Modeling Planetary Orbits

Orion Forowycz, Max Paik, Valeriia Rohoza, Jason Phelan
Final Project Presentation
Physics 352 - Team 5 - Winter 2021

Our System

Single Planet:

$$F_{G,1} = -\frac{GM_s m_1}{r_1^2}\mathbf{\hat{r}_1}$$

Two Planets:

$$F_{G,1} = -\frac{GM_s m_1}{r_1^2}\mathbf{\hat{r}_1} + \frac{Gm_1 m_2}{r_{1,2}^2}\mathbf{\hat{r}_{1,2}}$$

N Planets:

$$F_{G,i} = -\frac{GM_s m_i}{r_i^2}\mathbf{\hat{r}_i} + \sum_{j \neq i} \frac{Gm_i m_j}{r_{i,j}^2}\mathbf{\hat{r}_{i,j}}$$

For the simulations, we assumed all planets lie in a plane

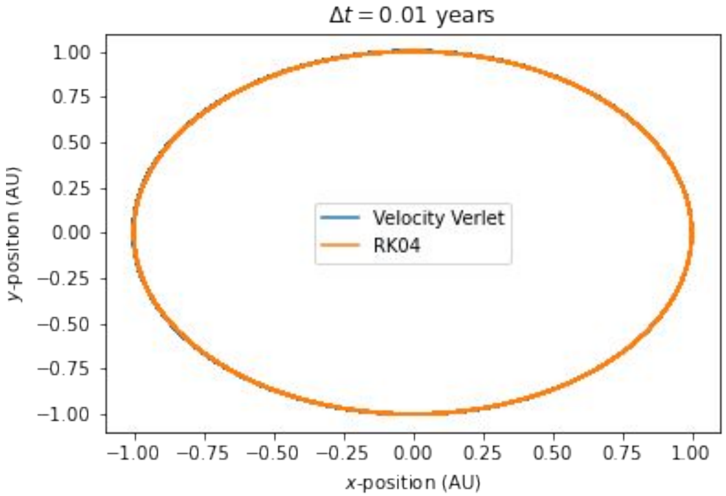
Symplectic Integrators

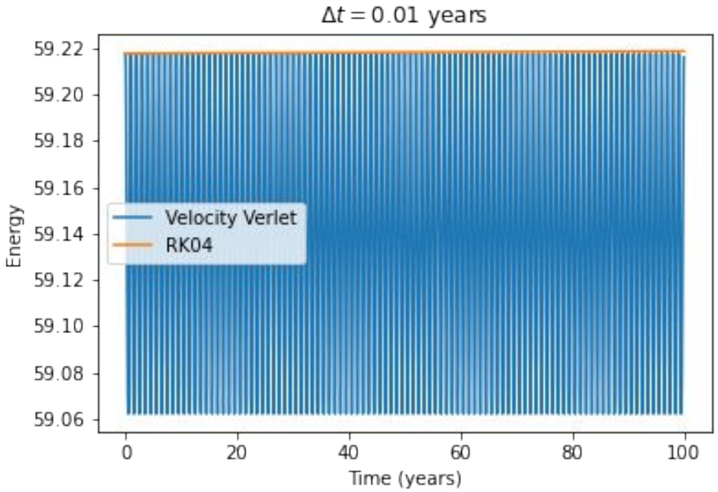
- Expect that instabilities will manifest over many thousands of years
  - Require robust solver
- Symplectic integrator conserves the Hamiltonian as well as the volume in phase space
  - For our system, conserve energy as well

