Modelling planetary systems with iterative methods

Aiden Lewis

Our goal

- To evaluate the strengths and weaknesses of different iterative methods
  - Euler’s method
  - Improved Euler’s method
  - Heun’s method
  - Runge-Kutta (3rd, 4th order)
  - Adams-Bashforth (2nd, 3rd order)
- Computation time, accuracy

Theory: gravity

- Newton's law of universal gravitation:
    - $F_g = GMm/r^2$
- Kinetic energy:
    - $T = ½mv^2$
- Potential energy:
    - $U = GMm/r$

Theory: motion

- Newton's second law:
  - $F = ma$
- EOM:
  - $x'' = a \rightarrow x' = v, v' = a$
- We thus have a system of 2 first-order ODEs

Implementation

- Coded in Python
  - Chosen for its object-oriented nature & ease of use
  - Packaging a planet's attributes into a single 'Planet' object helps avoid needlessly convoluted expressions
  - For instance, to find the force Mars exerts on Earth: `earth.Fg(mars)`

```python
def Fg(self, m2: 'Planet') -> np.ndarray[float]:
    '''
    Returns gravitational force vector between two planets.

    ### Parameters
    m2: The other planet in question
    '''
    dvec = m2.pos - self.pos
    rv = np.zeros_like(self.pos)
    if self == m2: return rv
    if interactionsAllowed or (not self.is_star and m2.is_star):
        rv = G*self.mass*m2.mass/(self.dist(m2)**2)*(dvec/np.linalg.norm(dvec))
    return rv
```

Iterative methods

Euler's method: $y_{n+1} = y_n + \Delta t y_n'$

```
def euler(plns: list[pln.Planet], dt: float, past=None):
    '''...
    ps = np.array([])
    for p in plns:
        p.pos += dt*p.vel
        p.vel += dt*p.acc(plns)
        ps = np.append(ps, p.rebuild())
    return ps
```

Improved Euler's method:

$y_{n+1} = y_n + 0.5(k_1 + k_2)$

```
def eulerImp(plns: list[pln.Planet], dt: float, past=None):
    '''...
    ps = np.array([])
    for p in plns:
        k1v = p.acc(plns)*dt
        p1 = p.rebuild()
        p1.vel += k1v
        k1r = p.vel*dt
        p1.pos += k1r
        k2v = p1.acc(plns)*dt
        p.vel += 0.5*(k1v + k2v)
        k2r = p1.vel*dt
        p.pos += 0.5*(k1r + k2r)
        ps = np.append(ps, p.rebuild())
    return ps
```

Iterative methods

4th-order Runge-Kutta:

$$y_{n+1} = y_n + (k_1 + 2k_2 + 2k_3 + k_4)/6$$

```
def rk4(plns: list[pln.Planet], dt: float, past=None)
    '''...
    ps = np.array([])
    for p in plns:
        p1 = p.rebuild()
        k1v = p.acc(plns)*dt
        p1.vel += 0.5*k1v
        k1r = p.vel*dt
        p1.pos += 0.5*k1r
        p2 = p.rebuild()
        k2v = p1.acc(plns)*dt
        p2.vel += 0.5*k2v
        k2r = p1.vel*dt
        p2.pos += 0.5*k2r
        p3 = p.rebuild()
        k3v = p2.acc(plns)*dt
        p3.vel += k3v
        k3r = p2.vel*dt
        p3.pos += k3r
        k4v = p3.acc(plns)*dt
        p.vel += (k1v + 2*k2v + 2*k3v + k4v)/6
        k4r = p3.vel*dt
        p.pos += (k1r + 2*k2r + 2*k3r + k4r)/6
        ps = np.append(ps, p.rebuild())
    return ps
```

3rd-order Adams-Bashforth:

$$y_{n+1} = y_n + \Delta t(23y_n' - 16y_{n-1}' + 5y_{n-2}')/12$$

```
def ab3(plns: list[pln.Planet], dt: float, past: list[np.ndarray[pln.Planet]])
    '''...
    ps = np.array([])
    p1s = past[-1]
    p2s = past[-2]
    for i in range(len(plns)):
        p, p1, p2 = plns[i], p1s[i], p2s[i]
        k0v = p.acc(plns)*dt
        k0r = p.vel*dt
        k1v = p1.acc(p1s)*dt
        k1r = p1.vel*dt
        k2v = p2.acc(p2s)*dt
        k2r = p2.vel*dt
        p.vel += (23*k0v - 16*k1v + 5*k2v)/12
        p.pos += (23*k0r - 16*k1r + 5*k2r)/12
        ps = np.append(ps, p.rebuild())
    return ps
```

