A Comparison For Methods Of Particle Based Simulation

Ben Lowe

April 2024

Abstract

In this project we have compared multiple numerical integration methods for modelling physical systems such as n-body systems, constant gravitational fields and a particle travelling through the earths core. In particular, the JPL ephemeris data has been used to compare the performance of the n-body simulation for different integration methods and time-steps by implementing a regex based method to import arbitrary ephemeris data into Particle class objects then compared different simulation methods using this data. In addition to this the graphics.py and matplotlib libraries have been used make graphical representations of the particle systems over time.

Our investigation concludes by finding that for n-body systems the midpoint method is 40.3 times as accurate as any first order method after the midpoint methods higher computational cost is taken into account. Resulting from the method being much better at conserving quantities such as energy. Explicit methods are more accurate at simulating constant forces than implicit methods, even compared to higher order methods which accumulated 112.5 times the error over the 5 simulation steps.

Introduction

To run our simulation we represent each interacting body[1] as a Particle object with some mass, and Vector objects to store the position and velocity in 3D space. Each moment in the evolution of our system is then represented with a SystemState object to store the time which this object represents and all the Particle objects in the model at this point in time. Finally to store all the data about the system through time we have a System object that stores information about both the states of the system being modelled and the method to find the next state of the model.

Each step of the integration is done by an Integrator object as well as the force object, these can be instantiated differently to produce simulations for each of the systems and tests we want to model.

The first type of simulation we're testing is the n-body simulation. In this simulation we instantiate n Particle objects within our System then we switch the force object to use the n-body force calculation. This is taken from Newtons law of universal gravitation 1.

$$F_i = \sum_{j=0}^{n} \frac{Gm_i m_j}{|r_j - r_i|^3}(r_j - r_i) \tag{1}$$

The code-base is also configured to simulate constant gravitational forces which approximate the gravitation field of the earths surface such as a projectile. The force from this is simply returned as the constant Vector(0, -9,81, 0).

Finally, the simulation can model a particle moving through a tunnel towards the core of the earth. The equasion to find this is derived from Newtons law of universal gravitation. We know that as a particle travels to the centre of a spherical object its the mass below it m_e compared to the total mass of the object M_e can be found geometrically using the equasion for the volume of a sphere[2]. It's found that $\frac{m_e}{M_e} = \frac{\frac{4}{3}\pi|\mathbf{r}|^3}{\frac{4}{3}\pi R_e^3}$. For $\mathbf{r}$ being the position vector measured from the centre of the earth and R_e is the radius of the earth. Then, $\frac{m_e}{M_e} = \frac{|\mathbf{r}|^3}{R_e^3}$. We can substitute this into the equasion for the Newtons law of universal gravitation acting on one body by equasion 2.

$$F_p = \frac{Gm_p m_e}{|-r_p|^3}(-r_p) = \frac{GM_e}{|-r_p|^2}\frac{|-r_p|^3}{R_e^3}(-r_p) = -\frac{GM_e}{R_e^3}r_p \tag{2}$$

This is all the simulations we implemented however the .create_force method can be used to allow for more eccentric forces to be applied to the simulation.

Also used for finding the next step in the simulation are the integration methods. We test five different integration methods: forwards Euler, backwards Euler, a semi implicit Euler, midpoint and rk4.

[1] Bodies such as planets, projectiles or particles

[2] This assumes that the earths interior has uniform density. It is a good approximation for small displacements about the interiors average density since magma is largely in-compressible.

For forwards Euler the integrator updates the position $r_{n+1} = r_n + v_n\delta$. Then calculates the force using the calculate method of the force object, divides by the Particle mass to get the acceleration and evaluates the new velocity $v_{n+1} = v_n + a_n\delta$ and returns Particle(r_{n+1}, v_{n+1}, m_n).

For backwards Euler integration the position is updated again as before, then a new virtual particle is created at the new position. We then input this new particle into the force calculation instead. And use this force as before to find the Particles next velocity.

The semi implicit Euler or Euler-Cromer method waits until the end to update the position. It then uses the velocity calculated through the same method as forwards Euler to get the new position $r_{n+1} = r_n + n_{n+1}\delta$.

Midpoint creates a Particle object at $r_{n+\frac{1}{2}} = r_n + v_n\delta$. And, uses this particle to calculate the force. Next updating the velocity and using the semi-implicit method to find the next position.

rk4 or the forth order Runge-Kutta integrator uses four different forces $K1, ..., K4$ calculated at four different particle positions represented by four different Particle objects $P1, ..., P4$. For our input particle $P0$ we first find that the force $K1 = F(P1) = F(P0)$ and use this force to find the position of $P2$ with a half time-step with semi implicit Euler. We evaluate the force $K2 = F(P2)$ and use this force to update $P1$ again with a half time-step to find $P3$. Using $P3$ to calculate the $K3 = F(P3)$ can then be applied to $P1$ again with a full time-step to get $P4$. Finally we can find $K4 = F(P4)$ and update our final velocities $v_{n+1} = v_n + \frac{\delta}{6m}(K1 + 2K2 + 2K3 + K4)$ and update the position $r_{n+1} = r_n + v_{n+1}\delta$.

Ephemeris Tests

We'll will be using the Ephemeris data from JPL to test performance of different integrators and time-steps as well as the time each of the integrators take to compute. This can be done by modelling the solar system as an n-body system then comparing our simulated results with the observations by calculating the average errors for each of the Particle positions.

First, the program opens the ephermeris files retrieved by hand from the Horizons System website before using regex to find the mass, position and velocity data. This is then initialised to a Particle object then these are used to initialise the stat state of the System. For our code the ephemeris data must be downloaded manually however as we'll see on the discussion of figure 2 using the Horizons API to import the data can be used to improve future implementations.

The System is then simulated using different integrators and time-steps. Then the average particle error is found between the simulated Systems final state, the Ephemeris data reference and a simulated reference using very small time-steps to test compared to observation and an idealised n-body respectively.

Using this method to change the time-step for forwards Euler integration results in 1 where it can be seen that changing the time-step linearly decreases

the error. This approaches a none zero error suggesting that increasingly small time-steps will not converge to the ephemeris observations. This may be due to other astronomical objects[3] not included in our model changing the ephemeris results. In future implementations this could be tested by using the Horizons API to load all the Ephemeris data and run the simulation without excluding these bodies.