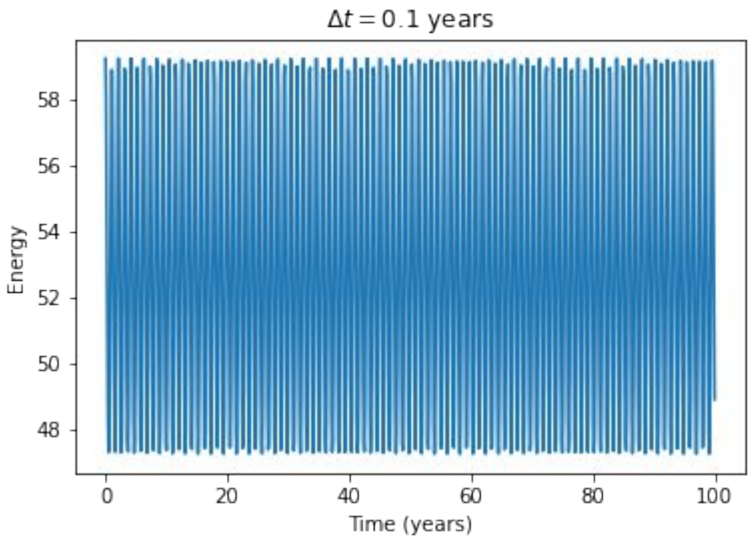
Velocity Verlet Method

- Second order symplectic integrator

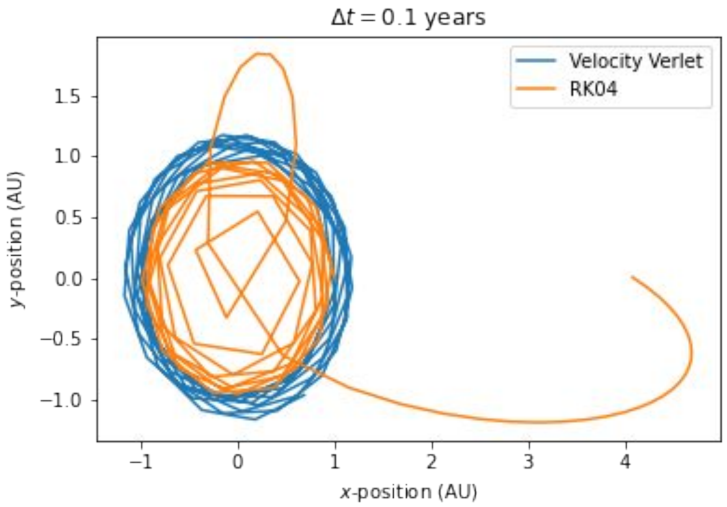
$$a_n = F(x_n)$$
$$v_{n+1/2} = v_{n-1/2} + (a_n)\Delta t$$
$$x_{n+1} = x_n + (v_{n+1/2})\Delta t$$
$$v_{n+1} = v_{n+1/2} + \frac{1}{2}(a_n)\Delta t$$


$\Delta t = 0.01$ years
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
y-position (AU)
Velocity Verlet
RK04
−1.00 −0.75 −0.50 −0.25 0.00 0.25 0.50 0.75 1.00
x-position (AU)


$\Delta t = 0.01$ years
59.22
59.20
59.18
59.16
59.14
59.12
59.10
59.08
59.06
Energy
Velocity Verlet
RK04
0 20 40 60 80 100
Time (years)



