Project 3: The application of a symplectic method in the Kirkwood gaps problem

Gengpu Li[1] *✉ **and Yinan Chen**[1] *✉

[1]Zhiyuan College, Physics, Shanghai Jiaotong University, Shanghai, CN
*These authors contributed equally.

Abstract

In this project, we simulate the Kirkwood gaps which represents the absence of asteroids at certain semi-majors. Firstly, we will introduce some observed results of these gaps, as well as the theoretical method and approximations we conduct in our simulation. Furthermore, we will illustrate the symplectic algorithm we conduct in order to make the simulation more accurate. Finally we will show various results of our simulation and compare them to the observed values.

Kirkwood gaps| Symplectic integrators
Correspondence: ***ligengpu_lim@sjtu.edu.cn yinan1999@sjtu.edu.cn***

1. INTRODUCTION.

Kirkwood gaps are absence of some values of the semi-major axis of the asteroid orbits. This phenomenon is due to the orbit resonance which will repel the astoroid away from certain semi-major states. The resonance source for the asteriod belt is Jupiter, and since the mass of Jupiter is far less than that of the sun, so the resonace effect requires a long period of observation as well as any possible numerical calculation.

Despite, there are some typical resonance orbits. One of them is the orbit with semimajor axis near 2.50AU and period 3.95 years. Notics its period is three times that of the Jupiter and hence this absent orbit corresponds to a 3:1 orbital resonance. Note that not all resonance will result in a depletion of asteriods, but what is observed shows the weaker resonances, compared with this 3:1 resonaces, lead only to a depletion of asteroids.

The most prominent Kirkwood gaps are located at mean orbit radii of: 2.06AU(4:1resonance),2.50AU(3:1 resonance), 2.82AU(5:2 resonance), 2.95AU(7:3 resonance), and 3.27AU(2:1 resonance); while some weaker gaps appear at: 1.90AU(9:2 resonance), 2.25AU(7:2 resonance), 2.33AU(10:3 resonance), 2.71AU(8:3 resonance), 3.03AU(9:4 resonance), 3.075AU(11:5 resonance), 3.47AU(11:6 resonance), and 3.7 AU(5:3 resonance) as in figure 1

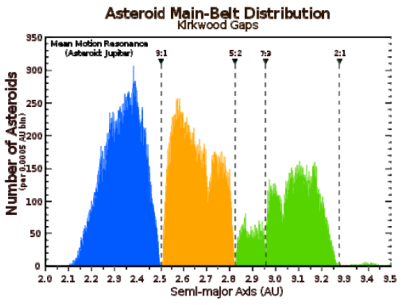

Fig. 1. Caption

Our goal is to simulate these observed results as many as possible, and we choose Runge-Kutta method and Simplectic method to conduct the simulation.

2. PHYSICAL DESCRIPTIONS.

A. Dimensionless Equations of Motion(1). This is an n-body problem involving Jupiter, asteroids, and the Sun. Newton's second law tells:

$$m_{ij}\frac{d^2\boldsymbol{r_i}}{dt^2}=\sum_{j\neq i}\frac{Gm_im_j(\boldsymbol{r}_j-\boldsymbol{r}_i)}{|rj-ri|^3} \tag{1}$$

But the mass of the sun is much larger than the others. So our first assumption is that the Sun is inert at the origin point. For Jupitor, the gravitational effect of the asteroid belt is ignorable compared with that of the Sun, so our second assumption reads the Jupitor is only influenced by the Sun, and its orbit is a closed ellipse. Based on these two assumptions, we can write down the equation of motion of Jupiter

$$M_J\frac{d^2\boldsymbol{r_J}}{dt^2}=\frac{M_sM_J(-\boldsymbol{r_J})}{|r_J|^3} \tag{2}$$

where M_s and M_J are the mass of the Sun and Jupiter, respectively; and $\boldsymbol{r_J}$is the displacement of the Jupiter.