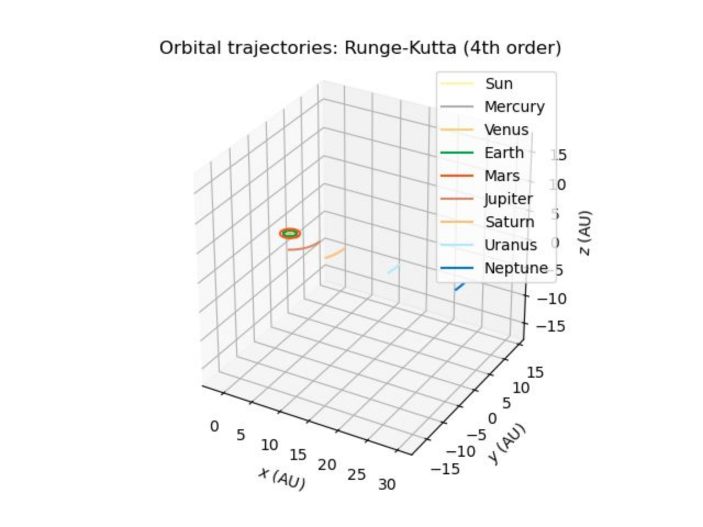
Iterative methods

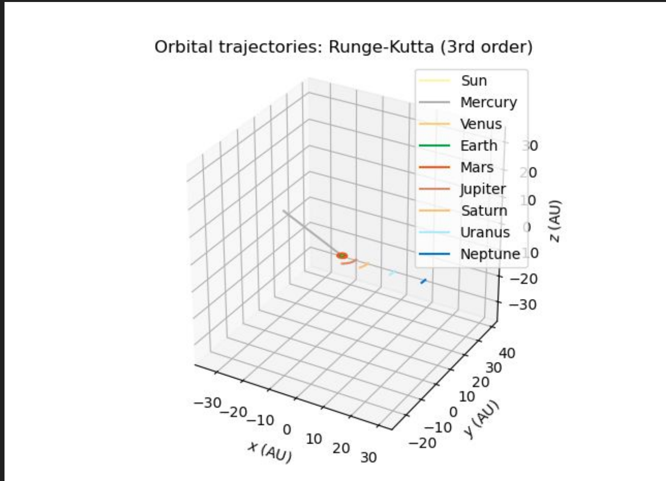
- Simulation shell: steps the method forward in time
  - Highly modular
  - AB methods get their first couple points from RK4

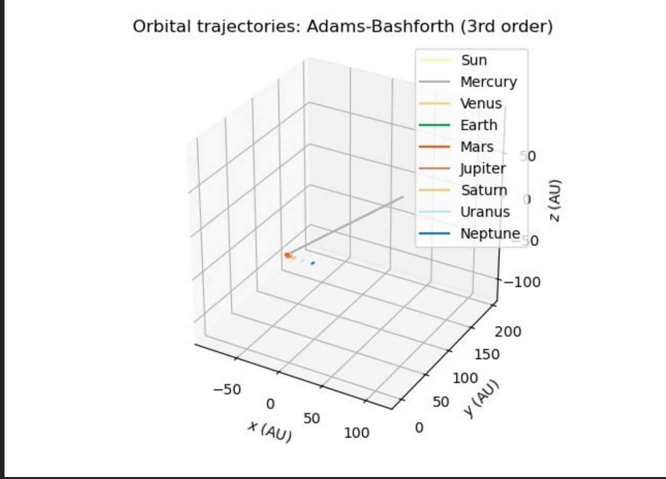
```python
def simulate(plns: list[pln.Planet], dt: float, tmax: float=5.0, method=euler) -> tuple:
    '''...

    ts = np.linspace(0, tmax, int(tmax/dt))
    all_ps = np.array([[p.rebuild() for p in plns]])
    init_num = 1
    if method.__name__[:2] == 'ab': init_num = int(method.__name__[-1])
    if init_num != 1:
        for t in ts[1:init_num]: all_ps = np.append(all_ps, [rk4(plns,dt)], axis=0)
    for t in ts[init_num:]: all_ps = np.append(all_ps, [method(plns,dt,all_ps[-init_num:-1])], axis=0)
    return all_ps, ts
```

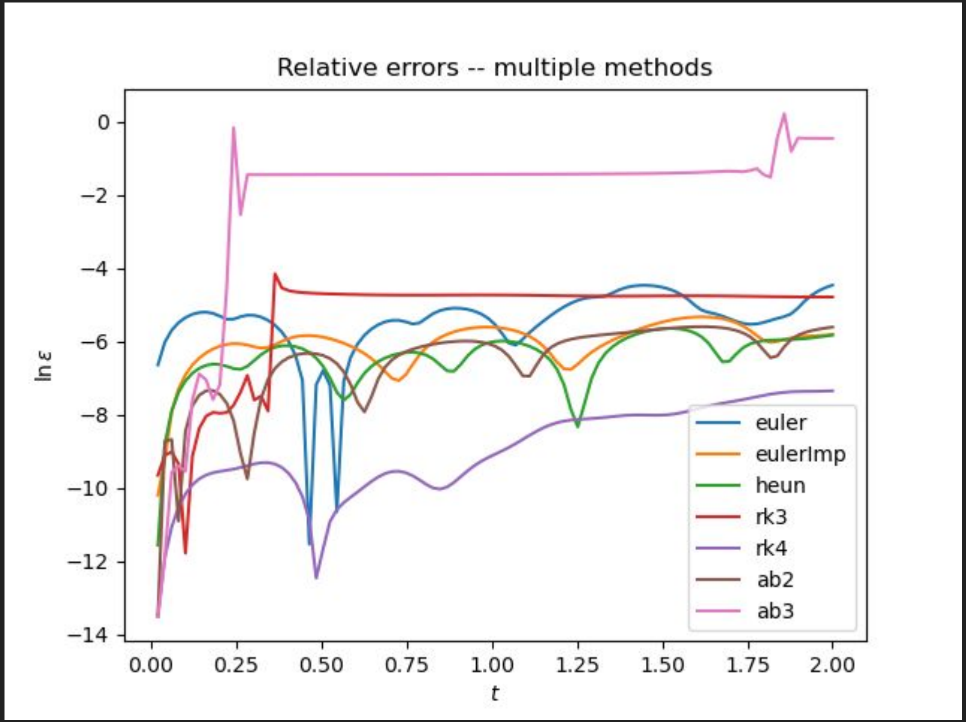
Results


Orbital trajectories: Runge-Kutta (4th order)
Sun
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
x (AU)
y (AU)
z (AU)


Orbital trajectories: Runge-Kutta (3rd order)
Sun
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
x (AU)
y (AU)
z (AU)


Orbital trajectories: Adams-Bashforth (3rd order)
Sun
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
x (AU)
y (AU)
z (AU)