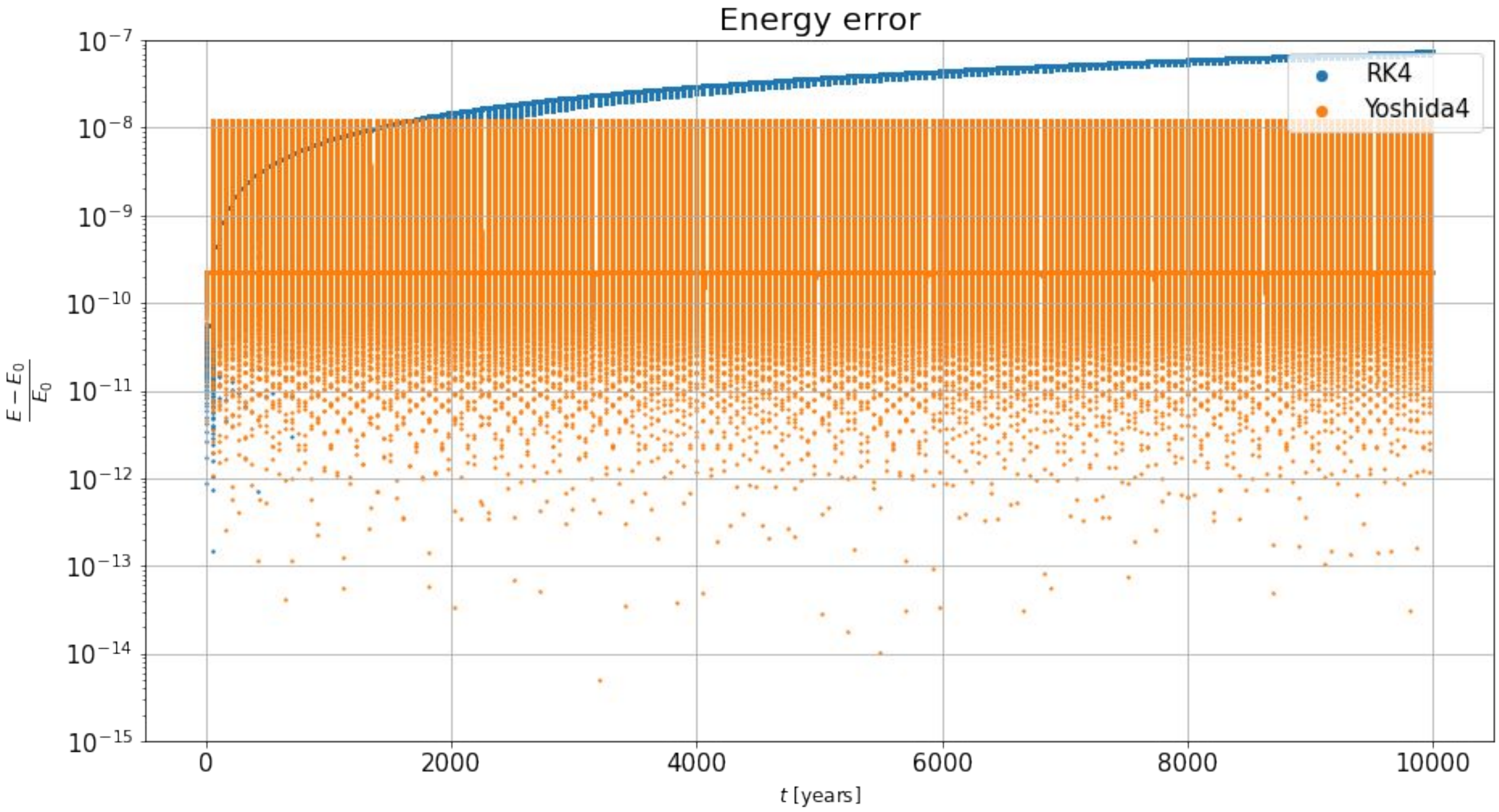
$\Delta t = 0.1$ years
Energy
58
56
54
52
50
48
0
20
40
60
80
100
Time (years)



$\Delta t = 0.1$ years
Velocity Verlet
RK04
1.5
1.0
0.5
0.0
−0.5
−1.0
y-position (AU)
−1
0
1
2
3
4
x-position (AU)

