UNIVERSIDAD NACIONAL DE COLOMBIA

Astrofísica Computacional

Proyecto Final
Problema de los N cuerpos

Juan Sebastian Montoya Combita
Joar Esteban Buitrago Carrillo

4 de Agosto de 2021

1. Introducción

En el problema de los N cuerpos busca predecir los movimientos individuales de un determinado grupo de cuerpos celestes interactuando entre sí gravitacionalmente.
El problema de los N cuerpos clásico se plantea cómo: «Dadas las propiedades orbitales iniciales (masa, posición y velocidad iniciales) en un grupo de cuerpos celestes, se pueden determinar las fuerzas interactivas que actúan entre si, logrando calcular sus movimientos orbitales para cualquier instante».
El problema de N cuerpos considera N puntos de masas m_i para $i = 1, 2, \ldots, N$ en un sistema de referencia inercial en el espacio tridimensional $\mathbb{R}^3$ moviéndose bajo influencia gravitacional mutua. Al igualar la segunda ley de Newton con la ley de gravitación universal se obtendrá la siguiente ecuación:

$$m_i \ddot{x}_i = -G m_i \sum_{j=1, i\neq j}^{N} \frac{m_j}{|x_{ij}|^2} \tag{1}$$

Donde $x_{ij} = x_i - x_j$ es el vector relativo que apunta desde la partícula j a la partícula i.
Tenemos que a partir de la ecuación (1), se puede escribir el sistema como una ecuación diferencial de primer orden.

$$\begin{aligned} \frac{d\vec{x_i}}{dt} &= \vec{v_i} \\ \frac{d\vec{v_i}}{dt} &= -G \sum_{j=1, i\neq j}^{N} \frac{m_j}{|\vec{x_{ij}}|^3} \vec{x_{ij}} \end{aligned} \tag{2}$$

Con el fin de garantizar el procedimiento de integración, se introducirá la energía total del sistema para el problema de N cuerpos, la cual es una cantidad conservada.

1.1. Integración de Strömer-Verlet

El algoritmo de Strömer-Verlet es un procedimiento de integración para ecuaciones diferenciales ordinarias de segundo orden en donde se tienen los valores iniciales conocidos. Este algoritmo es muy útil en donde la expresión de la segunda derivada solo es función de las variables, es decir, no se requiere de la participación de la primera derivada.

En el caso del cálculo del problema de N cuerpos, podemos obtener la siguiente posición a partir de la posición actual y la aceleración sin necesidad de calcular la velocidad, lo cual será muy útil ya que se optimiza el algoritmo evitando cálculos extra.

La formula de integración de Strömer-Verlet sin velocidades es:

$$\begin{aligned} \vec{x}(t + \Delta t) &= 2\vec{x}(t) - \vec{x}(t - \Delta t) + \vec{a}(t)\Delta t^2 \\ \vec{x}_{n+1} &= 2\vec{x}_n - \vec{x}_{n-1} + \vec{a}_n \Delta t^2 \end{aligned} \tag{3}$$

Pero para el paso inicial del algoritmo se requiere conocer x_{-1} y x_0, pero solamente se conocen las condiciones iniciales x_0, v_0 y a_0, de manera que para el primer paso es necesario utilizar otro método. Utilizando el método de Euler, debido a la baja complejidad que ofrece, hallaremos el primer paso utilizando la siguiente formula:

$$\vec{x}_1 = \vec{x}_0 + \vec{v}_0\Delta t + \frac{1}{2}\vec{a}_0\Delta t^2 \tag{4}$$

1.2. Energía total del sistema

Para asegurar que el algoritmo tiene una buena precisión, se va a calcular la energía total del sistema, la cual es una cantidad conservada.

$$E = \frac{1}{2}\sum_{i=1}^{N} m_i|\vec{v}_i|^2 - \frac{1}{2}G\sum_{i=1}^{N}\sum_{j=1,i\neq j}^{N}\frac{m_i m_j}{r_{ij}} \tag{5}$$

2. Desarrollo del algoritmo

Representaremos las cantidades vectoriales como arreglos usando la librería NumPy. Utilizando las diferentes operaciones aritméticas incluidas en la librería, con el fin de tener una mayor optimización en las operaciones y reducir el tiempo de ejecución del algoritmo.[5][4]
Debido a que los valores de masa, velocidad y distancia en astronomía son muy grandes, llevarlos en el Sistema Internacional de Unidades (kg, m, s) obtendríamos valores demasiado grandes. Para poder dar un mejor tratamiento a los valores, se requieren normalizarlos en términos de:

$$\begin{aligned} x &\Rightarrow \text{au [Distancia (Unidades Astronómicas)]} \\ t &\Rightarrow \text{year [Tiempo (años)]} \\ \dot{x} = v &= \frac{\text{au}}{\text{year}} \text{ [Velocidad (Distancia/Tiempo)]} \\ \text{mass} &= \text{M}_\odot \text{ Masa (Masas solares)} \end{aligned}$$