Rewriting the equation in xy coordinates(note that we have postulated all possible motions are restrained in the xyplane, which is a reasonable requirement)

$$\frac{d^2x_J}{dt^2}=-\frac{GM_Sx_J}{((x_J)^2+(y_J)^2)^{\frac{3}{2}}} \tag{3}$$

$$\frac{d^2y_J}{dt^2}=-\frac{GM_Sy_J}{((x_J)^2+(y_J)^2)^{\frac{3}{2}}} \tag{4}$$

For asteroids, we ignore the effects of other asteroids. So the equation of motion of any one asteroid reads(this is quite

similar to the independent electrons assumption in the solid theory, where the interaction between electrons are ignored)

$$m_a \frac{d^2 \boldsymbol{r_a}}{dt^2} = \frac{GM_s m_a(-\boldsymbol{r_a})}{|r_a|^3} + \frac{GM_J m_a(-\boldsymbol{r_J} - \boldsymbol{r_a})}{|r_J - r_a|^3} \quad \textbf{(5)}$$

Again, write this in xycoordinates

$$\frac{d^2 x_a}{dt^2} = \frac{GM_s(-x_a)}{((x_a)^2 + (y_a)^2)^{\frac{3}{2}}} + \frac{GM_J(x_J - x_a)}{((x_J - x_a)^2 + (y_J - y_a)^2)^{\frac{3}{2}}} \quad \textbf{(6)}$$

$$\frac{d^2 y_a}{dt^2} = \frac{GM_s(-y_a)}{((x_a)^2 + (y_a)^2)^{\frac{3}{2}}} + \frac{GM_J(x_J - x_a)}{((y_J - y_a)^2 + (y_J - y_a)^2)^{\frac{3}{2}}} \quad \textbf{(7)}$$

To make these equations of motion more applicable in numerical simulation, it is vital to choose a proper unit system. To do this, we assume

$$m = m' m_0; x = x' d_0; y = y' d_0; u = u' u_0; t = t' t_0 \quad \textbf{(8)}$$

pluging these to the motion of Jupiter, we obtain

$$\frac{dx'_J}{dt'} = -\frac{t_0 u_0}{d_0} u'_{Jx} \quad \textbf{(9)}$$

$$\frac{du'_{Jx}}{dt'} = \frac{-Gm_0 t_0}{d_0^2 u_0} \frac{M'_S x'_J}{((x'_J)^2 + (y'_J)^2)^{\frac{3}{2}}} \quad \textbf{(10)}$$

Let

$$\frac{t_0 u_0}{d_0} = \frac{Gm_0 t_0}{d_0^2 u_0} = 1 \quad \textbf{(11)}$$

and notice that

$$t_0 = 1yr; d_0 = 1AU; \quad \textbf{(12)}$$

We get

$$G = \frac{d_0^3}{m_0 t_0^2} = \frac{4\pi^2 AU^3}{M_S yr^2} = \frac{4\pi^2 d_0^3}{M_S t_0^2} \quad \textbf{(13)}$$

$$M_s = 4\pi^2 m_0 \xrightarrow{M'_S = 4\pi^2} and m_0 = \frac{4\pi^2}{M_S} \quad \textbf{(14)}$$

Then the equations of motion become

$$\frac{dx'_J}{dt'} = u'_{Jx} \quad \textbf{(15)}$$

$$\frac{u'_{Jx}}{dt'} = -\frac{M'_S x'_J}{((x'_J)^2 + (y'_J)^2)^{\frac{3}{2}}} \quad \textbf{(16)}$$

$$\frac{dy'_J}{dt'} = u'_{Jy} \quad \textbf{(17)}$$

$$\frac{u'_{Jy}}{dt'} = -\frac{M'_S y'_J}{((x'_J)^2 + (y'_J)^2)^{\frac{3}{2}}} \quad \textbf{(18)}$$

