大的误差。

下图是运行十万小时的效果图。

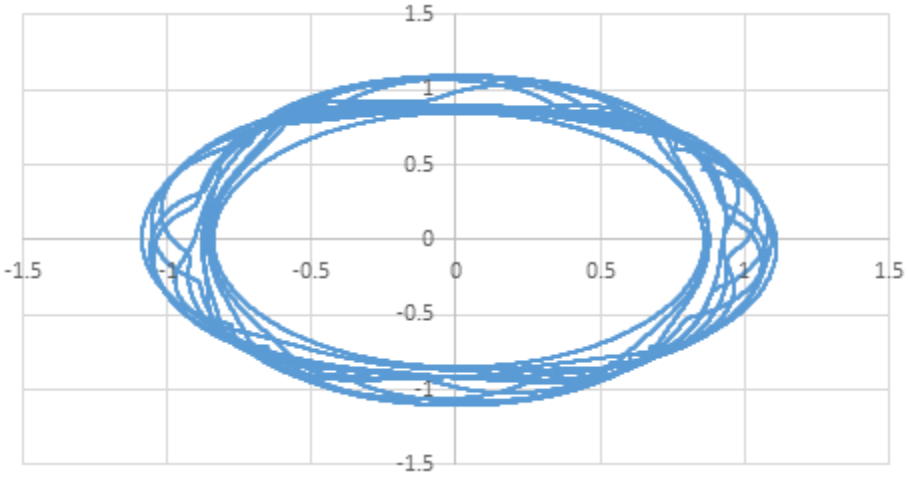

图 25 第一拉格朗日点运行效果图（10^5 h）

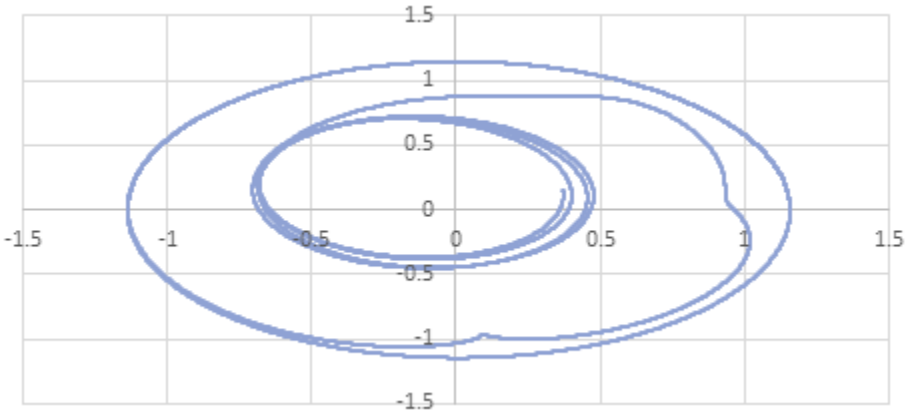

图 26 第二拉格朗日点卫星运行情况(10^5 h)

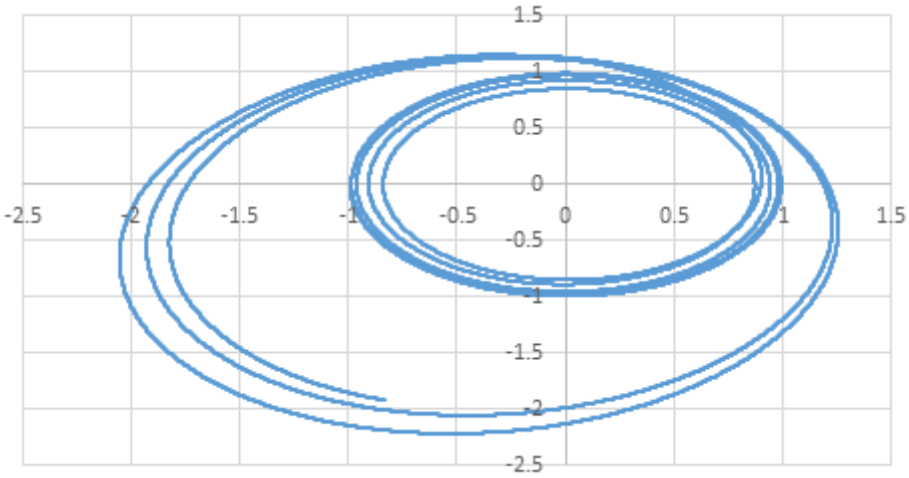

图 27 第三拉格朗日点卫星运行情况(10^5 h)

可以看出卫星轨道已经完全脱离了第一拉格朗日点。在接下来的运动中，卫星的轨道是否毫无规律呢？并不是！

在看似毫无规律的运动中，卫星表现出了一种惊人的周期性运动。为了更好的表现这种周期运动，我将卫星相对其平衡位置的坐标绘制出来（注意这个平衡位置是一直在变的，这是一个移动的参考系）。

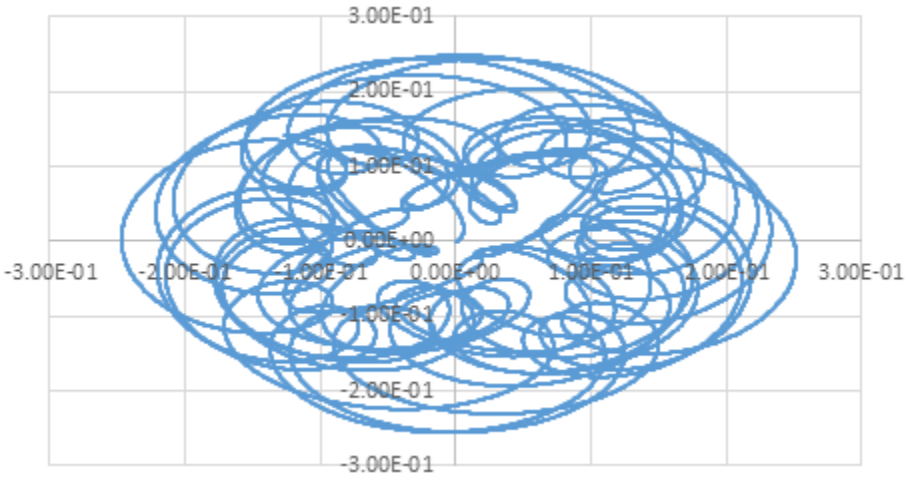

图 28 第一拉格朗日点卫星的相对坐标变化(10^5 h)