Para poder implementar dichas unidades de manera eficiente en el algoritmo, se decidió usar la librería **Astropy** para el manejo de unidades y así como también realizar las conversiones de unidades mediante la función `.to`. Otra ventaja que se tiene al usar las variables con sistema de `astropy`. (valor, unidad de medida) es que permite garantizar que las funciones de calculo sean correctas mediante el análisis dimensional. Así como también nos permite disponer de manera sencilla de las constantes como lo es la constante de gravitación universal mediante la libreria `astropy.constants`.[1][2][3]
Una vez se normalicen las unidades, se procede a plantear la ODE (Ecuación Diferencial Ordinaria) para el problema. La cual está expresada en la ecuación (2). Así pues, la ODE estará cómo una función implementada en el código de la manera:

```
def acc(self, mass, distance):
  up = mass * distance * const.G
  down =  np.power(np.linalg.norm(distance), 3)
  return - (up / down)
```

Donde *mass* corresponde a la masa que el objeto actual esta orbitando m_j, *distance* al vector relativo de posición x_{ij}. *up* corresponde a la operación del numerador en la ecuación (2) y *down* a la operación en el denominador de la ecuación (2).
Ahora, una vez puesta la ODE, se procede a determinar el método de integración, en este caso, el método de Strömer-Verlet debido a su precisión y eficiencia. Dicha implementación se encuentra dentro de la función denominada **add_interaction** para calcular la aceleración. Es importante recalcar que está función únicamente añadirá la interacción con otro cuerpo, es decir, que para un sistema de N cuerpos, se requiere llamar la función $N-1$ veces para actualizar el comportamiento de un cuerpo con respecto a todos los demás cuerpos.

```
def add_interaction(self, other):
  distance = (self.pos - other.pos).to(u.au)
  self._acc += self.acc(other.mass, distance)
```

Donde *other* será la información de la posición y masa del otro cuerpo.
Finalmente, con los datos hallados se busca actualizar la información de la posición del objeto para almacenarlo en la siguiente iteración. Éste paso será implementado dentro de la función **step**.

```
def step(self, dt, prev = None):
  if prev == None:
```

```
      #self.pos = self.pos + self.vel * dt + 1/2 * self._acc * dt ** 2
      self.pos = np.add(
                  np.add(self.pos, np.multiply(self.vel, dt)),
                  np.multiply(0.5*np.power(dt,2), self._acc)
                 )
    else:
      #self.pos = 2 * self.pos - prev.pos + self._acc * dt ** 2
      self.pos = np.add(
      np.subtract(
        np.multiply(2, self.pos), prev.pos),
        np.multiply(np.power(dt,2), self._acc)
      )
    self._clear()
```

Hay que tener en cuenta que al final es importante reiniciar los valores de `self._acc` en 0. Lo cual hace la función `self._clear`. Cómo se mencionó anteriormente, al tener un problema de N cuerpos se requiere analizar todas las iteraciones gravitaciones que hay entre los cuerpos. Para ello y de una manera mucho más intuitiva, se crea un objeto **Astrobject** el cual guarda todas las funciones anteriormente mostradas. Cada objeto tendrá como atributos iniciales de entrada los datos de posición, velocidad y masa con las unidades normalizadas.

```
class Astrobject:

  @u.quantity_input
  def __init__(
      self,
      pos: u.au = np.zeros(3) * u.au,
      vel: u.au/u.year = np.zeros(3) * u.au/ u.year,
      mass: u.M_sun = 1. * u.M_sun
  ):
    """Constructor method."""
    self.pos = pos
    self.vel = vel
    assert(mass > 0)
    self.mass = mass
    self._clear()
```

En caso de no ingresar ningún atributo de entrada, asumirá valores por defecto de posición y velocidad iguales a cero, con una masa solar. Notesé también que no pueden existir masas negativas, así que si el usuario llega a introducir una masa negativa, el programa arrojará un error de aserción.
Con el objetivo de que el programa permita al usuario manejar un número arbitrario de cuerpos, entonces se creará una clase **Astrosystem** la cual almacena un arreglo de `Astrobject`, guardará la información de cada cuerpo y inicializará todos los objetos con sus respectivos valores de posición, velocidad y masa normalizados a partir de los archivos importados.

```
class Astrosystem:
  @u.quantity_input
  def __init__(
      self,
      pos_unit: u.au = 1.*u.au,
      vel_unit: u.au/u.year = 1.*u.au/u.year,
      mass_unit: u.M_sun = 1.*u.M_sun
  ):
    """Constructor method."""
    self.pos_unit = pos_unit
    self.vel_unit = vel_unit
    self.mass_unit = mass_unit

    self.data = np.array(())

  def loadtxt(self, *args, **kwargs):
    self.data = np.loadtxt(*args, **kwargs)
```

```
    self.objects = np.array([
      Astrobject(
        (row[0:3] * self.pos_unit).to(u.au),
        (row[3:6] * self.vel_unit).to(u.au/u.year),
        (row[6] * self.mass_unit).to(u.M_sun)
      )
        for row in self.data
    ])
```