Results

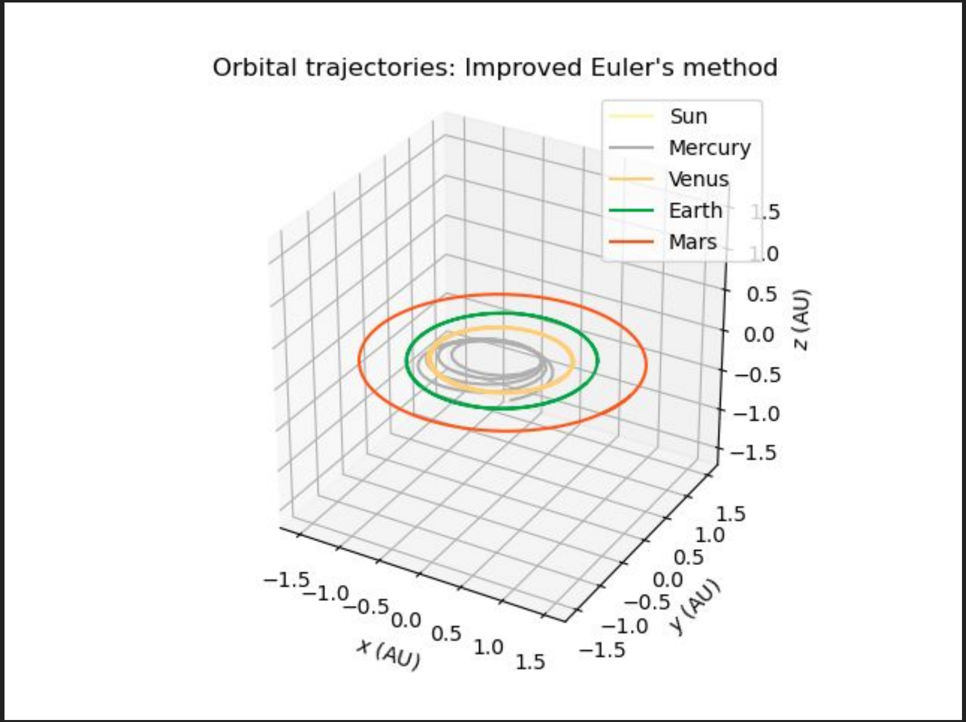


Relative errors -- multiple methods
0
−2
−4
−6
−8
−10
−12
−14
ln ε
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
t
euler
eulerImp
heun
rk3
rk4
ab2
ab3

Results

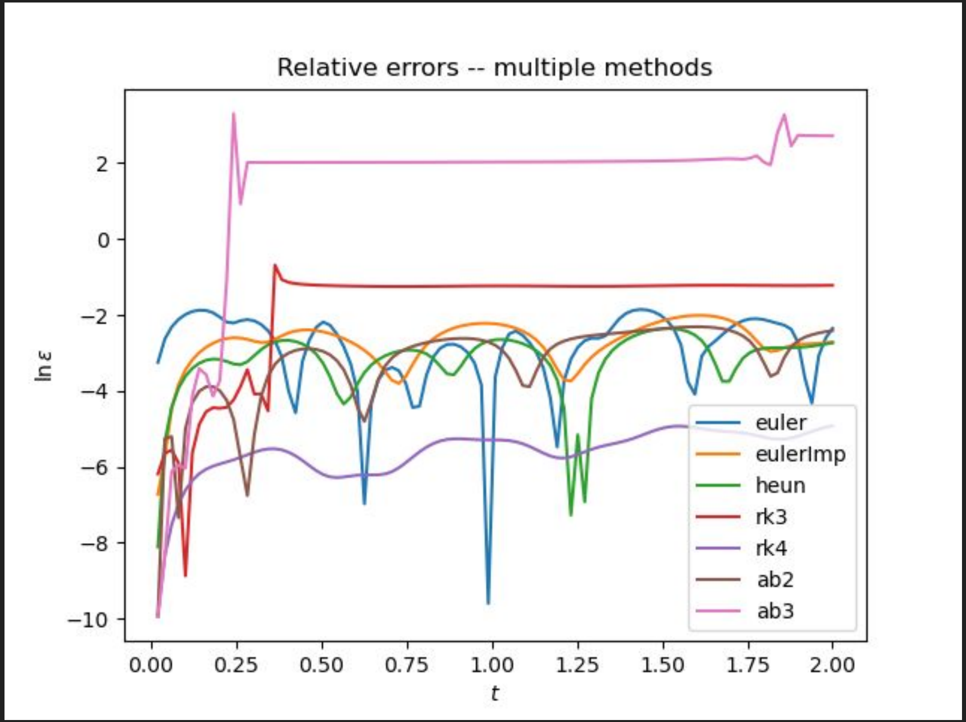
- The different timescales of the inner/outer planets' orbits make them difficult to model together
    - Mercury is particularly challenging – its small orbit makes it the most sensitive planet
- More useful to focus on 'chunks' of the solar system

Results



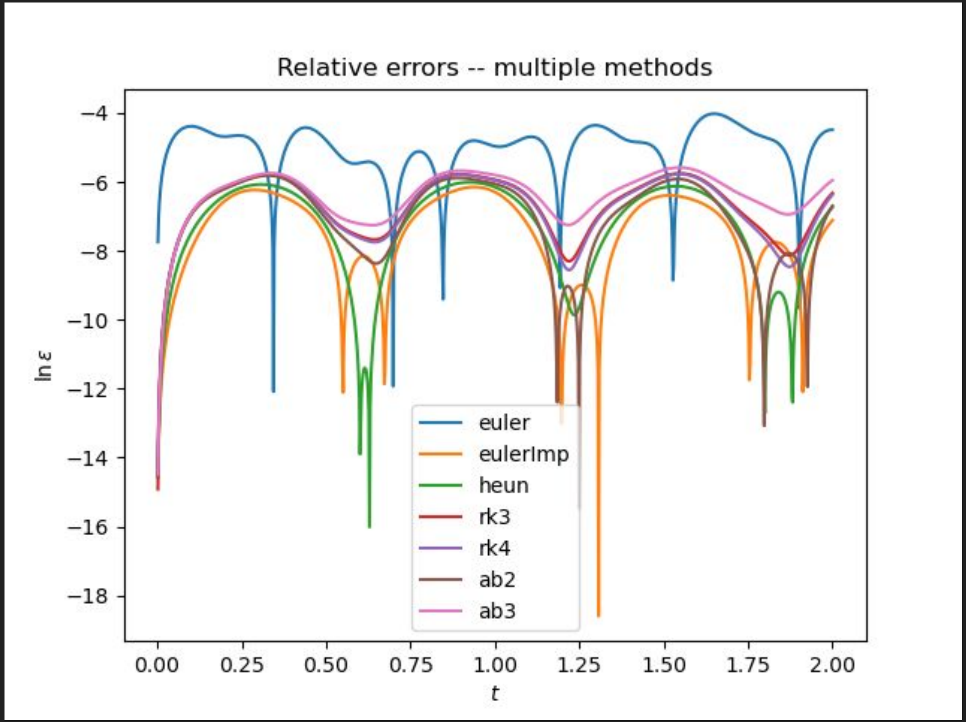
Orbital trajectories: Improved Euler's method
Sun
Mercury
Venus
Earth
Mars
x (AU)
y (AU)
z (AU)
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