更直观的是相对距离和时间的关系图。

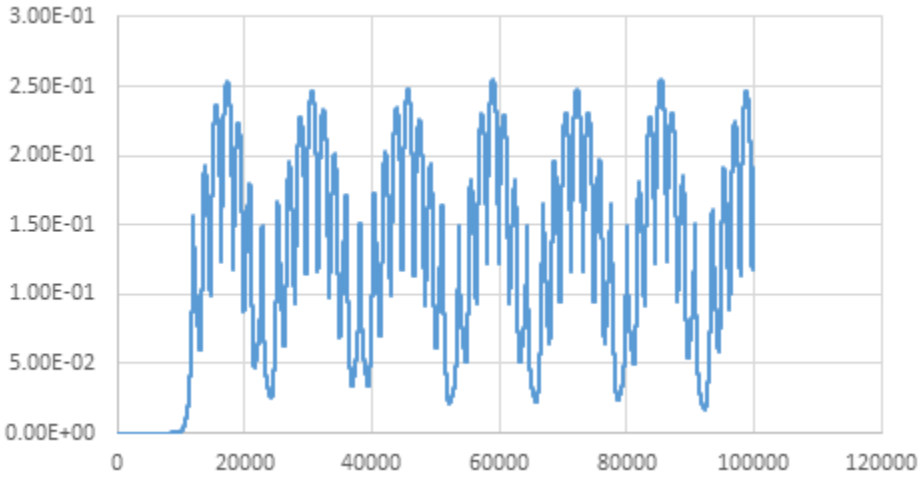

图 29 第一拉格朗日点相对距离和时间的关系(10^5 h)

在更大的时间尺度上，有下图所示的图像。

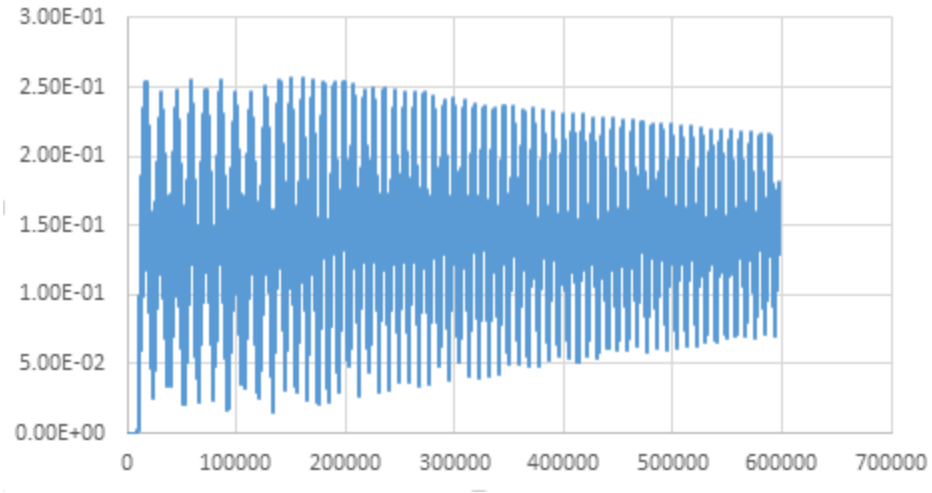

图 30 第一拉格朗日点相对距离和时间的关系(6 * 10^5 h)

我们可以看出，其实拉格朗日点偏离之后的三体运动，也并非完全没有规律可循。它会在第一、第二、第三拉格朗日轨道上来回跳跃，其运动并不是精确的周期运动，但大约 16.3 年会回到初始的情况。

我们设想，如果我们发射的某颗卫星在第一、第二或者第三拉格朗日点的 halo 轨道上运动，丧失了动力而陷入混沌，我们可以根据它三体运动的周期（不精确的周期），大致判断它的位置，从而便于我们去寻找这个周期。

囿于实验精度的限制，我们没能找到 halo 轨道的身影。希望以后有更好的方法去模拟 halo 轨道。

第四和第五拉格朗日点是稳定的。在很长时间的运动之后，它依然保持了稳定。我绘制了第四和第五拉格朗日点的图像。

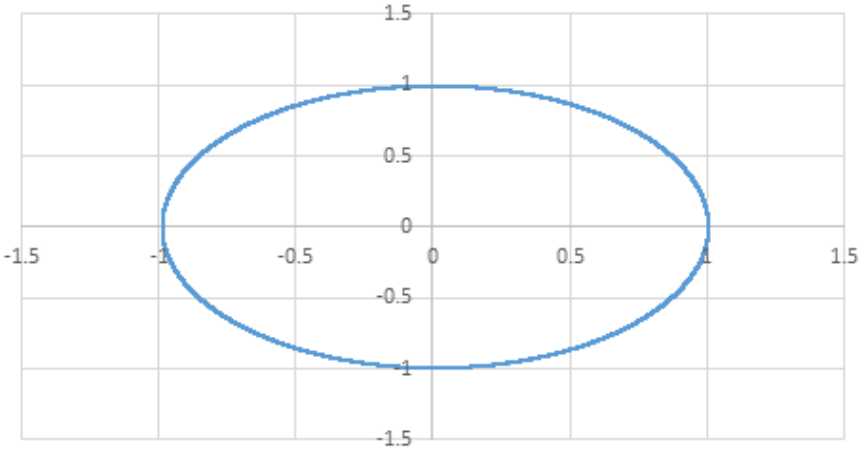

图 31 第四拉格朗日点轨道图(10^5 h)