Similarly, for the asteroid, we have

$$\frac{dx'_a}{dt'} = u'_{ax} \quad \textbf{(19)}$$

$$\frac{u'_{ax}}{dt'} = -\frac{M'_S x'_J}{((x'_J)^2 + (y'_J)^2)^{\frac{3}{2}}} + \frac{M'_J(x'_J - x'_a)}{((x'_J - x'_a)^2 + (y'_J - y'_a)^2)^{\frac{3}{2}}} \quad \textbf{(20)}$$

$$\frac{dy'_a}{dt'} = u'_{ay} \quad \textbf{(21)}$$

$$\frac{u'_{ay}}{dt'} = -\frac{M'_S y'_J}{((x'_J)^2 + (y'_J)^2)^{\frac{3}{2}}} + \frac{M'_J(y'_J - y'_a)}{((x'_J - x'_a)^2 + (y'_J - y'_a)^2)^{\frac{3}{2}}} \quad \textbf{(22)}$$

B. Initial Condition. Now to start the numerical simulation, we introduce the following dimensionless initial conditions. For Jupitor, the starting position is

$$x'_{Ji} = \frac{5.4588AU}{1AU} = 5.4588; y'_{Ji} = 0 \quad \textbf{(23)}$$

and the dimensionless mass reads

$$M'_J = \frac{M_J}{m_0} = \frac{\frac{M_S}{1047}}{\frac{M_S}{4\pi^2}} = 0.0376807 \quad \textbf{(24)}$$

and the velocity reads

$$v_{Ji} = \sqrt{\frac{GM_S}{x_{Ji}}(1-e)} = 2.62267 \quad \textbf{(25)}$$

For asteroids, the initial condition should locate in a range

$$\frac{2AU}{1AU} \leq r'_{ai} \leq \frac{3.5AU}{1AU}; 0 \leq \theta \leq 2\pi \quad \textbf{(26)}$$

and the corresponding velocity reads

$$v'_{axi} = -\sqrt{\frac{4\pi^2}{r'_{ai}}} sin\theta; v'_{ayi} = \sqrt{\frac{4\pi^2}{r'_{ai}}} cos\theta \quad \textbf{(27)}$$

3. ALGORITHM.

Now, how to solve this ODE efficiently and precisely and keep the conservation of the enerygy is the core problem in this project. The classical methods like Runge-Kutta method fail after long priod of times and may lead to wrong answers(Appendix 2). We overcome this problem by employing the higher order symplectic integrators(3)

For a general Hamilton problem, the time evolution of a dynamics variable is $\dot{z} = \{z, H\}$ or using differential operator $\dot{z} = D_H z$. The general solution is:

$$z(\tau) = \exp(\tau D_H) z(0) \quad \textbf{(28)}$$

For a Hamiltonian of $H(p,q) = T(q) + V(p)$, the above equation can be simplified to

$$z(\tau) = \exp(\tau(D_T + D_V)) z(0) \quad \textbf{(29)}$$

The numerical method using successive mapping to approch the true solutionEq. (3):

$$z(\tau) = \left\{ \prod_{i}^{k} \exp(\tau c_i D_T) \exp(\tau d_i D_V) + o(\tau^{n+1}) \right\} z(0) \tag{30}$$

Expand Eq. (3) we have

$$\begin{cases} q_i = q_{i-1} + c_i \tau \dfrac{\partial T}{\partial p}(p_{i-1}) \\ p_i = p_{i-1} - d_i \tau \dfrac{\partial V}{\partial q}(q_i) \end{cases} \tag{31}$$

Yoshida gave a sets of (c_i, d_i) to reach the order 6th and 8th. And Spaletta et. al.(2) further constructed method of 10th-order symplectic integrators. However, we need at least 32 steps to reach 10th-order precision while 16 steps are requested by 8th-method. We choose Yoshiba's method A(3) after consideration. Also, a benchmark of this method is also attached in Appendix B.

d:1-4	1.042426	1.820206	0.15774	2.440027
d:5-8	-0.00717	-2.44699	-1.61582	-1.78083
d:9-12	-1.61582	-2.44699	-0.00717	2.440027
d:13-16	0.15774	1.820206	1.042426	0
c:1-4	0.521213	1.431316	0.988973	1.298884
c:5-8	1.216429	-1.22708	-2.03141	-1.69833
c:9-12	-1.69833	-2.03141	-1.22708	1.216429
c:13-16	1.298884	0.988973	1.431316	0.521213

Table 1. C and D table

We implement this alogrithm in C++. You can access to the full program from Github https://github.com/joeyli99/Project3_The-Kirkwood-gaps. To reach the better speed, we recommand you enable the "Release" methods in IDEs or using -O3 compiler option in gcc or clang/LLVM. In our test, under the conditions of 2000 years and 1/16 steps, the program takes 0.08s on average to solve ODE. The document may give you further guidence of parallel computation.