Results



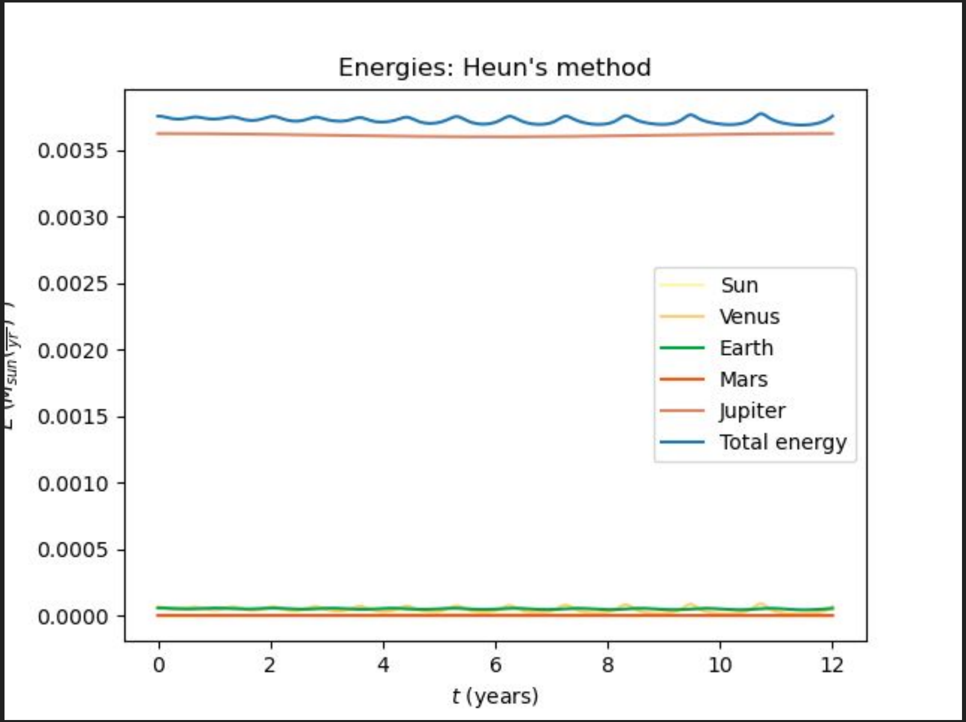
Relative errors -- multiple methods
$\ln \varepsilon$
t
2
0
−2
−4
−6
−8
−10
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
euler
eulerImp
heun
rk3
rk4
ab2
ab3

Results



Relative errors -- multiple methods
−4
−6
−8
−10
−12
−14
−16
−18
ln ε
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
t
euler
eulerImp
heun
rk3
rk4
ab2
ab3

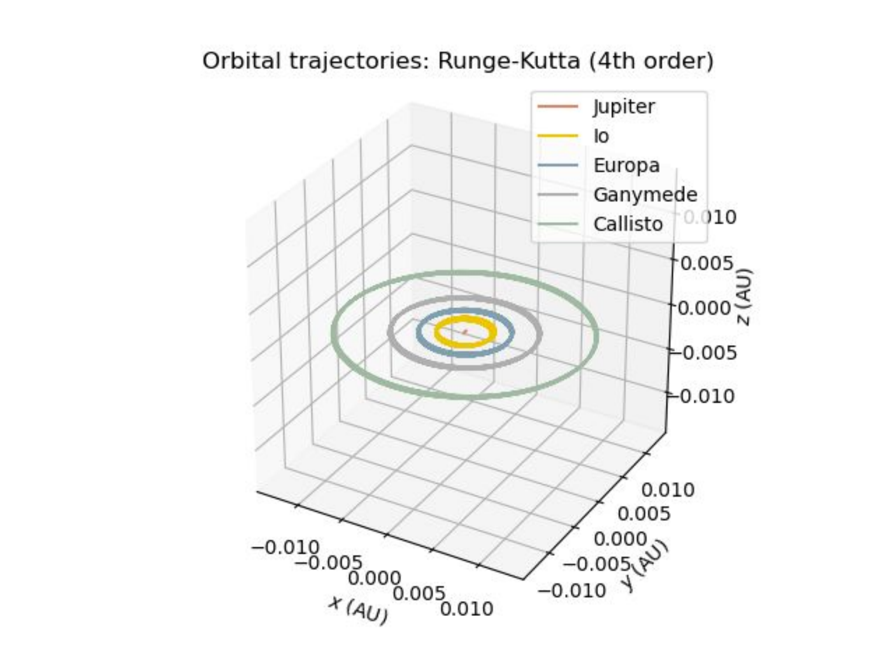
Results


Energies: Heun's method
0.0035
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
0
2
4
6
8
10
12
t (years)
Sun
Venus
Earth
Mars
Jupiter
Total energy