Leapfrog integration
4th order Yoshida integrator

- Time-reversible
- Conserves angular momentum
- Preserves area in phase space

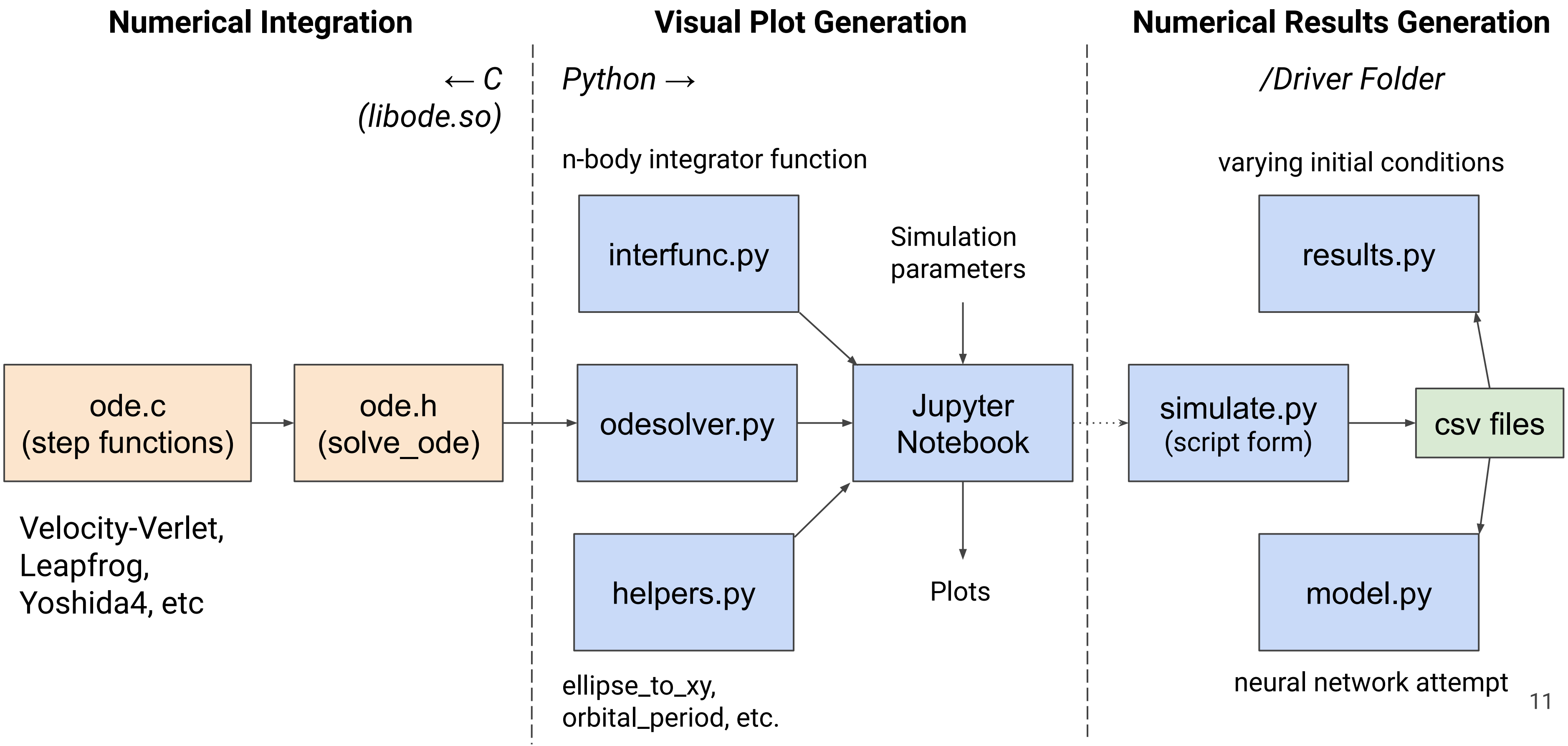


Energy error
RK4
Yoshida4
$\frac{E-E_0}{E_0}$
10^{-7}
10^{-8}
10^{-9}
10^{-10}
10^{-11}
10^{-12}
10^{-13}
10^{-14}
10^{-15}
0
2000
4000
6000
8000
10000
t [years]

```c
void stepYoshida4(double dt, double t, double a[], double anew[], int nvar,
    double params[], double* buf,
    void (*dxdt)(double t, double a[], double params[], double derivs[])) {
    int i, k;

    /* f stores the values of dVdt */
    /* temp_xV stores temporary values for x,V */
    /* coeff_xV includes 4 values for x and 3 values for V, order in x,V */

    double* f = buf, * temp_xV = f + nvar, * coeff_xV = temp_xV + nvar;

    /* set coefficients */
    const double num0 = -1.7024143839193153215916254;
    const double num1 = 1.3512071919596577718181152;

    /* c1, c2, c3, c4 coefficients for x */
    coeff_xV[0] = 0.5 * num1;
    coeff_xV[2] = 0.5 * (num0 + num1);
    coeff_xV[4] = coeff_xV[2];
    coeff_xV[6] = coeff_xV[0];

    /* d1, d2, d3 coefficients for V */
    coeff_xV[1] = num1;
    coeff_xV[3] = num0;
    coeff_xV[5] = coeff_xV[1];
```

```c
    for (i = 0; i < nvar; i++) {
        temp_xV[i] = a[i];
    }

    /* take three Euler steps with different dt
    assume that dxdt = V and dVdt is defined by the given function */

    for (k = 0; k < 3; k++) {
        for (i = 0; i < (nvar / 2); i++) {
            temp_xV[2 * i] = temp_xV[2 * i] + coeff_xV[2 * k] * temp_xV[2 * i + 1] * dt;
        }

        (*dxdt)(t + coeff_xV[2 * k] * dt, temp_xV, params, f);

        for (i = 0; i < (nvar / 2); i++) {
            temp_xV[2 * i + 1] = temp_xV[2 * i + 1] + coeff_xV[2 * k + 1] * f[2 * i + 1] * dt;
        }
    }

    /* compute new values */
    for (i = 0; i < (nvar / 2); i++) {
        anew[2 * i] = temp_xV[2 * i] + coeff_xV[6] * temp_xV[2 * i + 1] * dt;
        anew[2 * i + 1] = temp_xV[2 * i + 1];
    }
}
```