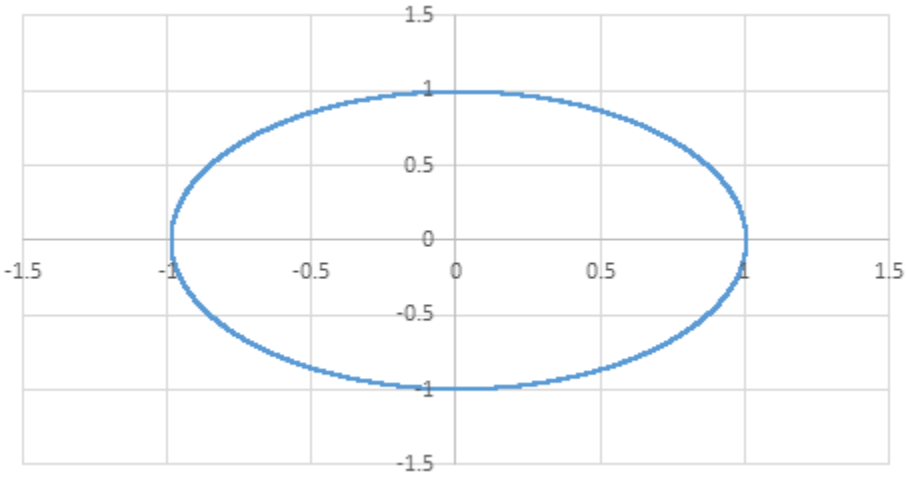

图 32 第五拉格朗日点的轨道图(10^5 h)

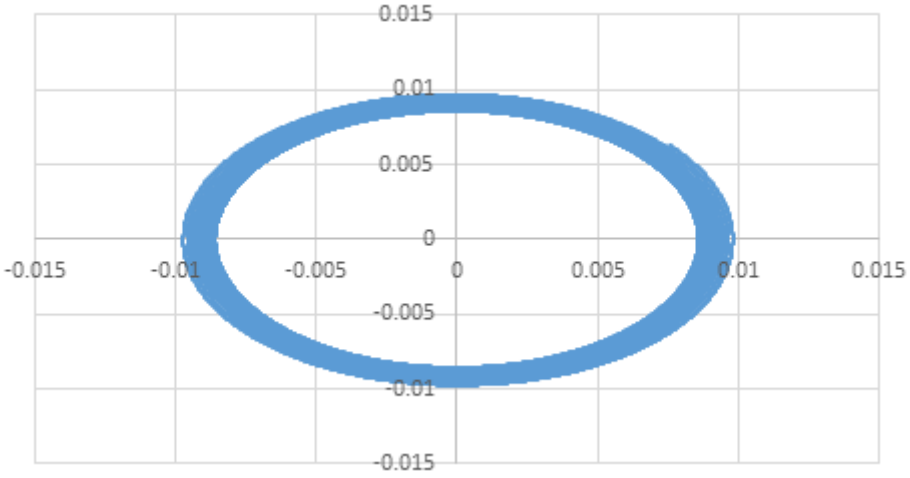

图 33 第四拉格朗日点相对坐标图(10^5 h)

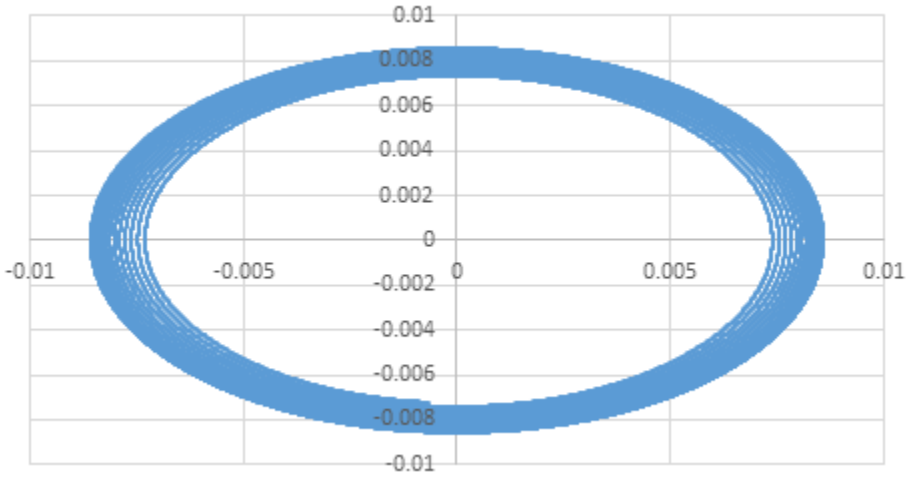

图 34 第五拉格朗日点相对坐标图(10^5 h)

可以看出，第四和第五拉格朗日点非常稳定，在十万小时的模拟中，最大误差不大于百分之一。而且随着时间的增加，误差不会累积。

如果我们发射卫星到第四和第五拉格朗日点上，我们不需要额外的能量，就能让卫星稳定在平衡点附近。

4、加入一颗行星之后太阳系的稳定性

如果我们在太阳系中加入一颗行星，太阳系还会维持稳定吗？太阳系中的行星会不会发生偏移？为了测试太阳系的稳定性，我做了如下实验。

在太阳系模拟一段时间之后，突然在地球和火星之间加入一颗质量相对较大（本次实验中我假设它的质量为 10、100 和 1000 倍的地球质量）的行星。我打印了地球和火星的轨道。