La función `loadtxt` carga los datos enviando cómo parámetros el nombre del archivo, para que funcione con cualquier archivo se requiere que el formato sea en CSV y que los datos de cada cuerpo por fila sea el siguiente: Posición(x,y,z), Velocidad(x,y,z), y la masa.
Una vez se carguen los datos y se guarden como objetos, el objetivo es aplicar las funciones para que calculen los datos de posición, para cada intervalo de tiempo para todos los cuerpos. Esta parte requiere que para cada cuerpo, debe llamar la función `add_iteration` y `step` para cada cuerpo presente en el arreglo con excepción de él mismo. Después debe almacenar los nuevos cálculos en cada objeto y avanzar un paso de tiempo. Dicha función se llama **advance_time** la cual está dentro de la clase *Astrosystem*

```
  @u.quantity_input
  def advance_time(
      self,
      total_time: u.year,
      delta_time: u.year
  ):
    time = int ((total_time.to(delta_time.unit)) / delta_time)

    self.time_objects = np.empty((time, self.size), dtype=Astrobject)

    for t in range(time):
      for object_a in self.objects:
        for object_b in self.objects:
          if object_a is object_b:
            continue

          object_a.add_interaction(object_b)

      time_objects = np.array([], dtype=Astrobject)

      for index in range(self.size):
        body = self.objects[index]

        time_objects = np.append(time_objects, deepcopy(body))

        if t == 0:
          prev_body = None
        else:
          prev_body = self.time_objects[t-1, index]

        body.step(delta_time, prev_body)

      self.time_objects[t] = time_objects

    self.calc_vel(delta_time)
```

Los valores que calcula la función serán almacenados en *time_objects* que es un arreglo que almacena la información de todos los objetos para cada instante de tiempo.
Finalmente se realiza el calculo de la energía total del sistema, para ello se requiere aplicar la ecuación (5). Cómo podemos ver se requiere saber la velocidad de cada objeto en cada instante de tiempo y cómo se usó el algoritmo de Stömer-Verlet sin velocidades no se ha calculado esta variable. Si tenemos un Δt lo suficientemente pequeño, es posible obtener un buen valor para la velocidad

calculando la *Velocidad media* para cada instante de tiempo.

$$v_i = \frac{x_{i+1} - x_{i-1}}{t_{i+1} - t_{i-1}} + \mathcal{O}(\Delta t^2) \tag{6}$$

En donde v_0 será equivalente al dato importado desde el archivo y v_N será igual a:

$$v_N = \frac{x_N - x_{N-1}}{t_N - t_{N-1}} + \mathcal{O}(\Delta t) \tag{7}$$

Para calcular la velocidad en cada objeto se requiere llamar el arreglo de objetos y calcular la velocidad de cada uno para cada instante de tiempo, para ello se implementa la siguiente función **calc_vel** dentro de la clase Astrosystem.

```
def calc_vel(
    self,
    dt
):
  for time in range(1, self.time_objects.shape[0]):
    time_objects = self.time_objects[time]

    for index in range(time_objects.shape[0]):
      body = time_objects[index]

      if time == (self.time_objects.shape[0] - 1):

        # body.vel = self.time_objects[time, index].pos - self.time_objects[time - 1, index].pos
        body.vel = np.subtract(
                    self.time_objects[time, index].pos,
                    self.time_objects[time - 1, index].pos
                  )
        # body.vel /= dt
        body.vel = np.divide(body.vel, dt)

      else:
        #body.vel = self.time_objects[time + 1, index].pos - self.time_objects[time - 1, index].pos
        body.vel = np.subtract(
                    self.time_objects[time + 1 , index].pos,
                    self.time_objects[time - 1, index].pos
                  )
        #body.vel /= 2 * dt
        body.vel = np.divide(body.vel, 2*dt)
```