Simulation Pipeline



N-Body Integrator Function

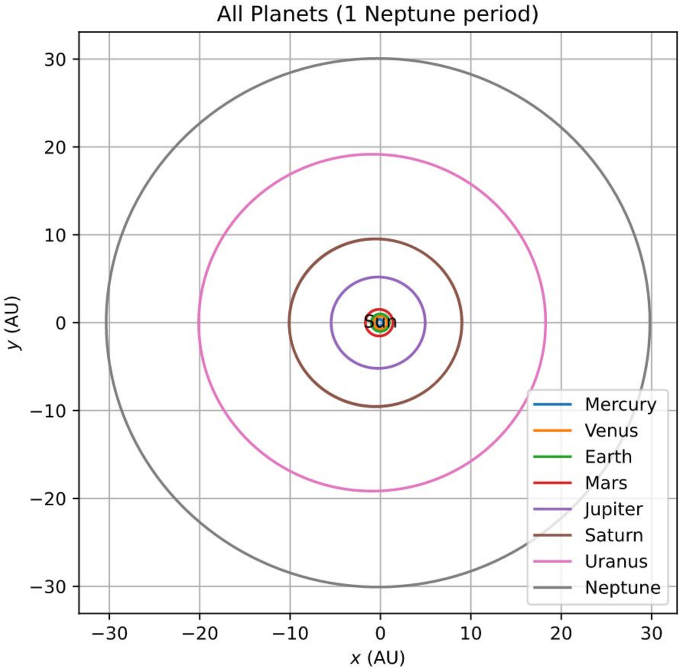
Uses a frame of reference around a stationary central body (such as the sun).

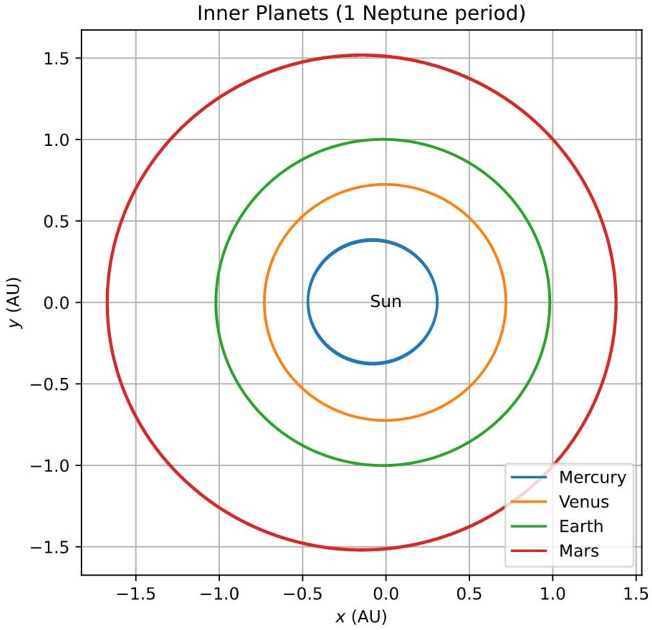
Iterates through each orbital body given and sums the gravitational acceleration from every other body to get the time derivatives for x and y velocities needed for each step.

```python
GM_S = params[0]
n = params[1]

for i in range(n):
    #Object input values
    x = sol[ind_x(i)]
    v_x = sol[ind_v_x(i)]
    y = sol[ind_y(i)]
    v_y = sol[ind_v_y(i)]
    r = np.sqrt(x**2 + y**2)

    #Initial object output values
    dydt[ind_x(i)] = v_x #dx/dt
    dydt[ind_v_x(i)] = -GM_S/(r**3)*x #dv_x/dt
    dydt[ind_y(i)] = v_y #dy/dt
    dydt[ind_v_y(i)] = -GM_S/(r**3)*y #dv_y/dt

    #Adding on forces from other orbiting objects
    for j in range(n):
        if (j == i): continue
        GM_j = params[j+2] # Skipping GM_S and n in params
        x_j = sol[ind_x(j)]
        y_j = sol[ind_y(j)]
        x_diff = x - x_j
        y_diff = y - y_j
        r_diff = np.sqrt(x_diff**2 + y_diff**2)
        dydt[ind_v_x(i)] = dydt[ind_v_x(i)] - GM_j/(r_diff**3)*x_diff #dv_x/dt
        dydt[ind_v_y(i)] = dydt[ind_v_y(i)] - GM_j/(r_diff**3)*y_diff #dv_y/dt
```