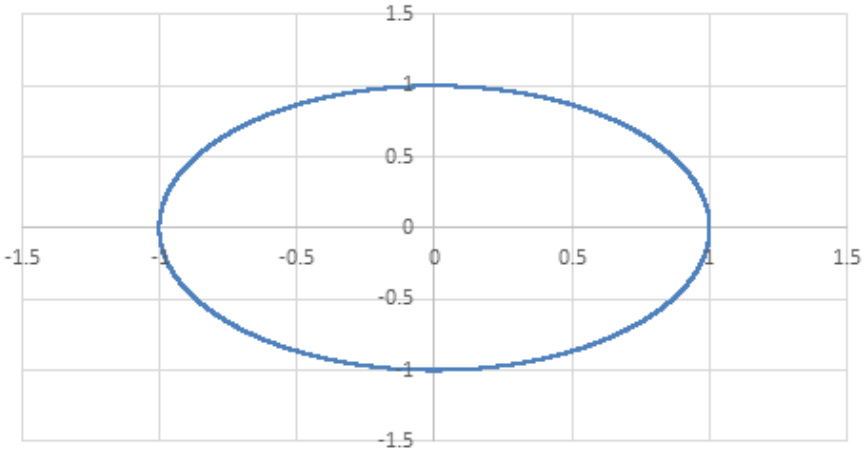

图 35 运行一年后突然加入一颗质量为 100 倍地球质量的行星

地球运行轨道图

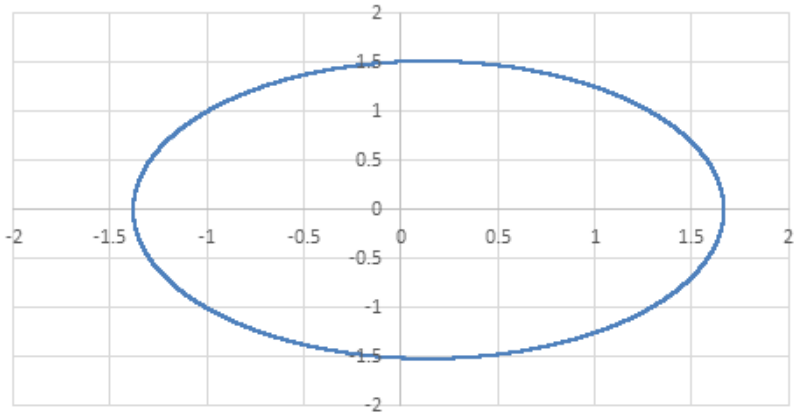

图 36 运行一年后突然加入一颗质量为 100 倍地球质量的行星

火星运行轨道图

这样显示效果很不好，毕竟 100 倍地球质量的行星对地球和火星造成的影响不至于使其轨道发生明显的变形。我们不妨看看地球和火星在加入行星之后距离太阳距离绝对值随时间的变化关系。