The Jovian system

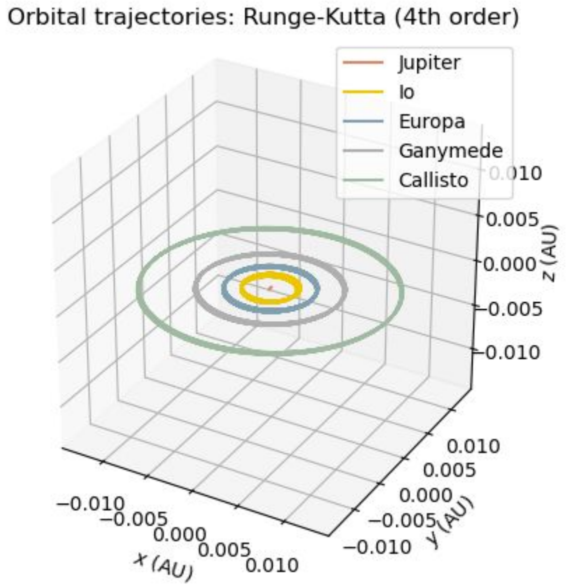
- The model's long-term behavior is not clear
- We need a smaller-scale system, like the moons of Jupiter
    - First observed by Galileo in 1610

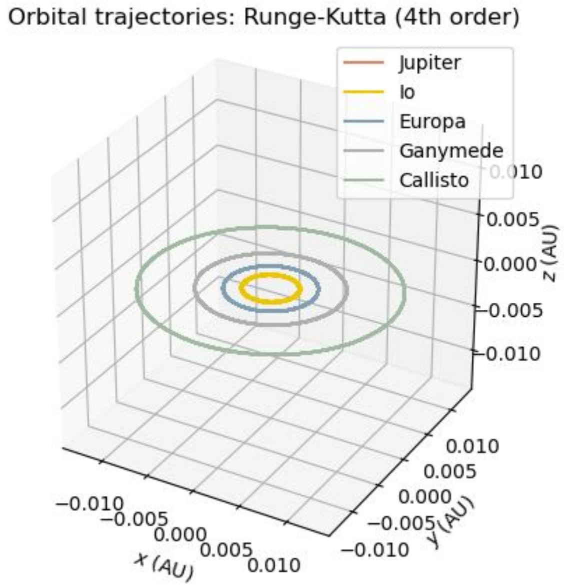
Results



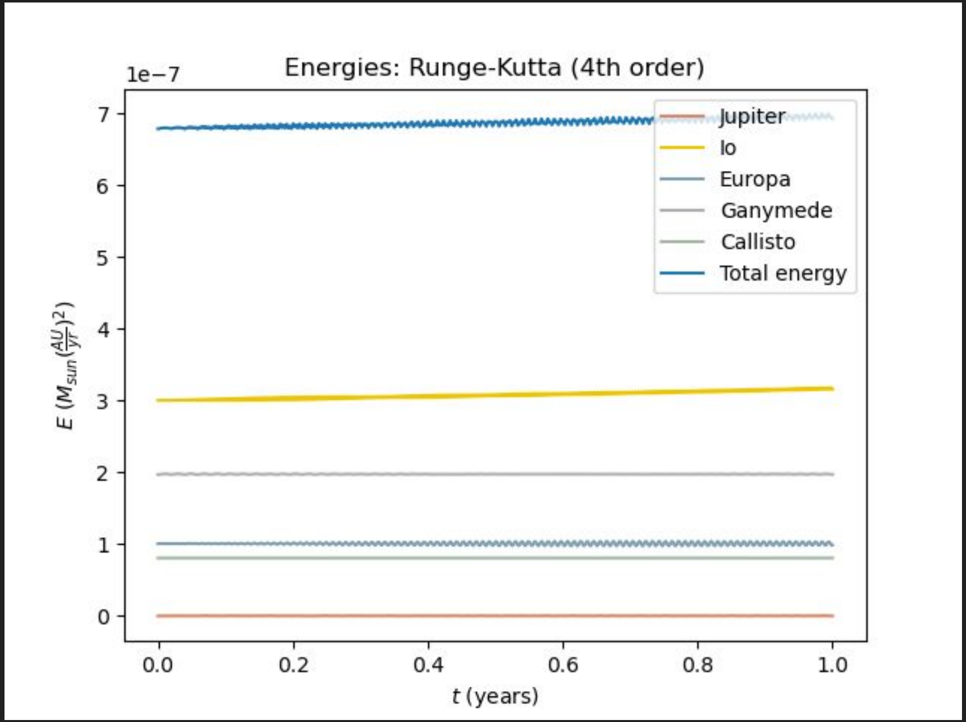
Orbital trajectories: Runge-Kutta (4th order)
Jupiter
Io
Europa
Ganymede
Callisto
x (AU)
y (AU)
z (AU)
−0.010
−0.005
0.000
0.005
0.010

Results


Orbital trajectories: Runge-Kutta (4th order)
Jupiter
Io
Europa
Ganymede
Callisto
0.010
0.005
0.000
−0.005
−0.010
z (AU)
0.010
0.005
0.000
−0.005
−0.010
y (AU)
−0.010
−0.005
0.000
0.005
0.010
x (AU)