Solar System Simulation

All planets initially set to their average orbital radius and eccentricity, with $\theta = \theta_E = 0$

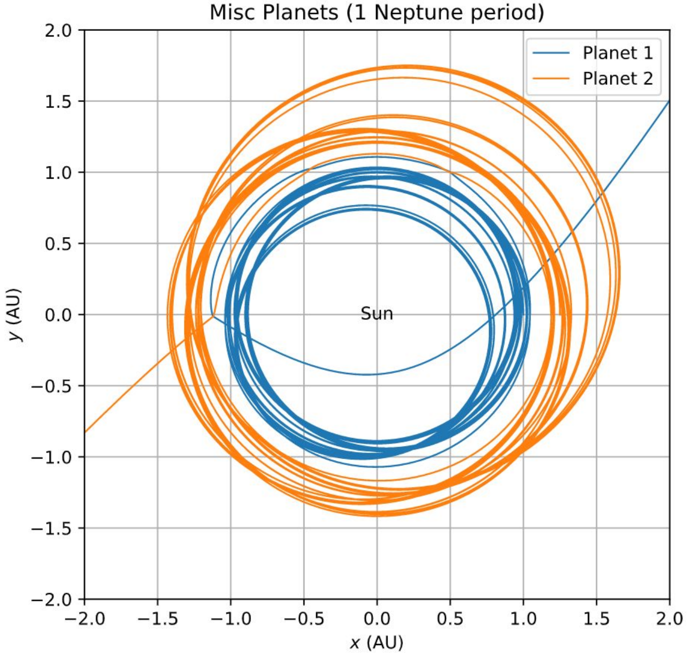


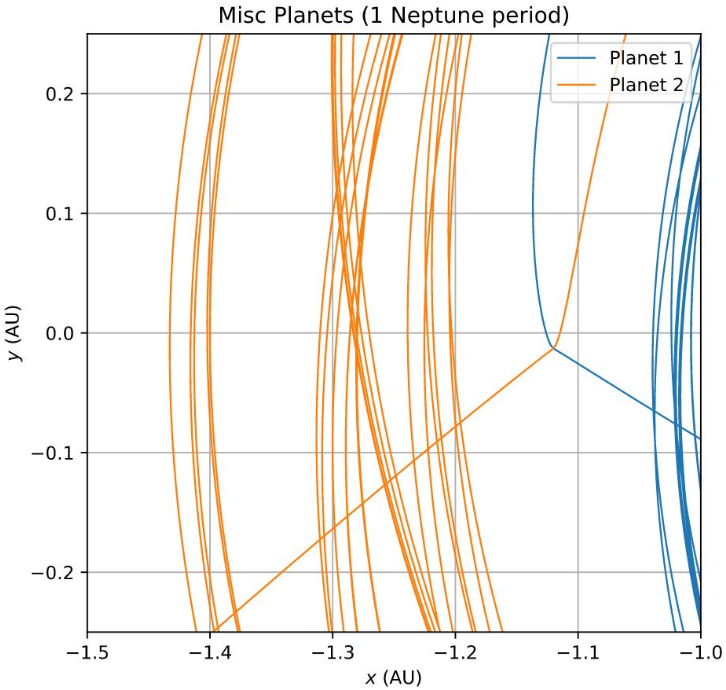


Unstable Configuration Example

Two Jupiter-mass planets orbiting the Sun.