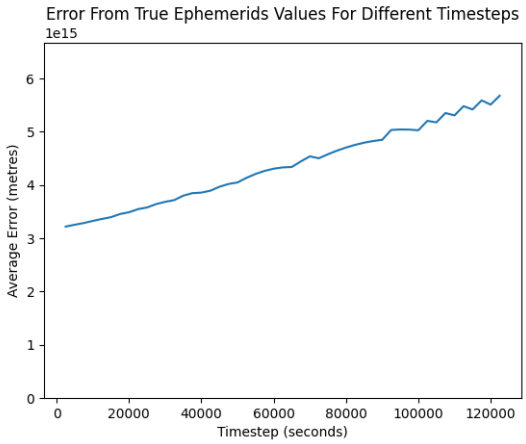

Figure 1: Plot of the errors for an n-body simulation using forwards Euler integration compared to Ephemeris data.

Next, comparing the different integration methods outlined above and comparing them with the Ephemeris tests give figure 2. All of the integration methods have the same convergence error as seen in the time-step test. The two higher order methods both have less error than the first order methods as they use intermediate force calculations which give a better approximation of the acceleration compared to the first order method. The best of the five integration methods for this simulation is the midpoint. This might be because the midpoint method is a symplectic method meaning it approximately conserves properties of the Hamiltonian. This is useful in our case because the planetary system is very stable[4] so energy and momentum must be conserved very well to give an accurate result.

Since, all the integrators have the same error associated with Ephemeris data not accounted for in our simulations it's useful to compare the integrators to a reference simulation with very small time-steps. This produced figure 3. It is seen that the midpoint method is 78 times as accurate compared to the semi implicit Euler method as well as being 12 times as accurate compared to the higher order rk4 method.

[3]The data used doesn't include data about moons which might influence the planets positions.

[4]A further example of this property of the midpoint method is demonstrated in section .

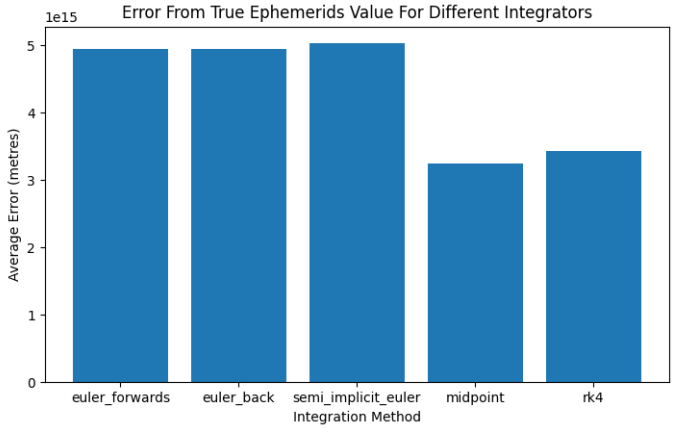

Figure 2: Comparison of different integration method errors from the Ephemeris data.

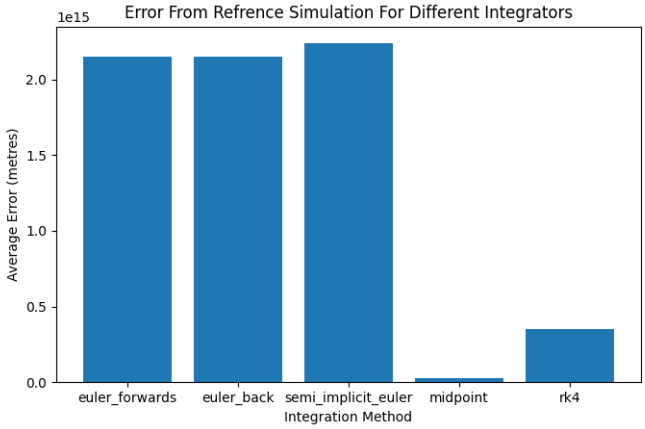

Figure 3: Comparison of different integration method errors from the a reference simulation with small time-steps.

We know however that since rk4 and the midpoint method both require more than one call to the force.calculate() method them will make them compute slower than the Eulerian methods or require larger time-steps which outweigh the benefits of this method. To quantify this the time standard liberary can be used to get information about the current time. Each of the simulations can be ran with a measure of the time before and after. Finding the difference then quantifies the time taken for each method to complete.

This is plotted in figure 4 where it is seen that unlike what we expect the

midpoint method is less than twice as fast as the expicit and semi explicit methods. This is likely due to inefficiencies with the implementation having to remove the integrating particle from the state each time one integration is computed. This is an $O(n)$ process so for larger number of particles other processes dominate[5] the computation time.

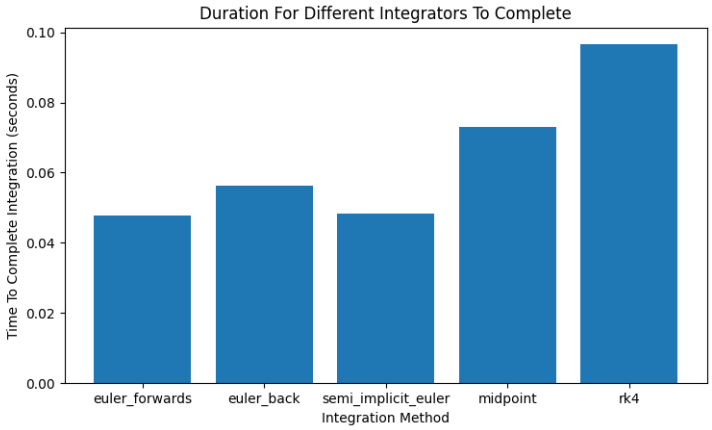

Figure 4: A Comparison of the computational time for each simulation method to finish.

Removing this inefficient but necessary step[6] gives the expected results 5 where the duration is proportional to the force.calculate calls in the integration. So approximately forth order rk4 method takes twice as long as the second order midpoint method which is twice as long as the explicit methods.