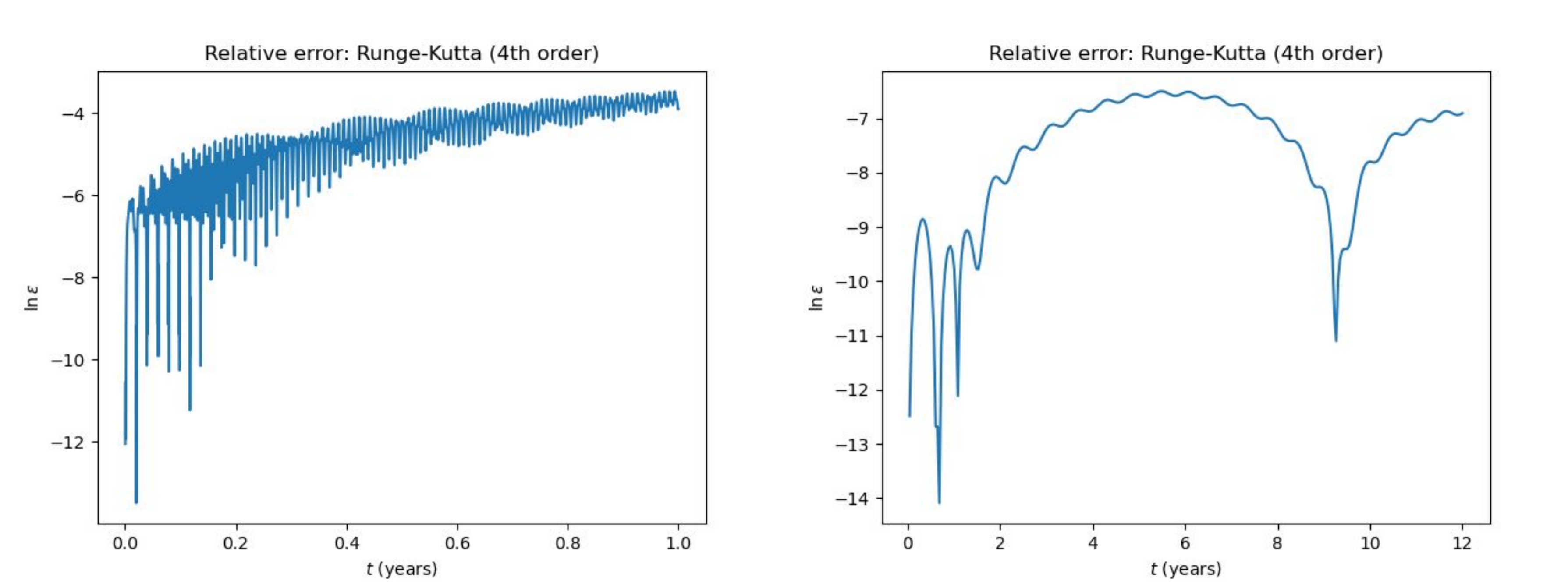

Orbital trajectories: Runge-Kutta (4th order)
Jupiter
Io
Europa
Ganymede
Callisto
0.010
0.005
0.000
−0.005
−0.010
z (AU)
0.010
0.005
0.000
−0.005
−0.010
y (AU)
−0.010
−0.005
0.000
0.005
0.010
x (AU)

Results



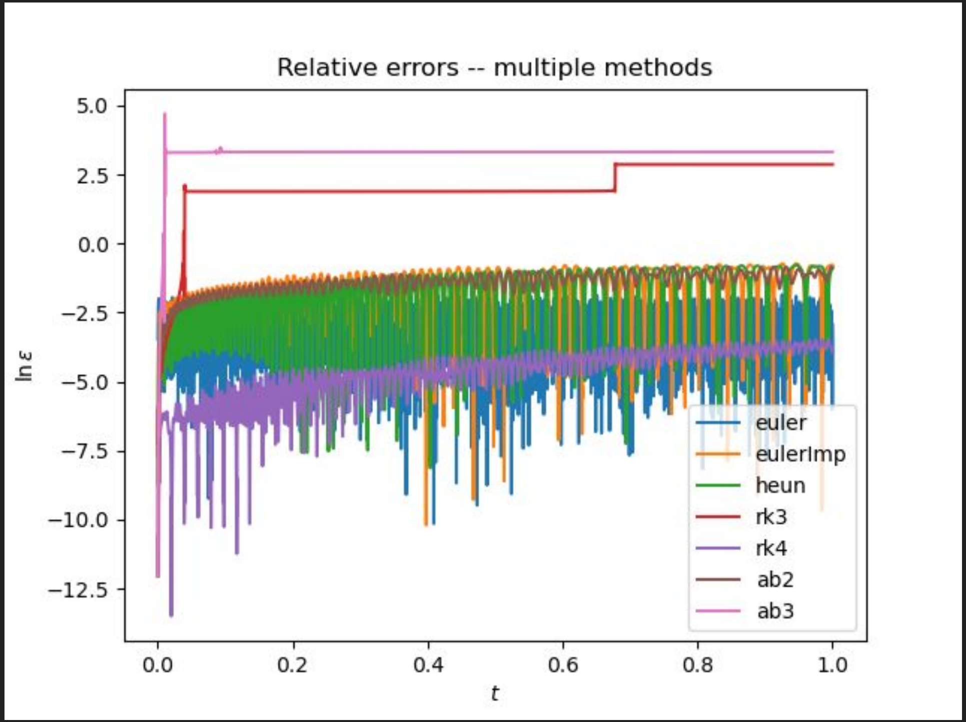
Energies: Runge-Kutta (4th order)
1e−7
7
6
5
4
3
2
1
0
$E\ (M_{sun}(\frac{AU}{yr})^2)$
0.0
0.2
0.4
0.6
0.8
1.0
t (years)
Jupiter
Io
Europa
Ganymede
Callisto
Total energy