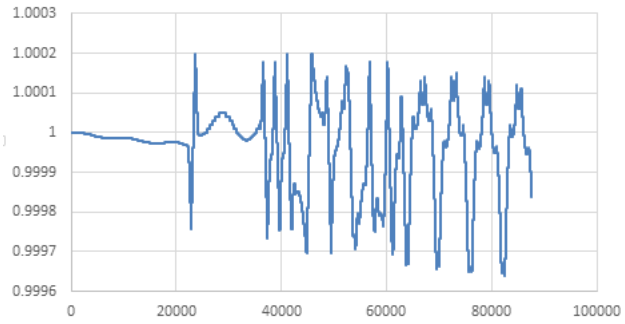

图 37 加入质量为 10 倍地球质量之后地球日心距-时间 关系图

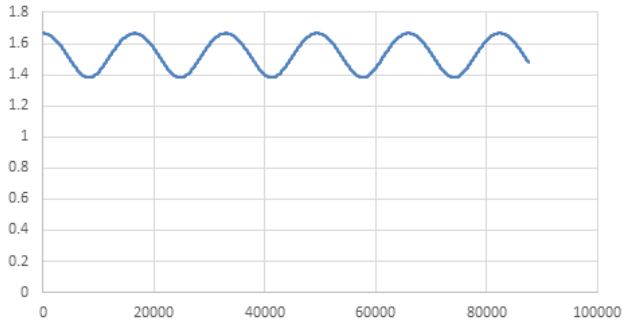

图 38 加入质量为 10 倍地球质量之后火星日心距-时间 关系图

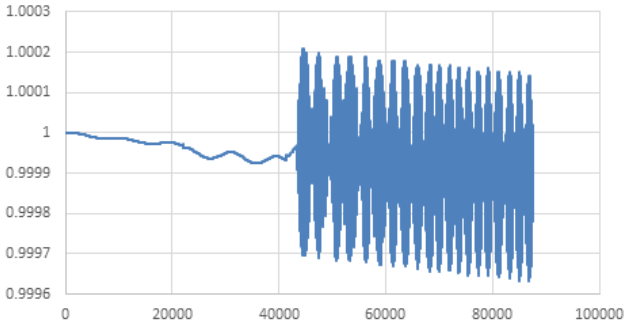

图 39 加入质量为 100 倍地球质量之后地球日心距-时间 关系图

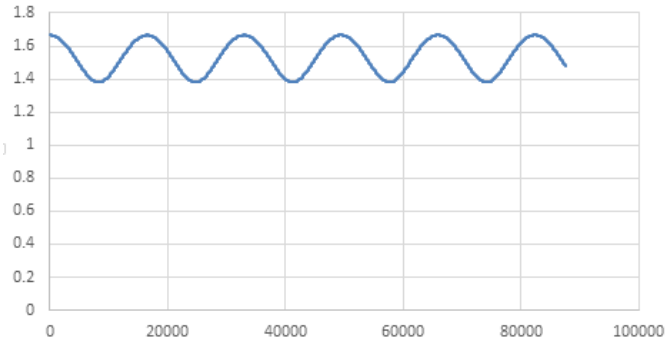

图 40 加入质量为 100 倍地球质量之后火星日心距-时间 关系图

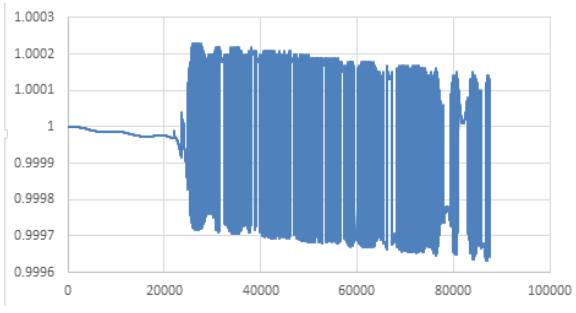

图 41 加入质量为 1000 倍地球质量之后地球日心距-时间 关系图

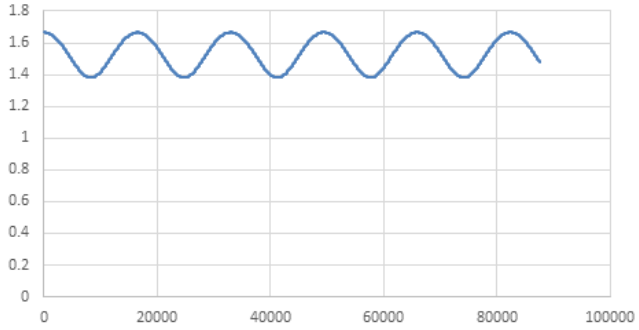

图 42 加入质量为 1000 倍地球质量之后火星日心距-时间 关系图

从图中我们可以看出，新的行星产生之后，地球和火星的轨道都发生了不同程度的偏离，而且可以预测，随着行星质量的增加，地球和火星的轨道偏离程度会越来越大。

这种偏离没有固定的周期，但是会有一个大致的周期。

5、行星碰撞碰撞实验

如果有一颗彗星撞击了地球，地球会怎么样？我用模拟实验实现了这一撞击。

为使得撞击比较直观，撞击的慧星质量设置为地球的一倍。其轨道是一个抛物线，焦点是太阳。其轨道和黄道面的焦点距离太阳的距离是地日距离的一倍。设置好恰当的初始值之后，彗星就可以击中地球。

为了研究不同的彗星撞击地球造成的影响，我分了四种情况，每种情况下彗星的轨道面和黄道面的夹角不同，分别是 0°（和黄道面共面），30°，60°，90°。每种情况把地球的位置信息输出，投影到三个平面上，下面是效果图。