This shows us that for this problem the best integrator is the midpoint method as it's the most accurate while being sufficiently efficient that lowering the time-step for the explicit methods such that they take an equal duration to complete results in the midpoint method being 40.3 times as accurate[7] compared to the semi implicit euler method. Similarly, accounting for the computation time rk4 is 3.1 times more accurate than the semi implicit Euler method. However the extra complexity of rk4 may mean this isn't worth it for many applications[8].

[5]Integrating all the particles is $O(n^2) > O(n)$ for large n.

[6]Its necessary to remove the particle from state so that the n-body particles can't self interact.

[7]Assuming the error decreases linearly as in figure 1.

[8]Such as for the university outreach visualisations used on the RingMind project at Lancaster which uses a first order method with optimisations using the Barnes-Hut algorithm.

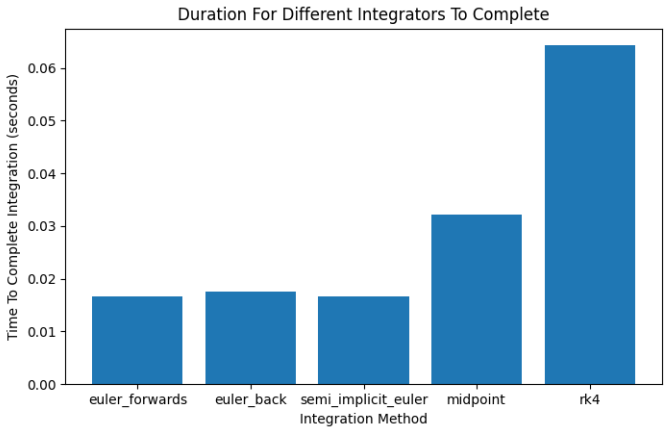

Figure 5: A Comparison of the computational time for each simulation method to finish having removed the O(n) time state.pop(i).

Earths Core Test

To compare the integrators capability we can also compare the results for the earth_core force. Since the force is directly proportional to the displacement we expect this to result in simple harmonic motion and so the position should be a sinusoidal curve. We'll be able to see that if energy is conserved by our integrators the amplitude of the sinusoidal curve will remain constant.

Plotting the displacements for the forwards Euler method, an explicit integration method and the midpoint method which is symplectic gives figure 6. It is seen that for the explicit Euler method the amplitude grows indicating that total energy of this model is increasing whereas the midpoint method has its amplitude increasing much slower than Euler forwards energy is increasing.

This can be seen more clearly in figure 7 where the energy for both of the integrators are oscillating however the explicit method is growing much more rapidly. Moreover, the energy of the explicit method appears to grow exponentially whereas the symplectic method grows much slower. This is best seen on longer time-spans such as in figure 8 where the symplectic method appears to be approximately constant whereas the energy for the explicit method grows rapidly.

Earth Surface

Both the tests above have made comparisons with either an accurate reference simulation or observations which may have attenuating factors not accounted for in the simulation. Some physical systems however we can find analytic solutions for, one example is a particle acting under a constant force such as the

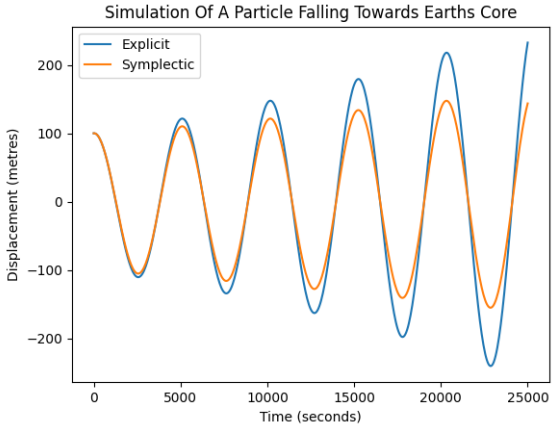

Figure 6: Displacement of a particle falling to the earths core using explicit and symplectic methods.

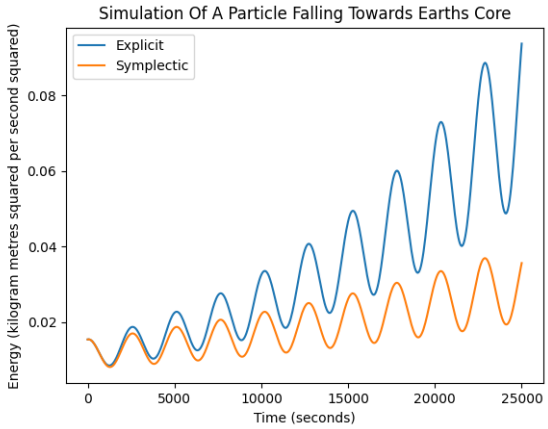

Figure 7: The total energy of a particle simulation for explcit and symplectic integration methods over the runtime of the simulation.

approximately constant gravitational field of the earth.

Under the earths gravitational field the particles will experience a constant force $F = -gm$ where $g \approx 9.81$ and m is the mass of the particle. This means the the acceleration is $a = -\frac{gm}{g} = -g$ a constant. Therefore since $\frac{dr}{dt} = a = g$ we integrate to find that $r = r_0 + v_0 t - gt^2$.

Running the simulation with different integration methods and comparing it with the theoretical values gives figure 9. The plot shows that in this situation the explicit methods are more accurate than the implicit methods. This is likely

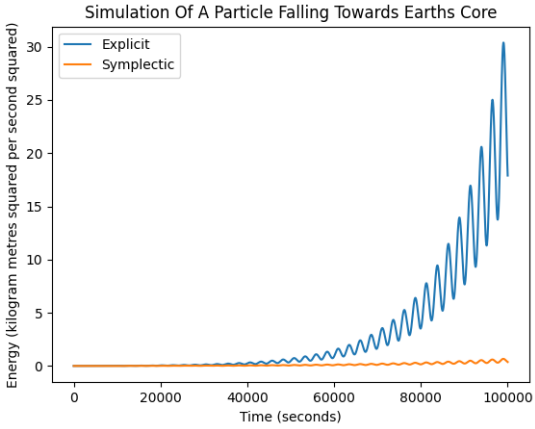

Figure 8: The total energy of a particle simulation for explcit and symplectic integration methods over a long runtime for the simulation.

because the particles velocity $v = v_0 - gt$ is linear and so the extra orders for rk4 and midpoint don't improve the accuracy.