4. RESULT.

We arranged 100000 asteriods along the scale between 2 AU to 3.5AU, and simulate their motion with both fixed initial velocity and position(say $\theta = 0$)as well as random initial condition(say $\theta = rand[0, 2\pi]$). The fixed initial condition has the result shown in 3

It can be shown in 3 that there are two most significant gap at 2.5AU and 3.27AU, which is quite consistent with the experiment observation. And there are distinguishable depletions in 2.06AU, 2.82AU, and 2.95AU, but these depletions are not as obious as the former two because we cannot distinguish them from many other numerical depletions(such as 3.18AU). The way we distinguish these non-special depletions is by comparision with the existed observation results, which we've illustrated in the introduction part. Further more, we can even see some tiny depletion at 2.33AU,

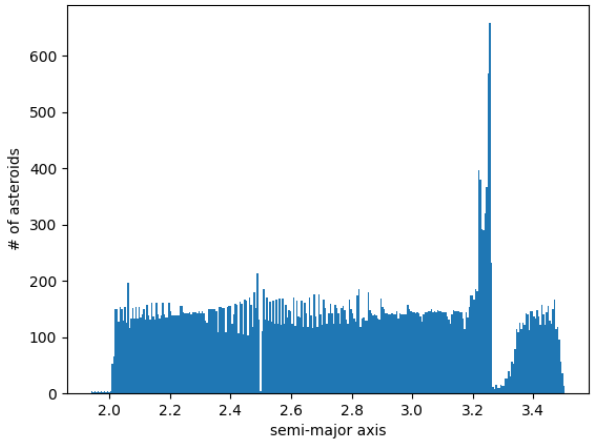

Fig. 2. Distribution of asteroids refering to their semi major with fixed inital angular $\theta = 0$

2.71AU, 3.075AU along the semi major axis, which shows the possibility of out algorithm to simulate the weaker gaps.

The result is m δa over a figure

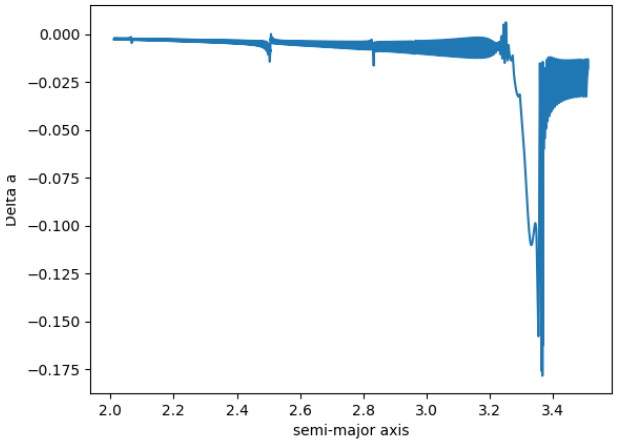

Fig. 3. The final semi-major axises as a function of initial a

To make the result more convincing, we also simulate the motion of these asteroids with random intitial angulat(say θ), and the result is shown at 4

However, this result is not as intuitive as the former one, whose initial angular is set to zero. As shown in 4,we can only safely say that there are gaps at 2.5AU and 3.27AU, the other gaps are hard to distinguish, even these two gaps still hold a number of asteroids. Despite this, this result enhance our conclusion that there must be depletion of asteroids at 2.5AU as well as 3.27AU.

To make this conclusion more apparent, we draw the variation of asteroids' semi major as a function of time(with a fixed initial angular $\theta = 0$). This result is shown in 5.