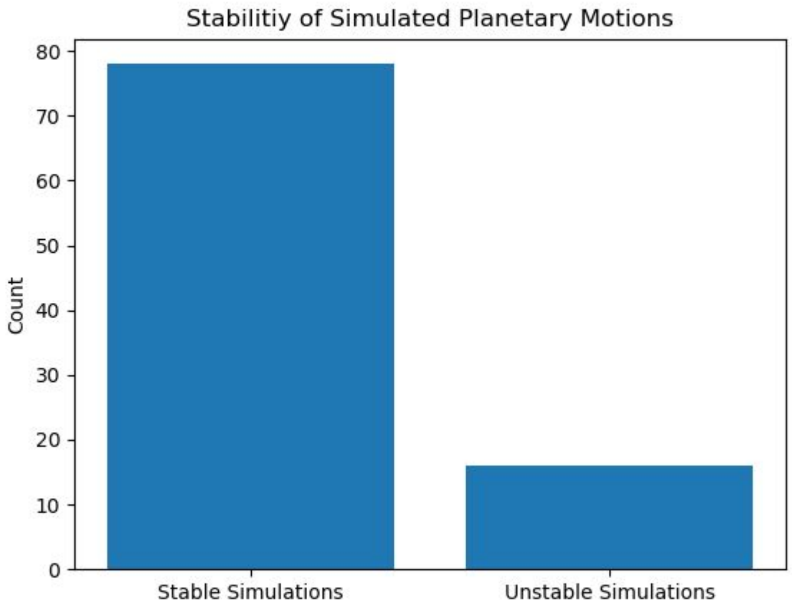
They start at 1 AU and 1.25 AU respectively, but their orbits become unstable within a Neptune period timescale





Results

- We then ran our simulation a hundred times with various initial conditions
- We measured the stability of the system simply as whether or not Mercury was ejected from its orbit
- Most simulations were stable over 1000 Neptune Periods
- We added small (or large in the case of Mercury) perturbations into every planet’s initial conditions



```python
# add a bit of variation to all planetary initial conditions
init_Mer = np.array(ellipse_to_xy(random.gauss(0.3870993, .0001), random.gauss(0.20564, .0001), 0., 0.))
init_Ven = np.array(ellipse_to_xy(random.gauss(0.723336, .0001), random.gauss(0.00678, .0001), 0., 0.))
init_Ven[ind_v_y(0)] *= -1
init_Ear = np.array(ellipse_to_xy(random.gauss(1.000003, .0001), random.gauss(0.01671, .0001), 0., 0.))
init_Mar = np.array(ellipse_to_xy(random.gauss(1.52371, .0001), random.gauss(0.09339, .0001), 0., 0.))
init_Jup = np.array(ellipse_to_xy(random.gauss(5.2029, .0001), random.gauss(0.0484, .0001), 0., 0.))
init_Sat = np.array(ellipse_to_xy(random.gauss(9.537, .0001), random.gauss(0.0539, .0001), 0., 0.))
init_Ura = np.array(ellipse_to_xy(random.gauss(19.189, .0001), random.gauss(0.04726, .0001), 0., 0.))
init_Nep = np.array(ellipse_to_xy(random.gauss(30.0699, .0001), random.gauss(0.00859, .0001), 0., 0.))
n_planets = 8

# add more significant noise to Mercury's initial conditions
init_Mer = list(map(lambda x : x + random.gauss(0, .05), init_Mer))
params = [GM_Sun, n_planets, GM_Mer, GM_Ven, GM_Ear, GM_Mar, GM_Jup, GM_Sat, GM_Ura, GM_Nep]
noisy_planets = np.concatenate((init_Mer,init_Ven,init_Ear,init_Mar,init_Jup,init_Sat,init_Ura,init_Nep))

a_Nep = 30.0699
total_time = 1000 * orbital_period(a_Nep,GM_Sun) # 1 Neptune period
step_size = orbital_period(a_0,GM_S)/100 # 1/100 of Mercury period
n_steps = int(total_time/step_size)

# run simulations
t,sol_untransposed = solve_ode(func_n_body,[0.,total_time], n_steps, noisy_planets, args=params, method="Yoshida4")
sol = sol_untransposed.T
```