Results



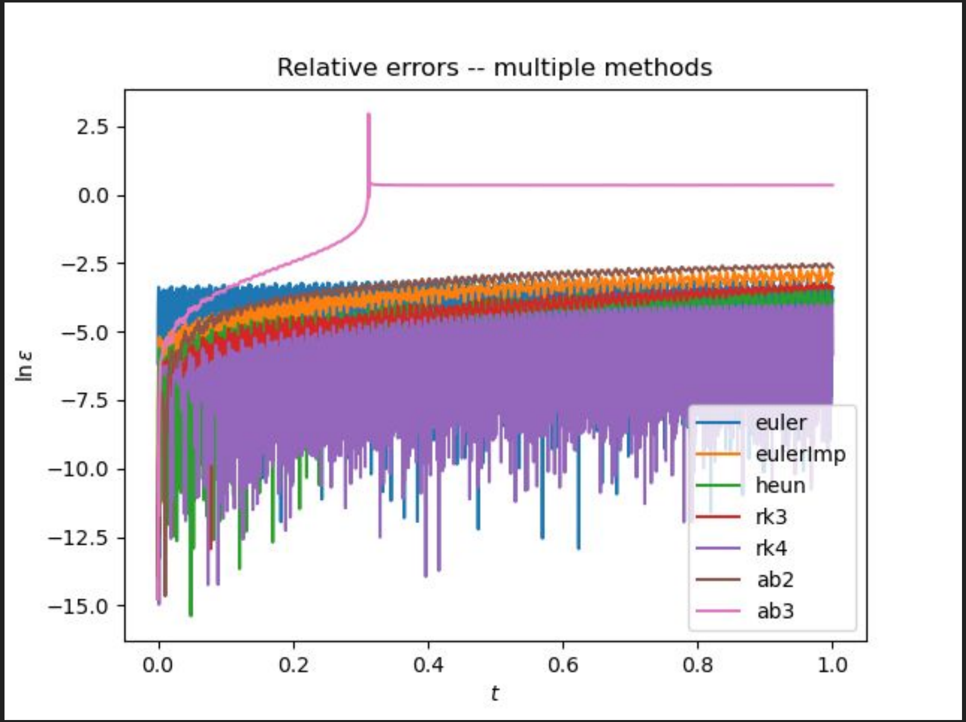
Relative error: Runge-Kutta (4th order)
ln ε
t (years)
-4
-6
-8
-10
-12
0.0
0.2
0.4
0.6
0.8
1.0
Relative error: Runge-Kutta (4th order)
ln ε
t (years)
-7
-8
-9
-10
-11
-12
-13
-14
0
2
4
6
8
10
12

Results



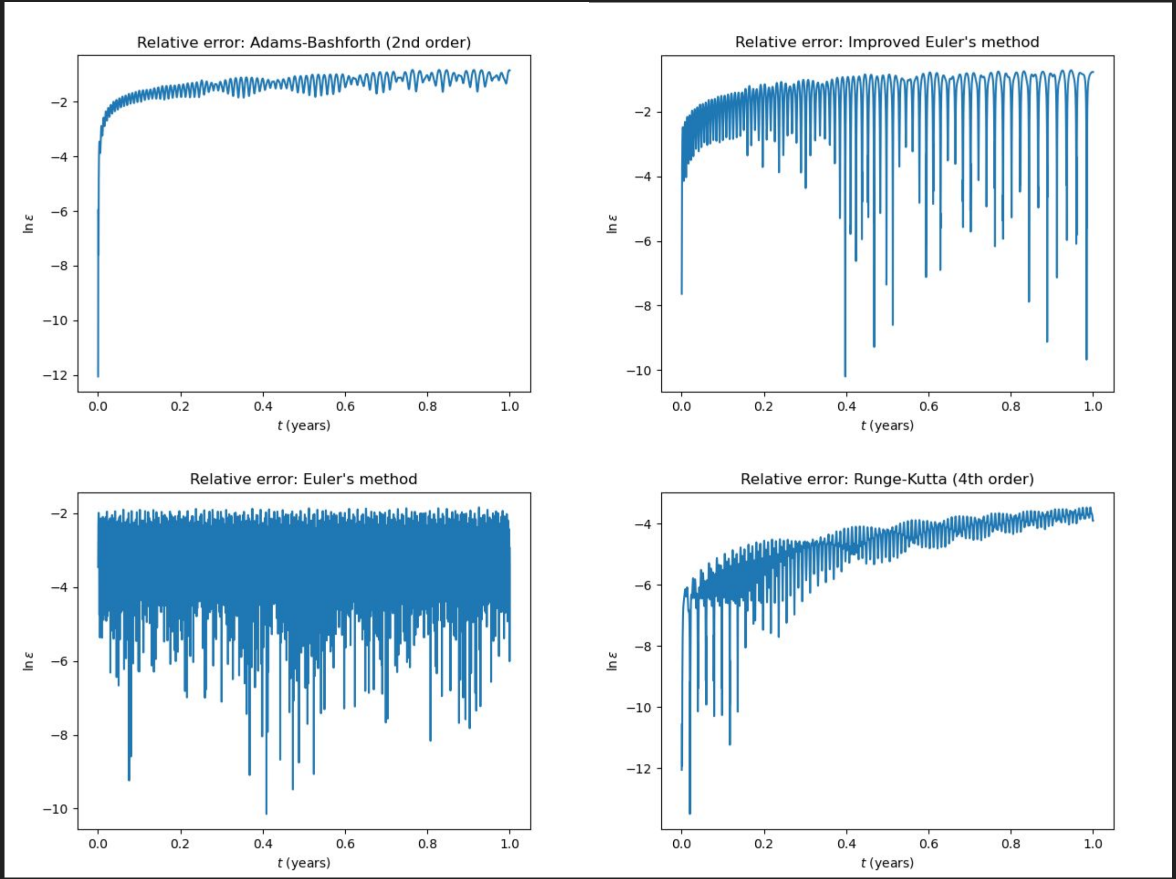
Relative errors -- multiple methods
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−12.5
ln ε
0.0
0.2
0.4
0.6
0.8
1.0
t
euler
eulerImp
heun
rk3
rk4
ab2
ab3