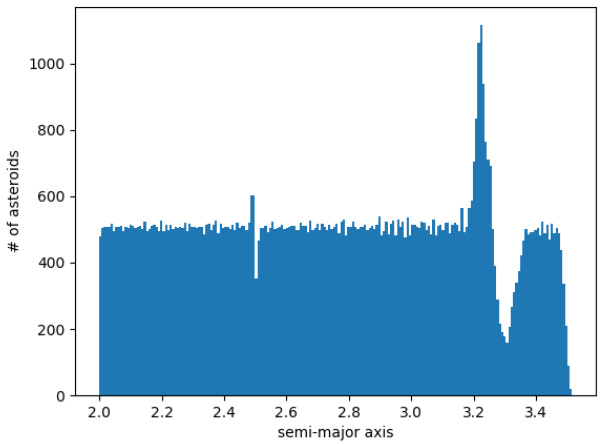

Fig. 4. Distribution of asteroids refering to their semi major with random intial angular

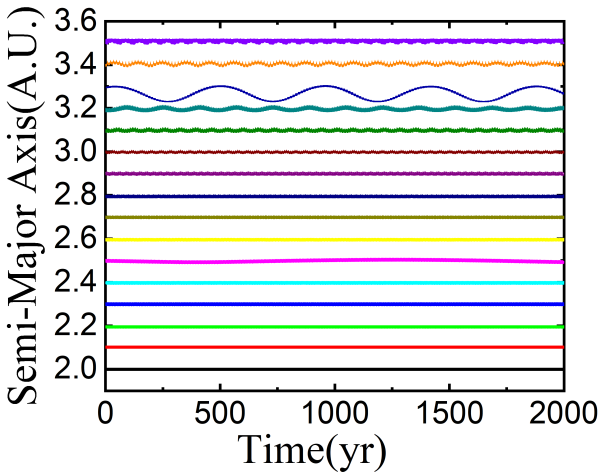

Fig. 5. Variation of asteroids' semi major as a function of time

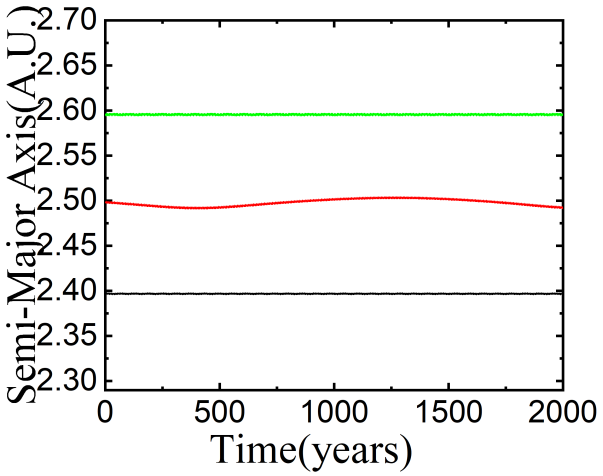

Fig. 6. Variation of asteroids' semi major as a function of time focused on the adjacent area of 2.5 AU

As shown in 5 and 6, we can see obvious oscillations at initial semi majors of 3.27AU, and a long period oscillation(corresponding to a low frequency) at 2.5AU. This phenomenon not only asserts that there do exists orbital resonance at these two semi majors, but also shows the strength of resonance at 3.27AU is far more intense than that at 2.5AU(In fact, the resonance strength is the most intense at 3.27AU). This futther validates our simulation results.

5. CONCLUSION.

In this paper, we successful explain the Kirkwood gaps using the numerical simulation, and we implement the high order symplectic ode method. And here is our conclusions:

1. We implement the high-order symplectic method which can be easily reused in other question.
2. We use the simple parallel computation to improve the speed and get enough data to support our result.
3. We suggest that there are two most significant gap at 2.5AU and 3.27AU.
4. The gaps originate from the unstability of orbits near the 2.5 AU and 3.27AU.
5. Simple ode method like Runge-Kutta may failed because they break the stability of orbits.

However, our result can also be improved from many aspects.