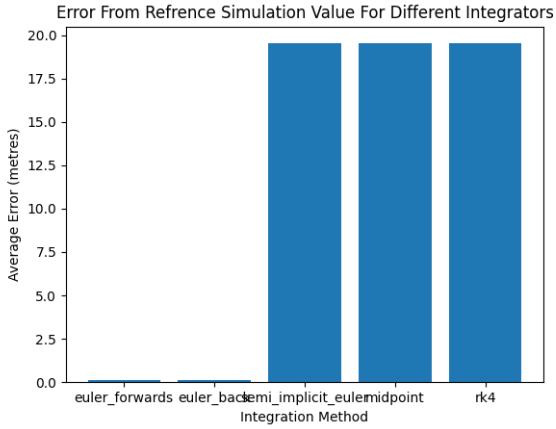

Figure 9: The error for different integration methods simulation a constant force.

The implicit methods then may be less accurate because they calculate the new position and then use the new position to update velocity as $r_{n+1} = r_n + v_{n+1}\delta$ rather than updating the position from the old velocity as $r_{n+1} = r_n + v_{n+1}\delta$. To test this, they can all be changed to update the position first, this produces figure 10. The error here is likely due to floating point error since

the distance.norm() can't return negative values and rk4 has involves more calculations so floating point error will build up more. Nevertheless, since the values are within $3e-15$ of one another this hypothesis for the source of implicit methods being more accurate is confirmed.

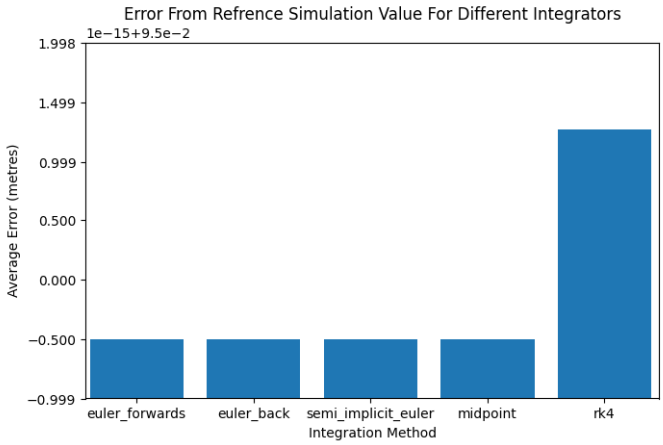

Figure 10: The error for different integration methods simulation a constant force by updating the position using $r_{n+1} = r_n + v_n\delta$

From this it can be concluded that future implementations should only use explicit methods for simulations involving a constant force.

Conclusion

It has been shown that for none linear simulations the midpoint method is most accurate even when the computation time is accounted for being 40.3 times more accurate at updating the positions of particles in a solar system simulation. Further it's seen that the midpoint method is best of those tested at conserving energy, this allows it to more accurately model stable systems such as harmonic oscillators and solar system simulations.

However, for linear simulations with constant forces explicit methods are required. This is because first order methods are sufficient for the first order system and therefore implicit methods which use the updated velocity to calculate the new position accumulate error in their position 112.5 times greater than the explicit methods for our test. Future implementations should consider using explicit methods if the force applied to particles is constant.

Further, it's been shown that the error increases approximately linearly with increasing time-steps for the simulation. And, that error accumulates exponentially the longer the simulation continues meaning that the rate at which the error accumulates increases with the error at some time.

From this we find that

References

[1] Giorgini, J. D., Yeomans, D. K., Chamberlin, A. B., Chodas, P. W., Jacobson, R. A., Keesey, M. S., Lieske, J. H., Ostro, S. J., Standish, E. M., Wimberly, R. N., "JPL's On-Line Solar System Data Service", Bulletin of the American Astronomical Society, Vol 28, p. 1099, 1997.

[2] OpenAI (2022). ChatGTP. Available at: chat.openai.com

[3] W3schools. Python Tutorial. Available at: w3schools.com

[4] M. Zelle. Graphics Refrence. Available at: mcsp.wartburg.edu/zelle/python/graphics

Code

Full code is otherwise available at https://github.com/BenLowe2003/ParticleSimulations .

Primitives.py creates Vector and Enum classes used in the Core module.
Dependencies: numpy

```
from numpy import sqrt

class Vector:
    """
    Class for handling vectors.
    Attributes:
        x (float): The x component of the vector.
        y (float): The y component of the vector.
        z (float): The z component of the vector.
    """

    def __init__(self, x, y, z):
        """
        Returns a Vector object.
        Args:
            x (float): x component of the Vector.
            y (float): y component of the Vector.
            z (float): z component of the Vector.
        Return:
            Vector object with x,y,z components.
        """
        self.x = x
        self.y = y
        self.z = z

    def __str__(self):
        """casts the Vector to a string"""
        return str(self.x) + ", " + str(self.y) + ", " + str(self.z)

    def __add__(self, other):
        """Operator adds two Vectors together."""
        x = self.x + other.x
        y = self.y + other.y
        z = self.z + other.z
        return Vector(x,y,z)

    def __sub__(self, other):
        """Operator subtracts two Vectors"""
        x = self.x - other.x
```

```
        y = self.y - other.y
        z = self.z - other.z
        return Vector(x,y,z)

    def __mul__(self, other):
        """Operator left multiplies a Vector by a float"""
        x = self.x * other
        y = self.y * other
        z = self.z * other
        return Vector(x,y,z)

    def __rmul__(self, other):
        """Operator right multiplies a Vector by a float."""
        return self.__mul__(other)

    def __truediv__(self, other):
        """Divides a Vector by a float. (Vector)"""
        if other != 0:
            return self.__mul__( 1 / other )
        else:
            raise ZeroDivisionError("Division of a Vector by Zero")

    def square_norm(self):
        """Returns the square of the norm of this Vector. (float)"""
        return self.x*self.x + self.y*self.y + self.z*self.z

    def norm(self):
        """Returns the norm of this Vector (float)"""
        return sqrt( self.square_norm())

    def transpose(self):
        """Returns (list <float>) the x,y,z coordinates of the Vector """
        return self.x, self.y, self.z

    def set_x(self, value):
        """Sets the x component of the Vector to a value (float)"""
        self.x = value
    def set_y(self, value):
        """Sets the y component of the Vector to a value (float)"""
        self.y = value
    def set_z(self, value):
        """Sets the z component of the Vector to a value (float)"""
        self.z = value

    def get_x(self):
        """returns the x value for this vector (float)"""
```

```
        return self.x
    def get_y(self):
        """returns the y value for this vector (float)"""
        return self.y
    def get_z(self):
        """returns the z value for this vector (float)"""
        return self.z


class Enum:

    def __init__(self, state_list, start = None):
        """Initialises a list of possitble states (list <any>) for the enum and
        self.state_list = state_list
        if start in state_list:
            self.state = start
        else:
            self.state = self.state_list[0]
        self.index = self.state_list.index(self.state)
        self.length = len(self.state_list)

    def cycle(self, n = 1):
        """Cycles through the nth (int) value of the enum"""
        self.index = (self.index + n) % self.length
        self.state = self.state_list[self.index]

    def set(self, state):
        """Sets the Enum to some state (any)"""
        if state in self.state_list:
            self.state = state

    def set_index(self, i):
        """returns the nth (int) value of teh Enum"""
        if i < self.length:
            self.state = self.state_list[i]

    def get_state(self):
        """Returns the current state of the Enum (any)"""
        return self.state

    def get_index(self):
        """gets the index of the current state of the Enum (int)"""
        return self.index

    def get_state_list(self):
        """Returns all possible states of the Enum (list <any>)"""
```

```
        return self.state_list

    def add(self, value):
        """Adds a state to the Enum (any)"""
        self.state_list.append(value)
```