Results



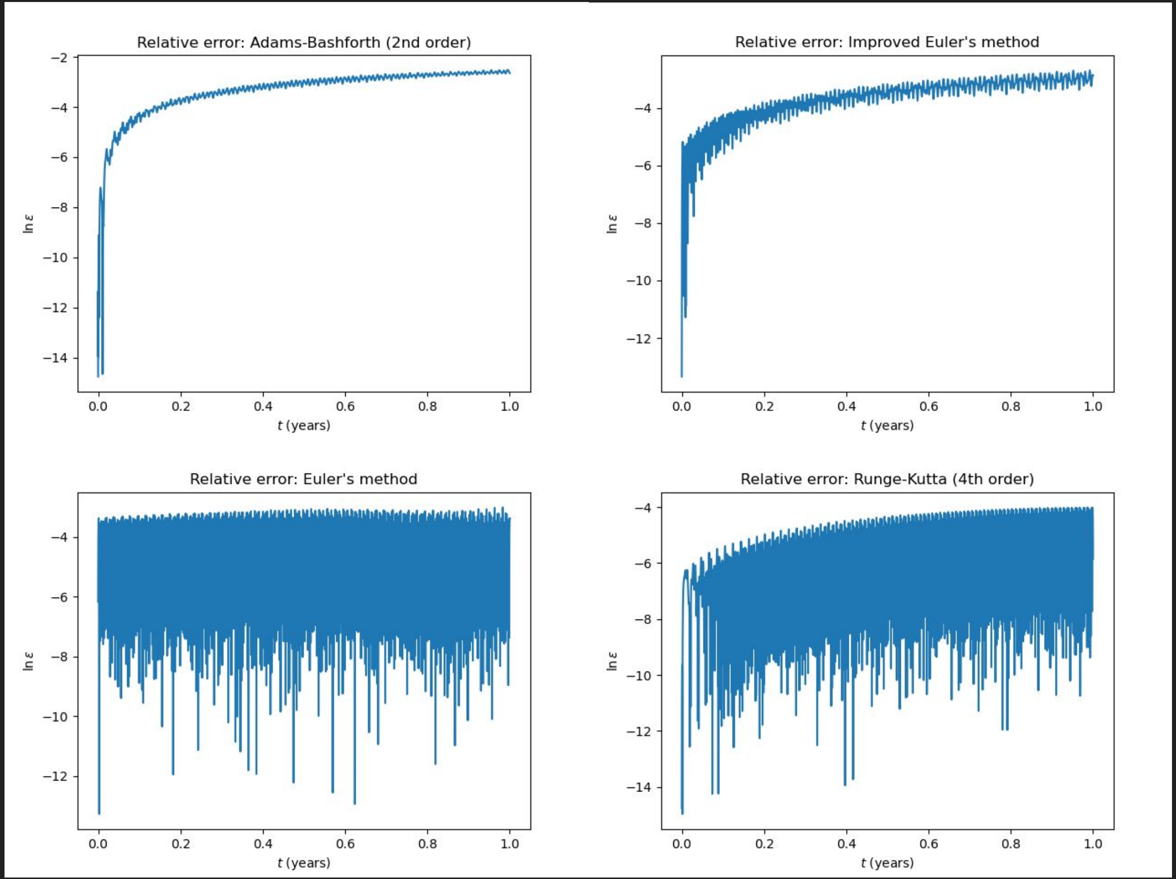
Relative errors -- multiple methods
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−12.5
−15.0
ln ε
0.0
0.2
0.4
0.6
0.8
1.0
t
euler
eulerImp
heun
rk3
rk4
ab2
ab3

Results



Relative error: Adams-Bashforth (2nd order)
Relative error: Improved Euler's method
Relative error: Euler's method
Relative error: Runge-Kutta (4th order)
ln ε
t (years)
0.0
0.2
0.4
0.6
0.8
1.0
−2
−4
−6
−8
−10
−12

Results


Relative error: Adams-Bashforth (2nd order)
Relative error: Improved Euler's method
Relative error: Euler's method
Relative error: Runge-Kutta (4th order)
ln ε
t (years)
0.0
0.2
0.4
0.6
0.8
1.0
−2
−4
−6
−8
−10
−12
−14

Results: maximum timestep

Euler’s method (1st order)	5.5e-4
Runge-Kutta (4th order)	2e-4
Heun’s method (2nd order)	1e-4
Improved Euler’s method (2nd order)	1e-4
Adams-Bashforth (2nd order)	1e-4
Runge-Kutta (3rd order)	6e-5
Adams-Bashforth (3rd order)	3e-5

Results: long-term* accuracy

Runge-Kutta (4th order)	-3.86
Runge-Kutta (3rd order)	-2.52
Heun’s method (2nd order)	-2.36
Adams-Bashforth (3rd order)	-2.18
Improved Euler’s method (2nd order)	-1.74
Adams-Bashforth (2nd order)	-1.65
Euler’s method (1st order)	-1.5

Conclusions

- Overall, the higher-order methods tend to produce more accurate results
- That said, they also tend to require smaller timesteps
- The all-around best method is 4th order Runge-Kutta

Expansions

- Could be repurposed to model almost any planetary system
- Easy to implement other iterative methods
  - ‘Purpose-built’ methods

Improvements

- More flexible satellites/parents
  - Moons of planets
  - Binary systems
- Eccentric orbits
- Longer-term simulations
  - Analysis of ‘collapse time’
  - Could be made more efficient

References

- D. Williams (Feb 2023), Planetary Fact Sheet, *National Aeronautics and Space Administration.* Retrieved from https://nssdc.gsfc.nasa.gov/planetary/factsheet/. Accessed Nov 2023.
- PIA01299: The Galilean Satellites (May 1998), *National Aeronautics and Space Administration*. Retrieved from https://photojournal.jpl.nasa.gov/catalog/PIA01299. Accessed Nov 2023.
- Planetary Satellites, *Jet Propulsion Laboratory.* Retrieved from https://ssd.jpl.nasa.gov/sats/. Accessed Nov 2023.
- S. Narayanamoorthy, T.L. Yookesh (2015), "Third Order Runge-Kutta Method for Solving Differential Equation in Fuzzy Environment," *International Journal of Pure and Applied Mathematics*, vol. 101, no. 5, p. 798. https://acadpubl.eu/jsi/2015-101-5-6-7-8/2015-101-8/21/21.pdf
- karlo (Jun 2015), *Stack Overflow.* Retrieved from https://stackoverflow.com/questions/13685386/how-to-set-the-equal-aspect-ratio-for-all-axes-x-y-z. Accessed Nov 2023.

Iterative methods were implemented as described in the class notes.