It's a pity that we don't give out the mathematical explanation that why the unstability occurs and is there chaos behavior near the 3.27 AU. We believe that Poincare section may help us but due to the limitation of time we can not finish it. And another work can be done is to improve the algorithm form two aspect: first, we using the final energy to figure out the semi-major axis but the energy will be effected by Jupiter. Second, the time span looks too narrow because the large time span is both time and memory consuming we believe that we can improve it.

Also, in Yuanye and Zheyu's project, they suggest that obliquity of the ecliptic may play the roll which is neglect by others. We can also make it.

ACKNOWLEDGEMENTS

We would like to express our thanks to the people who have helped us most throughout our project. We are grateful to our teacher Weidong Luo for support for the project. A special thank of ours goes to our classmates Yuanye Lin and Zheyu Wu who helped us out in completing the project, where they all exchanged their advanced results.

Reference

1. Shuaizhou Wang Shuai Liu, Zikun Lin. Kirkwood gaps numerical simulation. http://www.u.arizona.edu/~dpsaltis/Phys305/liuetal.pdf, 2017.
2. Mark Sofroniou and Giulia Spaletta. Derivation of symmetric composition constants for symmetric integrators. *Optimization Methods and Software*, 20:597–613, 08 2005.
3. Haruo Yoshida. Construction of higher order symplectic integrators. *Physics Letters A*, 150(5):262 – 268, 1990.

A The C++ implement

Hamilton.h

```cpp
#pragma once

#include<cmath>
#include<vector>
#include <array>
typedef std::array<double, 4> PhaseVector;
class Hamilton {
public:
    Hamilton(const PhaseVector &set_q0,
    const PhaseVector &set_p0,
    double set_start,
    double set_end) {
        q0 = set_q0;
        p0 = set_p0;
        start = set_start;
        end = set_end;
        t.push_back(set_start);
        p.push_back(p0);
        q.push_back(q0);
    }

    ~Hamilton() = default;

    std::vector<PhaseVector > p;
    std::vector<PhaseVector > q;
    std::vector<double> t;

    void solve(double step);

private:
    const int dim=4;
    PhaseVector q0{};
    PhaseVector p0{};
    double start;
    double end;

    PhaseVector dp(const PhaseVector &qarray);

    PhaseVector dq(const PhaseVector &parray);
};
```