Core.py contains all the classes to run the particle simulations.
Dependencies: Primitives.py, copy

```
    from Primitives import *
from copy import deepcopy

class Particle:
    """
    Object to hold all the infomation concerning a single particle in the system
    Attributes:
        position (Vector): The position of the particle.
        velocity (Vector): The velocity of the particle.
        mass (float): The mass of the particle.
    """

    def __init__(self, position, velocity,  mass):
        """
        Initialises a Particle object.
        Args:
            position (Vector): A Vector containing the x,y,z components of the p
            velocity (Vector): A Vector containing the x,y,z components of the p
            mass (float): A float to store the particles mass.
        Returns:
            Particle object.
        """
        self.velocity = velocity
        self.position = position
        self.mass = float(mass)

    def __str__(self):
        """Returns a string to display the particles attributes"""
        return " Velocity: {} Position: {} Mass: {}".format(self.velocity, self.

    def step(self, past_state, integrator, force, dt):
        """
        Completes one physics step, returning the resultant particle.
        Args:
            past_state (SystenState): The previous state of the system from whic
            integrator (Integrator): The Integrator object which computes the ne
            force (Force): The Force object to find all the forces applied to th
            dt (float): The small timestep over which the integration is done.
```

```
            Return:
                Particle with the attributes of this particle at the next timestep.
            """
            return integrator.integrate(past_state, self, force, dt)

        def get_position(self):
            """Returns this Particles position."""
            return self.position

        def get_velocity(self):
            """Returns this Particles velocity."""
            return self.velocity

        def get_mass(self):
            """Returns this Particles mass."""
            return self.mass

        def get_momentum(self):
            return self.mass * self.velocity

        def get_energy(self):
            G  = 6.6743015e-11
            kinetic_energy = (1/2) * self.mass * self.velocity.square_norm()

            earth_mass = 5.97219e24
            earth_radius = 6378137
            G = 6.6743015e-11

            distance = self.get_position()
            numerator = G * self.get_mass() * earth_mass
            denominator = earth_radius * earth_radius * earth_radius
            potential_energy = (numerator / denominator) * distance.square_norm()

            return kinetic_energy + potential_energy


    class SystemState:
        """
        Object for holding all the information concerning the physical system at som
        Attributes:
            particles (Particles[]): Array stores all the Particles in the system.
            num_particles (int): How many particles are in the system at this time.
            time (float): The time for which the object holds system information for
        """

        def __init__(self, particles, time, dt):
```

```
        """
        Initialises a SystemState object.
        Args:
            particles (Particle[]): An array for the initial value of the partic
            time (float): the initial time of the system.
        """
        self.particles = particles
        self.time = time

    def __str__(self):
        """Returns a string displaying the time and all the particles as a Parti
        string = ' '
        for particle in self.particles:
            string += "\n"
            string += str( particle )
        return string

    def step(self, integrator, force, dt):
        """
        Computes a the next state of the system after a small time step.
        Args:
            integrator (Integrator): An Integrator to compute the numerical inte
            force (Force): A Force object to compute the force on a particle.
            dt (float): The small timestep overwhich the the numerical integratio
        """
        new_system = SystemState([], self.time + dt, dt)
        for i in range(len(self.particles)):

            #new_state = deepcopy(self)
            #new_state.pop(i)

            new_particle = self.particles[i].step(self, integrator, force, dt)
            new_system.add_particle(new_particle)
        return new_system

    def add_particle(self, particle):
        """
        Adds a particle to the SystemState.
        Args:
            particle (Particle): Adds this Particle to the SystemState.
        """
        self.particles.append(particle)

    def get_particle(self, i):
        """Returns the ith (int) Particle (Particle)"""
        return self.particles[i]
```

```
def get_particles(self):
    """Returns all the particles (list <Particle>)"""
    return self.particles

def get_time(self):
    """Returns the simulation time of this state (float)"""
    return self.time

def num_particles(self):
    """Returns the number of Particles stored in the state (int)"""
    return len(self.particles)

def __mul__(self, other):
    """Compares this and some other (SystemState) returns the error (float)
    error = 0
    for i in range(self.num_particles()):
        self_particle = self.get_particle(i)
        other_particle = other.get_particle(i)
        difference = self_particle.get_position() - other_particle.get_posit
        error += difference.norm()
    error /= self.num_particles()
    return error

def total_momentum(self):
    """Returns the total momentum of the state (float)"""
    total_momentum = Vector(0,0,0)
    for particle in self.particles:
        total_momentum += particle.get_momentum()
    return total_momentum

def centre_mass(self):
    """Returns the centre of mass of the state (Vector)"""
    centre = Vector(0,0,0)
    total_mass = 0
    for particle in self.particles:
        particle_position = particle.get_position()
        particle_mass = particle.get_mass()
        particle_contribution = particle_position * particle_mass
        total_mass += particle_mass
        centre += particle_contribution
    centre /= total_mass
    return centre

def remove(self, particle):
    """Removes some input (Particle)"""
```

```
        self.particles.remove(particle)

    def pop(self, index):
        """Removes the ith (int) particle and returns it (Particle)"""
        particle = self.particles.pop(index)
        return particle

    def get_energy(self):
        """Finds the states energy for a earth_core system (float)"""
        total_energy = 0
        for particle in self.particles:
            total_energy = particle.get_energy()
        return total_energy



class Force:

    def earth_gravity(self, particle, state):
        """Returns the constant earther surface gravity (Vector) for a (Particle
        return Vector(0, -9.81, 0)

    def n_body(self, particle, state):
        """Returns the n-body force for a (Particle) and (SystemState)"""
        total_force = Vector(0, 0, 0)
        G = 6.6743015e-11
        for other_particle in state.get_particles():
            displacement = other_particle.get_position() - particle.get_position
            denominator = displacement.norm() * displacement.square_norm()
            if other_particle != particle and denominator != 0:
                numerator = G * other_particle.get_mass() * particle.get_mass()
                attraction_force = (numerator / denominator) * displacement
                total_force += attraction_force
        return total_force

    def earth_core(self, particle, state):
        """Returns the earth core force for a (Particle) and (SystemState)"""
        earth_mass = 5.97219e24
        earth_radius = 6378137
        G = 6.6743015e-11

        distance = particle.get_position()
        numerator = G * particle.get_mass() * earth_mass
        denominator = earth_radius * earth_radius * earth_radius
        force = (-numerator / denominator) * particle.get_position()
        return force
```