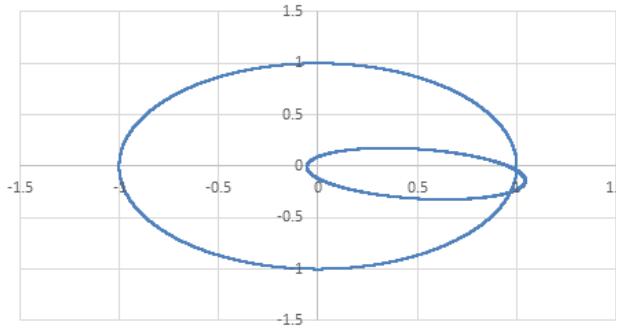

图 43 彗星轨道面同黄道面成 0°，质量为地球 0.2 倍撞击地球之后地球轨道的 XOY 投影

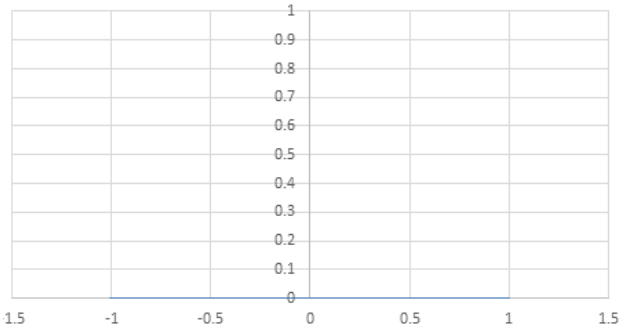

图 44 彗星轨道面同黄道面成 0°，质量为地球 0.2 倍撞击地球之后地球轨道的 YOZ 投影

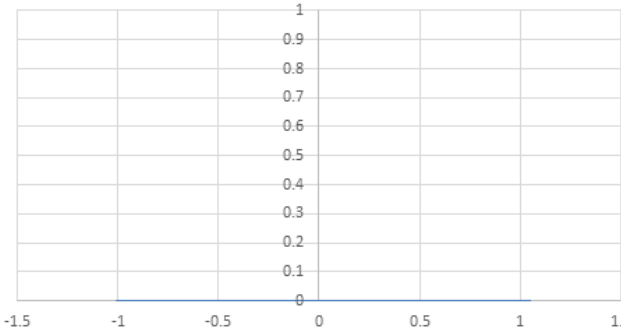

图 45 彗星轨道面同黄道面成 0°，质量为地球 0.2 倍撞击地球之后地球轨道的 XOZ 投影

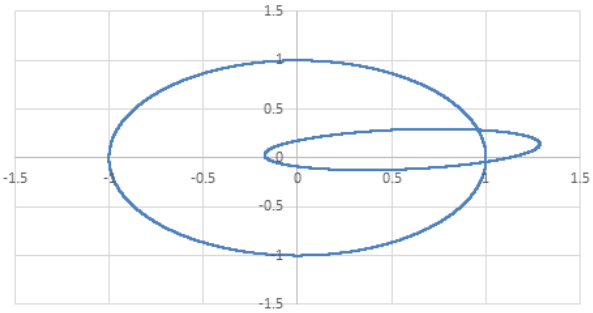

图 46 彗星轨道面同黄道面成 45°，质量为地球 1 倍撞击地球之后地球轨道的 XOY 投影

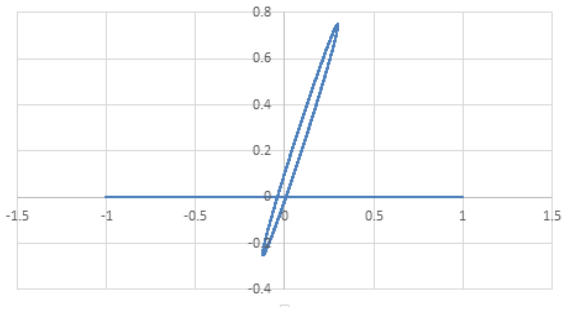

图 47 彗星轨道面同黄道面成 45°，质量为地球 1 倍撞击地球之后地球轨道的 XOY 投影

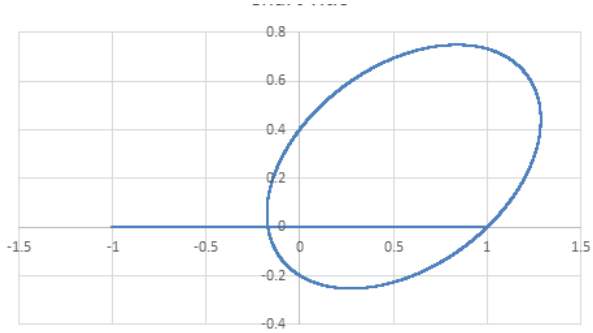

图 48 彗星轨道面同黄道面成 45°，质量为地球 1 倍撞击地球之后地球轨道的 XOY 投影

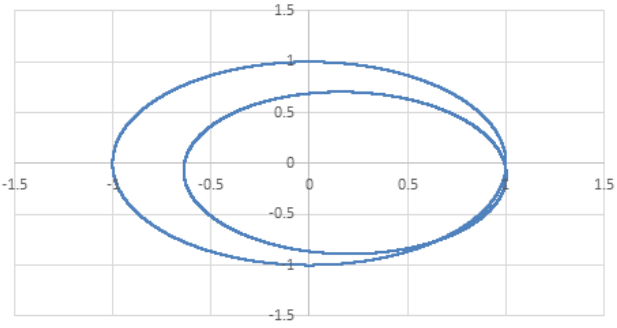

图 49 彗星轨道面同黄道面成 90°，质量为地球 0.1 倍撞击地球之后地球轨道的 XOY 投影

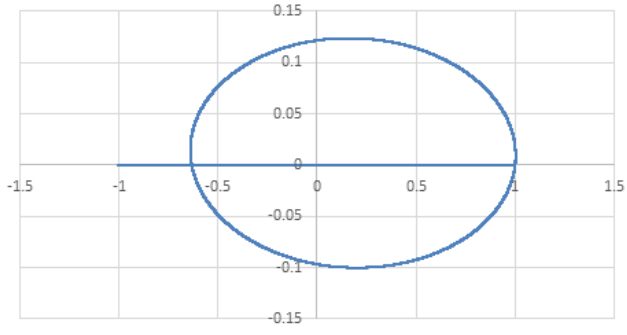

图 50 彗星轨道面同黄道面成 90°，质量为地球 0.1 倍撞击地球之后地球轨道的 YOZ 投影

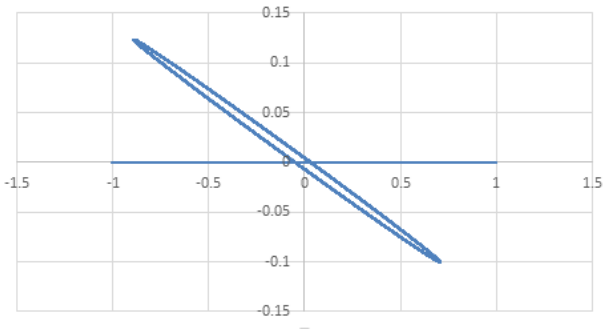

图 51 彗星轨道面同黄道面成 90°，质量为地球 0.1 倍撞击地球之后地球轨道的 XOZ 投影

可以看出，在被撞击之后，地球的轨道依然保持着周期性。在误差允许的范围内，地球的轨道依然保持了闭合性。如果撞击角度不为 0 的话，地球将偏离黄道面，在 z 轴上做周期运动。