Hamilton.cpp

```cpp
#include<iostream>
#include "Hamilton.h"

constexpr double PI = 3.14159265358979323846, Ms = 4 * PI * PI;
constexpr double Mj = Ms / 1047.56;

void Hamilton::solve(double step) {
    t.reserve(100000);
    p.reserve(100000);
    q.reserve(100000);
    //solve the Hamilton system by Yoshida 8-order.
    double d[16] = {
            1.04242620869991,
```

```
            1.82020630970714,
            0.157739928123617,
            2.44002732616735,
            -0.0071698941970812,
            -2.44699182370524,
            -1.61582374150097,
            -1.7808286265894515,
            -1.61582374150097,
            -2.44699182370524,
            -0.0071698941970812,
            2.44002732616735,
            0.157739928123617,
            1.82020630970714,
            1.04242620869991,
            0.0
    };
    double c[16] = {
            0.521213104349955,
            1.4313162592035251,
            0.9889731189153784,
            1.2988836271454836,
            1.2164287159851346,
            -1.2270808589511606,
            -2.031407782603105,
            -1.6983261840452109,
            -1.6983261840452109,
            -2.031407782603105,
            -1.2270808589511606,
            1.2164287159851346,
            1.2988836271454836,
            0.9889731189153784,
            1.4313162592035251,
            0.521213104349955
    };

    //Youshiba 8
    //#pragma omp parallel for
    for (double time = start + step; time < end; time += step) {
        std::array<double, 4> q_now = q.back();
        std::array<double, 4> p_now = p.back();

        for (int i = 0; i < 16; i++) {
            PhaseVector dq_now = dq(p_now);
            double c_step = c[i];
            double d_step = d[i];
            for (int j = 0; j < dim; j++) {
                q_now[j] += c_step * step * dq_now[j];
            }
            PhaseVector dp_now = dp(q_now);
            for (int j = 0; j < dim; j++) {
                p_now[j] -= d_step * step * dp_now[j];
            }
        }
        p.push_back(p_now);
        q.push_back(q_now);
        t.push_back(time);
    }
```

```
    //std::cout << "Solve END" << std::endl;
}

PhaseVector Hamilton::dp(const PhaseVector &qarray) {
    double xj = qarray[0],
           yj = qarray[1],
           xa = qarray[2],
           ya = qarray[3],
           rj = sqrt(xj * xj + yj * yj),
           ra = sqrt(xa * xa + ya * ya),
           daj = sqrt((xa - xj) * (xa - xj) + (ya - yj) * (ya - yj));
    PhaseVector Hq = {Ms * xj / pow(rj, 3), Ms * yj / pow(rj, 3),
              Ms * xa / pow(ra, 3) + Mj * (xa - xj) / pow(daj, 3),
              Ms * ya / pow(ra, 3) + Mj * (ya - yj) / pow(daj, 3)};
    return Hq;
}

PhaseVector Hamilton::dq(const PhaseVector &parray) {
    return parray;
}
```

```
#include<fstream>
#include<iostream>
#include<random>
#include<omp.h>
#include<fmt\time.h>
#include"Hamilton.h"

constexpr auto PI = 3.14159265358979323846, Ms = 4 * PI * PI, Mj = Ms / 1047.56;
const auto n = 50000;
const auto years = 2000;
using namespace std;

int main() {

    //the benchmark
    clock_t begin, finish;
    begin = clock();

    //get the system time, and seed to the random number generator.
    time_t systime = time(nullptr);

    //std::random_device rd; Fuck gcc.
    auto seed = systime;
    std::mt19937_64 e(seed); //seed the random generator by system time.
    std::uniform_real_distribution<double> u(0, 2 * PI); //uniform distribution phi.

    //using the time to name the data
    string input_file, output_file;
    input_file = fmt::format("IV{:%Y-%m-%d-%H-%M}-[n={},year={}].txt",
    *std::localtime(&systime), n, years);
    output_file = fmt::format("DATA{:%Y-%m-%d-%H-%M}-[n={},year={}].txt",
    *std::localtime(&systime), n, years);
    ofstream outinv(input_file);
    ofstream outdata(output_file);

    // Calculation Begin!
    std::array<double, n> rs = {};
```

```
    std::array<double, n> phis = {};
    std::array<double, n> as = {};
    //std::vector<double > as;

    for (int i = 0; i < n; i++) {
        phis[i] = u(e);
        //rs[i] = 2.5;
        rs[i] = 2 + ((double) i) / (n - 1) * 1.5;
        outinv << rs[i] << " " << phis[i] << endl;
    }
    cout << "Output Initial Value Finished!" << endl;

#pragma omp parallel for
    for (int i = 0; i < n; i++) {
        double r, phi;
        r = rs[i];
        phi = phis[i];
        PhaseVector q0 = {5.458104, 0, r * cos(phi), r * sin(phi)};
        PhaseVector p0 = {0, sqrt(Ms * (1 - 0.048912) / 5.458104),
                  -sqrt(4 * PI * PI / r) * sin(phi),
                   sqrt(4 * PI * PI / r) * cos(phi)};
        Hamilton H(q0, p0, 0, years);
        H.solve(1.0 / 16.0);

        //std::vector<double> vec;
        //vec.reserve(32000);
        // for (int i=0; i<32000; i++)
        // {
        //      double xj = (H.q)[i][0];
        //      double yj = (H.q)[i][1];
        //      double xa = (H.q)[i][2];
        //      double ya = (H.q)[i][3];
        //      double vxa = (H.p)[i][2];
        //      double vya = (H.p)[i][3];
        //      auto E= 0.5 * (vxa * vxa + vya * vya) - Ms / sqrt(xa * xa + ya * ya) -
        //                  Mj / sqrt((xa - xj) * (xa - xj) + (ya - yj) * (ya - yj));
        //      as.push_back(-2 * PI * PI / E);
        // }
        //as.push_back(vec);
        double xj = (H.q).back()[0];
        double yj = (H.q).back()[1];
        double xa = (H.q).back()[2];
        double ya = (H.q).back()[3];
        double vxa = (H.p).back()[2];
        double vya = (H.p).back()[3];
        double E = 0.5 * (vxa * vxa + vya * vya) - Ms / sqrt(xa * xa + ya * ya) -
                   Mj / sqrt((xa - xj) * (xa - xj) + (ya - yj) * (ya - yj));
        as[i] = -2 * PI * PI / E;
        // if (i % 100 == 0) {
        //     cout << i << " ";
        // }
    }
    finish = clock();
    // for (auto &&x : as)
    // {
    //      outdata<<x<<endl;
    // }
```

```
95
96      for (int i=0; i<n; i++)
97      {
98          outdata<< as[i] <<endl;
99      }
100     cout << "Output Result Finished! Total time: " <<
101     ((double) finish - begin) / CLOCKS_PER_SEC
102     << endl;
103     return 0;
104 }
```