```
    def __init__(self, G = 1):
        """Initialises a force object (Force)"""

        self.force_enum = Enum([self.earth_gravity, self.n_body, self.earth_core
        self.G = G

    def cycle(self, n = 1):
        """Cycles through the different force calculation functions (int)"""
        self.force_enum.cycle(n)

    def set_force(self, func):
        """Sets the force function (function)"""
        self.force_enum.set(func)

    def switch(self, string):
        """switches to the force with some name (str)"""
        for method in self.force_enum.get_state_list():
            if method.__name__ == string:
                self.force_enum.set(method)

    def calculate(self, particle, state):
        """calculates the force for a (Particle) and (SystemState)"""
        func = self.force_enum.get_state()
        out = func(particle, state)
        return out

    def new_force(self, force):
        self.force_enum.add(force)

class Integrator:

    def euler_forwards(self, particle, state, force, dt):
        """Input (Particle), (SystemState), (Force), timestep (float) return nex
        position = particle.get_position() + particle.get_velocity() * dt
        particle_force = force.calculate(particle, state)
        acceleration = particle_force / particle.get_mass()
        velocity = particle.get_velocity() + acceleration * dt
        return Particle(position, velocity, particle.get_mass())

    def euler_back(self, particle, state, force, dt):
        """Input (Particle), (SystemState), (Force), timestep (float) return nex
        position = particle.get_position() + particle.get_velocity() * dt
        test_particle = Particle(position, particle.get_velocity(), particle.get
        particle_force = force.calculate(particle, state)
```

```
        acceleration = particle_force / particle.get_mass()
        velocity = particle.get_velocity() + acceleration * dt
        return Particle(position, velocity, particle.get_mass())

    def semi_implicit_euler(self, particle, state, force, dt):
        """Input (Particle), (SystemState), (Force), timestep (float) return nex
        particle_force = force.calculate(particle, state)
        acceleration = particle_force / particle.get_mass()
        velocity = particle.get_velocity() + acceleration * dt
        position = particle.get_position() + velocity * dt
        return Particle(position, velocity, particle.get_mass())

    def midpoint(self, particle, state, force, dt):
        particle_force = force.calculate(particle, state)
        acceleration = particle_force / particle.get_mass()
        velocity_mid = particle.get_velocity() + acceleration * dt * 0.5
        position_mid = particle.get_position() + velocity_mid * dt * 0.5
        ref_particle = Particle(position_mid, velocity_mid, particle.get_mass())
        particle_force = force.calculate(ref_particle, state)
        acceleration_mid = particle_force / particle.get_mass()
        velocity = particle.get_velocity() + acceleration_mid * dt
        position = particle.get_position() + velocity * dt
        return Particle(position, velocity, particle.get_mass())

    def rk4(self, particle, state, force, dt):
        """Input (Particle), (SystemState), (Force), timestep (float) return nex
        mass = particle.get_mass()
        k1 = force.calculate(particle, state)
        k2_particle = Particle(
            particle.get_position() + 0.5 * particle.get_velocity() * dt,
            particle.get_velocity() + 0.5 * k1 * dt / mass,
            particle.get_mass())
        k2 = force.calculate(k2_particle, state)
        k3_particle = Particle(
            particle.get_position() + 0.5 * k2_particle.get_velocity() * dt,
            particle.get_velocity() + 0.5 * k2 * dt / mass,
            particle.get_mass())
        k3 = force.calculate(k3_particle, state)
        k4_particle = Particle(
            particle.get_position() + k3_particle.get_velocity() * dt,
            particle.get_velocity() + k3 * dt / mass,
            particle.get_mass())
        k4 = force.calculate(k4_particle, state)
        particle_force = (1/6) * (k1 + 2*k2 + 2*k3 + k4)
        acceleration = particle_force / particle.get_mass()
        velocity = particle.get_velocity() + acceleration * dt
```

```
        position = particle.get_position() + velocity * dt
        return Particle(position, velocity, particle.get_mass())


    def __init__(self):
        """Initialise a new Integrator"""

        self.integration_method = Enum([self.euler_forwards,
                                        self.euler_back,
                                        self.semi_implicit_euler,
                                        self.midpoint,
                                        self.rk4])

    def integrate(self, state, particle, force, dt):
        """Computes the integration for some (Particle), (SystemState), (Force),
        func = self.integration_method.get_state()
        out = func(particle, state, force, dt)
        return out

    def switch(self, string):
        """Switch to a different integration method (str)"""
        for method in self.integration_method.get_state_list():
            if method.__name__ == string:
                self.integration_method.set(method)

    def cycle(self, n):
        """Cycle to a different integration method (int)"""
        self.integration_method.cycle(n)

    def __str__(self):
        return self.integration_method.get_state().__name__


class System:

    def __init__(self, init_state, integrator, force, dt):
        """initialises a System with initaial (SystemState), (Integrator), (Forc

        self.states = [init_state]
        self.integrator = integrator
        self.force = force
        self.dt = dt

    def step(self, n = 1):
        """Computes n (int) integration steps returning the final (SystemState)"
        for _ in range(n):
```

```
            old_state = self.states[-1]
            new_state = old_state.step(self.integrator, self.force, self.dt)
            self.states.append(new_state)
        return self.states[-1]

    def step_time(self, time):
        """computes steps until some time (float)"""
        state = self.states[-1]
        while state.get_time() < time:
            state = self.step(1)
        return state

    def get_state(self, i):
        """Returns tha (SystemState) with some index"""
        return self.states[i]

    def get_state_time(self, time):
        """Returns the (SystemState) for a certain input time (float)"""
        for state in self.states:
            if state.get_time() + self.get_dt()/2 >= time:
                return state

    def get_states(self):
        """Returns all (list <SystemStates>)"""
        return self.states

    def get_dt(self):
        """Returns the timestep (float)"""
        return self.dt

    def num_states(self):
        """Returns (int) the number of (SystemStates) stored"""
        return len(self.states)

    def __str__(self):

        string = ''
        for state in self.states:
            string += " Time: "
            string += str(state.time)
            string += str(state)
            string += "\n \n"
        return string

    def __mul__(self, other):
        """Returns (float) the total error between this and another (System) """
```

```
        error = 0
        for i in range(self.num_states()):
            my_state    = self.get_state(i)
            other_state = other.get_state(i)
            error_change = my_state * other_state
            error += error_change
        error /= self.num_states()
        return error

    def get_momentum(self):
        """Returns (list <float>) the momentum of all states"""
        momentum_list = []
        for state in self.states:
            momentum_list.append(state.total_momentum())
        return momentum_list

    def get_times(self):

        times_list = []
        for state in self.states:
            times_list.append(state.get_time())
        return times_list

    def get_energies(self):
        energy_list = [ state.get_energy() for state in self.states]
        return energy_list
```