地球的运动在时间上也保持着周期性，随便举一个例子，如图 52

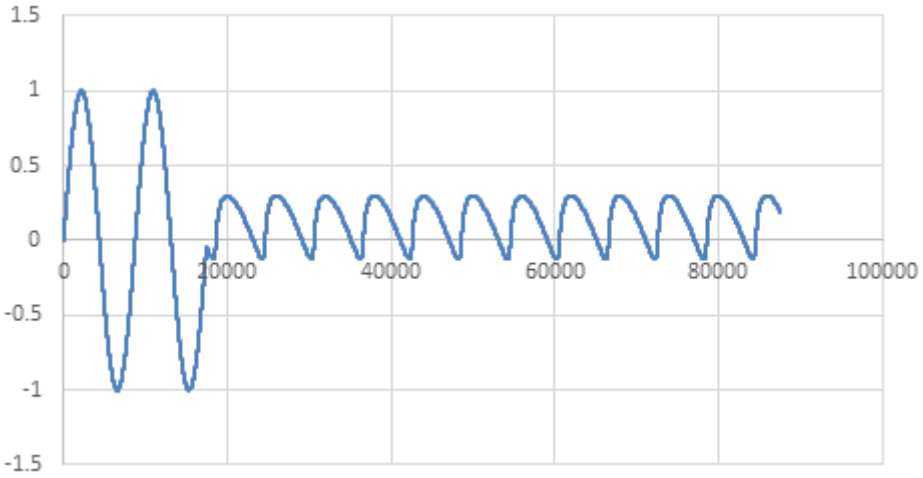

图 52 彗星轨道面同黄道面成 45°，质量为地球 1 倍撞击地球之后地球 y 坐标-时间关系图像

这个结论是符合物理定律的。

6、宇宙尘埃对行星运动的影响

如果宇宙中处处充满尘埃，行星的运动轨迹会是什么样的呢？我做了实验，试验中地球会受到一个和速度成正比的阻力，下面是地球的轨道：

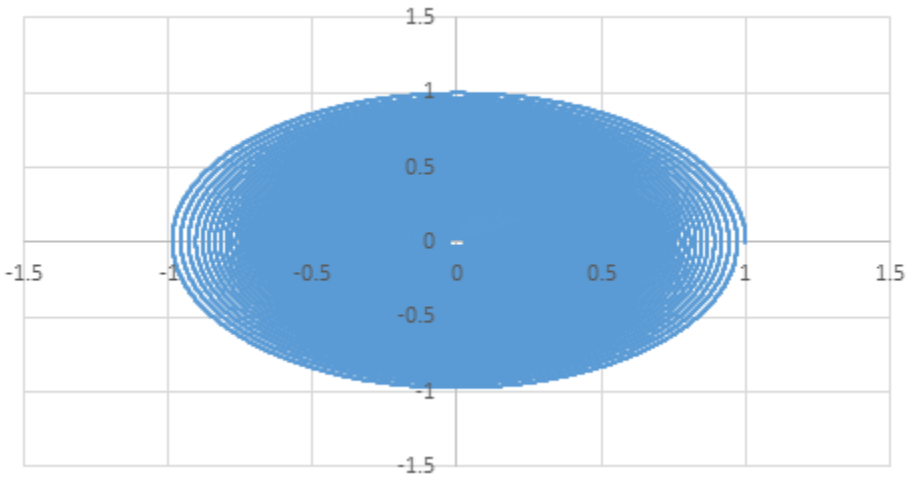

图 53 宇宙尘埃对地球运动的影响

可见，在加入宇宙尘埃之后，地球的轨道半径会慢慢缩小，直到撞到太阳。

第三部分 优化和改进

1、关于碰撞的处理

碰撞处理是一个很为难的问题。由于计算机模拟只能模拟蛙跳式的行星运动，所以不可避免的会产生两个行星之间的距离非常非常小，最后产生了巨大的引力，导致速度暴涨，飞出太阳系。这种在真实物理中是不可能发生的。为此我做了两项调整。

第一项调整是，当两颗星星之间的距离小于半径之和的 50 倍的时候，提高计算精度。此时行星之间的引力太大，需要降低 step。我在测试中将 step 降低到了原来的十分之一（为了避免 int 值不能被 10 整除，我把 int step 换作了 double double_step 来运算），有不错的效果。星体之间的碰撞不再会发生“纠缠不休”的情况，也不会突然速度暴涨。同理，当两个行星距离小于半径之和的 10 倍的时候，会调用另一个循环，计算精度会提高 50 倍。

第二项调整是，当两个星体距离小于半径之和的时候，调用 collision 函数，使两颗行星发生碰撞。Collision 会考虑两个星体的星体的相对距离，相对质量大小，连心线的角度等因素，然后用动量守恒定律算出碰撞结果。

做了上述调整之后，对于质量相当的星体碰撞，效果很好。看起来和真实世界中的弹性碰撞并没有很大的出入。但是如果遇到了例如地球和太阳这样质量悬殊的星体碰撞，上述手段不再起作用，仍然会出现能量不守恒的情况。

看起来，处理质量悬殊星体的碰撞，使用物理的方法求碰撞的理论解似乎不可避免了。我暂时没有想到很好的方法可以通过纯粹的模拟计算这种碰撞。

2、关于显示比例

星体的半径、质量、轨道等等数据，我都是从天文网站找到的，但是遇到了另一个问题，就是星体之间的距离相比星体的半径实在是太大了。如果让星体的半径放大，则距离太大，不好观察；如果星体之间的距离在屏幕上比例适当，则半径太小，除了太阳，所有星体都不可见。

为了解决这个问题，我把所有的星体全部放大 1500 倍，这时候我们就可以看到所有的星体了。

但是这又带来另一个问题，就是太阳实在太大了，这个时候太阳的半径已经比很多行星的轨道半径都大了，这个显示比例还是不合理。所以太阳我单独处理，放大了 50 倍。

这样图像的显示比例就很合适了，具体的效果见图 1。

而我也认识到，原来科普所用的那些图片，居然没有一张是真实的比例！真实比例下，行星变得非常非常渺小。

3、关于计算精度和显示精度的协调

可视化有一个问题，就是屏幕的刷新率。我把这个刷新周期定义为 period。

行星模拟使用的是蛙跳算法，我定义了一个变量 step，这个是蛙跳算法的步长。

写代码的时候我有想过，程序的刷新率应该设成多少？我想过设置一个固定的刷新频率，但是固定的刷新频率势必会让计算精度妥协。另外一种思路是设置一个联动的调节机制，可以同步调整 step 和 period，这样我们就可以获得一个比较稳定的显示效果：当我们使用较高的精度的时候，period 相应变小；当使用较低精度的时候，period 也会相应增加。

但是这个程序可能会用作不同的用途，如果我们用它来展示，那么较好的显示效果是很重要的。如果我们用它来计算，那么可视化仅仅是一个辅助的工具，那么很高的刷新率反而降低了计算的速度。权衡之下，我选择了可以分别调整 step 和 period 的方式。这样用途的决定权交给了用户。希望做一些计算，就使用低的刷新率，高的精度。如果用作展示，则可以选择高的刷新率，低的精度。