B The Runge-Kuuta failed

We using the simple ode method and it failed

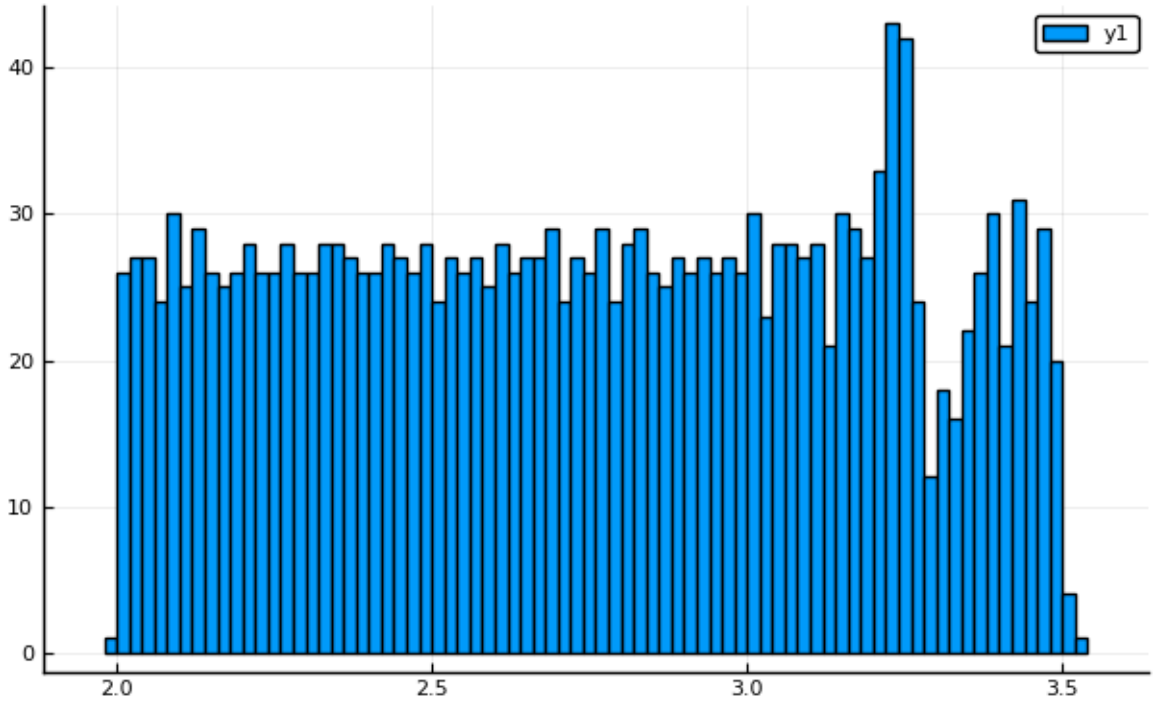

Fig. S1. Failed