EarthCoreTest.py is used to test the earth core force and create figures , 7, 8 and 6

Dependencies: Core.py re

```
  from Core import *
from matplotlib import pyplot as plt
#from Render import *

integrator_explicit = Integrator()
integrator_explicit.switch('euler_forwards')
integrator_syplectic = Integrator()
integrator_syplectic.switch('midpoint')

force = Force()
force.switch('earth_core')

dt = 50
total_steps = 2000

particle_init = Particle(Vector(100, 0, 0), Vector(0, 0, 0, ), 1)
```

```
state_init = SystemState([particle_init], 0, dt)
system_explicit = System(state_init, integrator_explicit, force, dt)
system_symplectic = System(state_init, integrator_syplectic, force, dt)

system_explicit.step(total_steps)
system_symplectic.step(total_steps)

x_position_explicit = [ state.get_particle(0).get_position().get_x()
                        for state in system_explicit.get_states()]
x_position_symplectic = [ state.get_particle(0).get_position().get_x()
                        for state in system_symplectic.get_states()]
times = [dt * i for i in range(total_steps+1)]

plt.plot(times, x_position_explicit, label = 'Explicit')
plt.plot(times, x_position_symplectic, label = 'Symplectic')
plt.legend()
plt.xlabel('Time (seconds)')
plt.ylabel('Displacement (metres)')
plt.title('Simulation Of A Particle Falling Towards Earths Core')
plt.show()

energies_explicit = system_explicit.get_energies()
energies_symplectic = system_symplectic.get_energies()

plt.plot(times, energies_explicit, label = 'Explicit')
plt.plot(times, energies_symplectic, label = 'Symplectic')
plt.legend()
plt.xlabel('Time (seconds)')
plt.ylabel('Energy (kilogram metres squared per second squared)')
plt.title('Simulation Of A Particle Falling Towards Earths Core')
plt.show()

#display_system(system_explicit, scale = 1)
```

Ephemeris.py is used to read the ephemeris data for other modules to import.

```
    import re
from Core import Particle, Vector

mass_patterns = [r"Mass,\s*10\^24\s*kg\s*=\s*~?(\d+(?:,\d{3})*)(?:\.\d+)?",
                 r"Mass\s+x10\^?23\s*\(kg\)\s*=\s*([-+]?\d*\.?\d+(?:[Ee][-+]?\d+
                 r"Mass\s+x10\^?24\s*\(kg\)\s*=\s*([-+]?\d*\.?\d+(?:[Ee][-+]?\d+
                 r"Mass\s+x\s*10\^?22\s*\(g\)\s*=\s*([-+]?\d*\.?\d+(?:[Ee][-+]?\
                 r"Mass\s+x10\^?26\s*\(kg\)\s*=\s*([-+]?\d*\.?\d+(?:[Ee][-+]?\d+

scales = [10 ** 24, 10 ** 23, 19 ** 24, 10 ** 22, 10 ** 26]
```

```
def epheperid_to_particle(ephemerid, index = 0):

    for i in range(len(mass_patterns)):
        match = re.search(mass_patterns[i], ephemerid)
        if match:
            scale = scales[i]
            break
    mass = float(match.group(1)) * scale

    position_pattern = r"X\s*=\s*([-+]?\d*\.?\d+(?:[Ee][-+]?\d+)?)\s+Y\s*=\s*([-
    match = re.findall(position_pattern, ephemerid)
    xyz = match[index]
    x = float(xyz[0])
    y = float(xyz[1])
    z = float(xyz[2])

    velocity_pattern = r"VX\s*=\s*([-+]?\d*\.?\d+(?:[Ee][-+]?\d+)?)\s+VY\s*=\s*(
    match = re.findall(position_pattern, ephemerid)
    xyz = match[index]
    vx = float(xyz[0])
    vy = float(xyz[1])
    vz = float(xyz[2])
    position = Vector(x, y, z)
    velocity = Vector(vx, vy, vz)
    return Particle(position, velocity, mass)

planet_names = ['Sun', 'Mercury', 'Venus', 'Mars', 'Jupiter', 'Saturn'
                , 'Uranus', 'Neptune']

planet_names = ['Planets/' + name + '.txt' for name in planet_names]
files = [ open(name, 'r') for name in planet_names]
planets = [file.read() for file in files]

start_planet_particles = [epheperid_to_particle(planet) for planet in
planets]
end_planet_particles = [epheperid_to_particle(planet, -1) for planet in
planets]
```

EphemeristestIntegrator tests the integrators using the ephemeris data and n-body force caluculation.