Extra Time - Attempt at a Neural Network

- With our simulator already built, we also attempted to build a neural network to predict the stability of Mercury's orbit.
- After one round of training, the network improved, but afterwards it stagnated.
- The relatively small amount of simulated data likely made training this network difficult.

```
In [203]: run -i model.py
Epoch 1
-------------------------------
loss: 1.748609  [    0/   37]
loss: 198.767395  [   32/   37]
loss: 0.679743  [   64/   37]
loss: 0.000000  [   96/   37]
loss: 0.000149  [  128/   37]
loss: 4.029831  [  160/   37]
loss: 1.414407  [  192/   37]
loss: 1.180764  [  224/   37]
loss: 0.844853  [  256/   37]
loss: 0.938981  [  288/   37]
loss: 0.077444  [  320/   37]
loss: 0.000075  [  352/   37]
loss: 0.000000  [  384/   37]
loss: 0.000000  [  416/   37]
loss: 48.977917  [  448/   37]
loss: 0.000000  [  480/   37]
Test Error:
 Accuracy: 43.2%, Avg loss: 0.723522

Epoch 2
-------------------------------
loss: 0.557691  [    0/   37]
loss: 0.831051  [   32/   37]
loss: 0.596296  [   64/   37]
loss: 0.784473  [   96/   37]
loss: 0.634041  [  128/   37]
loss: 0.743524  [  160/   37]
loss: 0.668943  [  192/   37]
loss: 0.677306  [  224/   37]
loss: 0.671439  [  256/   37]
loss: 0.654497  [  288/   37]
loss: 0.630331  [  320/   37]
loss: 0.602534  [  352/   37]
loss: 0.573933  [  384/   37]
loss: 0.865175  [  416/   37]
loss: 0.886776  [  448/   37]
loss: 0.895769  [  480/   37]
Test Error:
 Accuracy: 56.8%, Avg loss: 0.684852
```

```
Epoch 3
-------------------------------
loss: 0.530913  [    0/   37]
loss: 0.904699  [   32/   37]
loss: 0.914523  [   64/   37]
loss: 0.916996  [   96/   37]
loss: 0.912819  [  128/   37]
loss: 0.519964  [  160/   37]
loss: 0.523229  [  192/   37]
loss: 0.523216  [  224/   37]
loss: 0.902341  [  256/   37]
loss: 0.899933  [  288/   37]
loss: 0.527706  [  320/   37]
loss: 0.529971  [  352/   37]
loss: 0.889671  [  384/   37]
loss: 0.532439  [  416/   37]
loss: 0.532501  [  448/   37]
loss: 0.529481  [  480/   37]
Test Error:
 Accuracy: 56.8%, Avg loss: 0.684726
```

```python
# define neural network class
class StabilityNetwork(nn.Module):
    def __init__(self):
        super(StabilityNetwork, self).__init__()

        def init_weights(m):
            if type(m) == nn.Linear:
                torch.nn.init.xavier_uniform_(m.weight)
                m.bias.data.fill_(0.01)

        # choose shallow network for relatively simple task
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(8*4, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 1)
        )
        self.linear_relu_stack = self.linear_relu_stack.apply(init_weights)

    def forward(self, x):
        logits = self.linear_relu_stack(x)
        return logits
```

```python
# function for training the model on a given set of data
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, xy in enumerate(dataloader):
        for i in range((len(xy['data']))):
            # Compute prediction and loss
            pred = model(xy['data'][i])
            loss = loss_fn(pred, xy['stable'][i])

            # Backpropagation
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            if batch % 100 == 0:
                loss, current = loss.item(), i * len(xy['data'][i])
                print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

Follow Ups

- In the future, it would certainly be interesting to take a more robust look at precisely what changes to planetary initial conditions cause Mercury to be ejected from its orbit
- A larger collection of sample data may also allow for a more effective neural network to be trained