任何时候，只要暂停计算，刷新率都会大大提高。

（事实上，我最后并没有用可视化的代码做模拟实验，而是用一个非可视化的代码在服务器中运行的。）

4、关于计算精度的动态调整

计算精度的调整上面已经叙述完毕，这里总结一下：

A．计算步长的调整：

分为四级：

1. 行星距离大于半径之和的 50 倍，不做调整
2. 行星距离大于半径之和的 10 倍，小于 50 倍，步长缩为十分之一
3. 行星距离大于半径之和的 1 倍，小于 10 倍，步长缩为五十分之一
4. 行星距离小于半径之和的 1 倍，调用碰撞函数

B．需要考虑引力的行星列表会在一定量的计算之后更新，引力过小的同级行星会被踢出这个列表，低级行星的引力不会考虑在内。

5、关于贴图和模型

贴图我原来使用的是 BMP 位图格式导入的，这样显示的效果很好，但是也有一个问题，就是这种格式的贴图效率很低。显卡渲染贴图用时很短，最消耗时间的是导入过程。所以后来我用 compressor 压缩了图片，用 DDS 格式导入了贴图。

模型是我用 blender 做的。模型很粗糙（你可以看到一些比较明显的棱角），这也是为了提高运算速度。

6、关于三级索引哈希表

三级索引做的不是很成功！虽然基本实现了功能，但是表现实在是太差了。

1、 由于星体的空间运动是一个三维运动，所以索引需要接受三个参数，返回三个参数。这样就需要很大的存储空间。在每个坐标被分成 200 等份的情况下，索引表的建立需要 700MB 的内存空间。而二百等份对于星体运动而言，误差非常之大。
2、 由于索引表被拆成了 100 * 100 * 100 个哈希表，所以这个代码对内存的利用率很低，有很多碎片无法重新使用。虽然内存很大，但是其实实际使用的时候，很大的内存都是被白白浪费掉的。
3、 虽然付出了惨重的代价，但是程序的计算速度却并没有因此而有明显提高。检索也是需要时间的，在计算速度上，检索并没有比直接计算节省太多时间。
4、 如果要提高检索效率，就需要降低索引的级数。但是这并不现实！内存中一次申请太大的数组是不被允许的。
5、 作为一个索引表，首先需要保证待检索的内容是有限的。这样行星的运动范围就必须得严格控制，不能太大。如果范围大的话，检索表就需要随之扩大，需要重新加载。非常之不方便。

种种不便告诉我：三维行星的运动是不适合使用索引的。后来我做了一个二维的索引表。在二维索引表的情况下，运行状况如图所示：

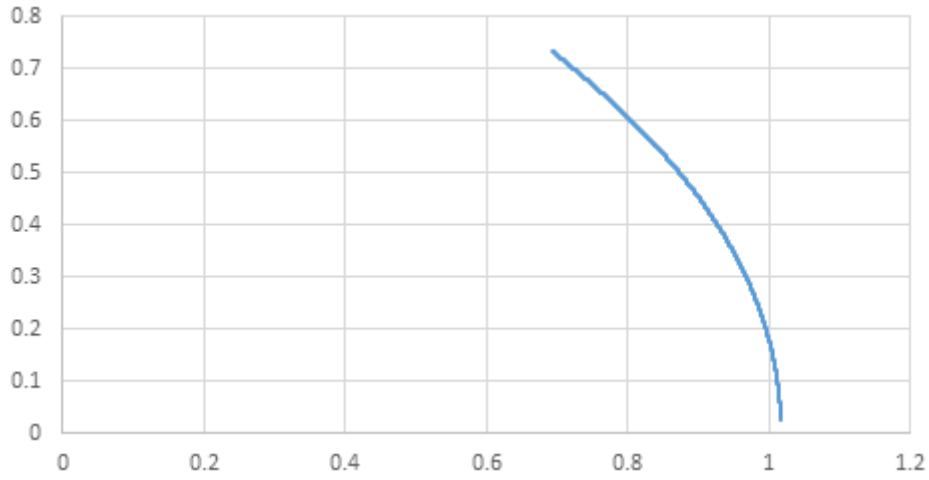

图 54 哈希表测试图线

因为内存限制，我没有设置很大的哈希表，所以只能跑一个小圆弧。但是我们可以看出，行星的运动轨迹基本保持了椭圆。囿于精度，可能不是闭合的。

这种算法相对直接计算，速度有了明显提升。但是缺点还是很多，尚未克服。

7、关于桶结构

这个算法实在太慢了。读取外存所浪费的时间已经足够蛙跳算法算出加速度了。但是桶结构也有一些优点。

1、桶结构比较省内存，不需要调用很多内存
2、桶结构不需要预处理很多数据，一次处理，可以一直使用
3、桶结构没有容量限制，可以随意添加元素进入

虽然如此，但是它真的是太慢了。我放弃了这一算法。

8、关于卫星的设置

由于我放大了比例尺，所有的行星的显示半径都比较大，所以如果我设置了行星的卫星的话，卫星就会缩进行星的半径中去，显示效果很不好。

我在程序中只加入了地球的卫星：月球。如果你想看到月球的话，必须把视角调到地球内部才能看到月球。

如果你想看更多的卫星运行情况的话，可以在 load_planet.cpp 中略做修改，同时切记调高计算精度，否则卫星很容易脱离母星。

参考文献和链结：

1、 台北市天文馆 2015 年天文年鉴（参数来源）
2、 OpenGL Tutorial (http://www.opengl-tutorial.org/)
3、 《C++ Primer》
4、 《数据结构与算法：c 语言描述》