Dependencies: Core time Ephemeris

```
from Ephemeris import start_planet_particles, end_planet_particles
from Core import *
from time import time
```

```
integrator = Integrator()
integrator.switch("semi_implicit_euler")
force = Force()
force.switch("n_body")

dt = 100000
start_time = 0
integrator_names = ['euler_forwards','euler_back','semi_implicit_euler',
                                        'midpoint','rk4']
integrators = [ Integrator() for name in integrator_names]
for i in range(len(integrators)):
    integrators[i].switch(integrator_names[i])
month = 30 * 24 * 60 * 60

init_state  = SystemState(start_planet_particles, start_time, dt)
final_state_real = SystemState(end_planet_particles, start_time, dt)

refrence_system = System(init_state, integrators[2], force, 1000)
refrence_system.step_time(month)
final_state_ref = refrence_system.get_state(-1)


errors_real = []
errors_ref = []
times = []

for integrator in integrators:

    test_system = System(init_state, integrator, force, dt)

    start_time = time()
    test_system.step_time(month)
    end_time = time()
    time_difference = end_time - start_time
    times.append(time_difference)

    end_state = test_system.get_state(-1)
    error_real = end_state * final_state_real
    error_refrence = end_state * final_state_ref
    errors_real.append(error_real)
    errors_ref.append(error_refrence)
```

EphemerisIntegratorPlot.py is used to plot the Ephemeris tests. In particular figures 2, 3 and 4.

Dependencies: matplotlib EphermerisTestIntegrator

```
from matplotlib import pyplot as plt
from EphemerisTestIntegrator import integrator_names, errors_real, errors_ref, t

integrator_names = ['euler_forwards','euler_back','semi_implicit_euler',
                                        'midpoint','rk4']


plt.bar(integrator_names, errors_real)
plt.xlabel("Integration Method")
plt.ylabel("Average Error (metres)")
plt.title("Error From True Ephemerids Value For Different Integrators")
plt.show()

plt.bar(integrator_names, errors_ref)
plt.xlabel("Integration Method")
plt.ylabel("Average Error (metres)")
plt.title("Error From Refrence Simulation For Different Integrators")
plt.show()

plt.bar(integrator_names, times)
plt.xlabel("Integration Method")
plt.ylabel("Time To Complete Integration (seconds)")
plt.title("Duration For Different Integrators To Complete")
plt.show()
```

EphemerisTestTimestep is used to test how the time-step size change the accuracy of the simulation and create 1.

Dependencies: Core.py Ephemeris.py

```
  from Ephemeris import start_planet_particles, end_planet_particles
from Core import *


integrator = Integrator()
integrator.switch("semi_implicit_euler")
force = Force()
force.switch("n_body")

dt = 100
start_time = 0
test_dt = [ i * 2500 for i in range(1,50)]
month_time = 30 * 24 * 60 * 60

init_state  = SystemState(start_planet_particles, start_time, dt)
final_state = SystemState(end_planet_particles, start_time, dt)
```

```
end_state_ref = final_state

errors = []

for dt in test_dt:
    test_system = System(init_state, integrator, force, dt)
    test_system.step_time(month_time)
    end_state = test_system.get_state(-1)
    error = end_state * end_state_ref
    errors.append(error)


from math import log10, exp
from matplotlib import pyplot as plt

x = test_dt
y = errors

plt.plot(x, y)

plt.ylim(0, errors[-1] + 1e15)
plt.xlabel("Timestep (seconds)")
plt.ylabel("Average Error (metres)")
plt.title("Error From True Ephemerids Values For Different Timesteps")
plt.show()
```

Render.py is used to create visualisations of the simulation. This was used for debugging.

Dependencies: math matplotlib Core.py Ephemeris.py

```
  import graphics as gr
from time import sleep

from Core import *


window = gr.GraphWin("Simulation", 1000, 1000)

def display_state(state, window = window, scale = 10, color = 'white'):
    for particle in state.get_particles():
        position = particle.get_position()
        position.set_y(-position.get_y())
        position *= scale
        centre_screen = Vector(window.getWidth(), window.getHeight(), 0) / 2
        position += centre_screen
        x,y,z = position.transpose()
        point = gr.Point(int(x), int(y))
```

```
        circle = gr.Circle(point,5)
        circle.setFill(color)
        circle.draw(window)

def display_system(system, window = window, scale = 50, time_scale = 1, color =

    states = system.get_states()
    num_states = len(states)

    for i in range( num_states ):
        state = states[i]
        display_state(state, window, scale = scale, color = color)
        sleep(system.get_dt() * time_scale)
```

SurfaceIntegratortest.py is used to test the errors for different integrators in constant force fields.

Dependencies: Core.py matplotlib

```
from Core import *
from matplotlib import pyplot as plt


integrator_names = ['euler_forwards','euler_back','semi_implicit_euler',
                                        'midpoint','rk4']
integrators = [ Integrator() for name in integrator_names]
for i in range(len(integrators)):
    integrators[i].cycle(i)

force = Force()
force.switch("earth_gravity")

dt = 0.1
time = 1

particle_init = Particle(Vector(0,0,0), Vector(0,5, 0), 1)
state_init = SystemState([particle_init], 0, dt)


state_ref = SystemState( [Particle(Vector(0,0.095,0), Vector(0,0, 0), 1)] , 5, 0

errors = []
for integrator in integrators:
    system = System(state_init, integrator, force, dt)
    system.step(int(time / dt) + 1)
    final_state = system.get_state(-1)
    print(final_state)
    error = final_state * state_ref
```

```
    errors.append(error)

plt.bar(integrator_names, errors)
plt.xlabel("Integration Method")
plt.ylabel("Average Error (metres)")
plt.title("Error From Refrence Simulation Value For Different Integrators")
plt.show()
```