La función está compuesta por dos ciclos, el primer ciclo es aquel en el cual se recorre cada instante de tiempo y el segundo es donde se calcula la velocidad para cada objeto.
Una vez obtenido el valor de la velocidad para cada objeto en cada instante de tiempo, se procede a calcular la energía total del sistema para cada instante de tiempo mediante la función **energy**

```
@property
def energy(self):
  energies = np.array([]) * u.J

  def kinetic(body):
    vel = np.linalg.norm(body.vel)
    return 0.5 * body.mass * vel ** 2

  def potential(object_a, object_b):
    dis = np.linalg.norm(object_a.pos - object_b.pos)
    return const.G * object_a.mass * object_b.mass / dis

  for time in self.time_objects:
    energy = 0
```

```
      for object_a in time:
        energy += kinetic(object_a)

        for object_b in time:

          # Skip same object
          if object_a is object_b:
            continue
          energy += potential(object_a, object_b)

      energies = np.append(energies, energy)

    return energies
```

La función define dos subfunciones, aquella que calcula la energía cinética, y la otra calcula la energía potencial gravitacional. El calculo de la energía total requiere un total de 3 ciclos para calcular, en la primera de recorre cada instante de tiempo, en el segundo ciclo se obtiene cada objeto y en el último ciclo se obtiene nuevamente todos los objetos menos para él mismo y posteriormente hace el calculo de la energía cinética y potencial, estás energías se van sumando y finalmente cuando se recorre todos los objetos se guarda en un arreglo llamado *energies* la cual finalmente retorna con el dato de la energía total del sistema para cada tiempo.

3. Implementación

3.1. Sistema Sol-Tierra

Para probar el algoritmo, se tienen los datos iniciales con respecto al sistema Sol-Tierra almacenados en un archivo llamado *sun_earth.csv*.
Primero, se crea el sistema llamando la clase Astrosystem, en donde se define las unidades de medida que se tienen en los datos del archivo, posteriormente se guardan los datos mediante la función de *NumPy.loadtxt*. Para comprobar si la carga fue exitosa se llama la clase, la cual sebe especificar cuantos objetos tiene la clase Astrosystem.

```
sun_earth = Astrosystem(1.*u.m, 1.*u.m/u.s, 1.*u.kg)
sun_earth.loadtxt("sun_earth.csv", delimiter=",")

sun_earth
```

Se espera que retorne lo siguiente "**Astronomical System with 2 objects**".

Ahora una vez cargados los datos iniciales y definidos los objetos, se procederá a calcular la trayectoria de los cuerpos, para ello se llama al método *advance_time* los cuales recibirá el parámetro de la cantidad de tiempo t y el paso de tiempo dt definiendo sus unidades de medida con $u.unit$

```
sun_earth.advance_time(5.*u.year, 1.*u.day)
```

Una vez se halla realizado la operación, se grafican los resultados obteniendo las trayectorias del sistema.

```
sun_earth.show()
```

La cual generará una gráfica cómo esta.

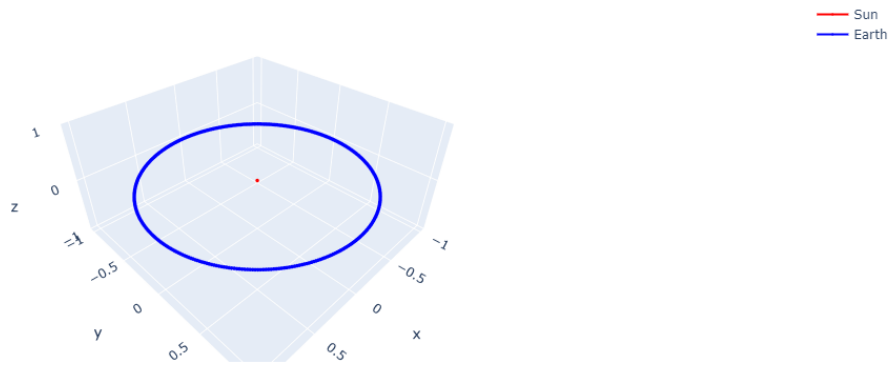

Figura 1: Dibujo Sol Tierra.

Ahora con el fin de asegurar la convergencia del algoritmo, se hacen diferentes gráficas de la energía total con respecto a diferentes pasos de integración, en este caso se escogieron dt de 1, 5, 15, 30 y 60 días. Para calcular la energía se requiere hacer el calculo de la trayectoria con el dt y llamar el método *show_energy*

```
sun_earth.loadtxt("sun_earth.csv", delimiter=",")
sun_earth.advance_time(5.*u.year, dt.*u.day)
sun_earth.show_energy()

np.std(sun_earth.energy)

max_energy = np.max(sun_earth.energy)
min_energy = np.min(sun_earth.energy)

deviation = (max_energy - min_energy) / min_energy * 100
```

Donde además de graficar la energía se muestra la cantidad de desviación de al energía tanto su cantidad como su relación porcentual.

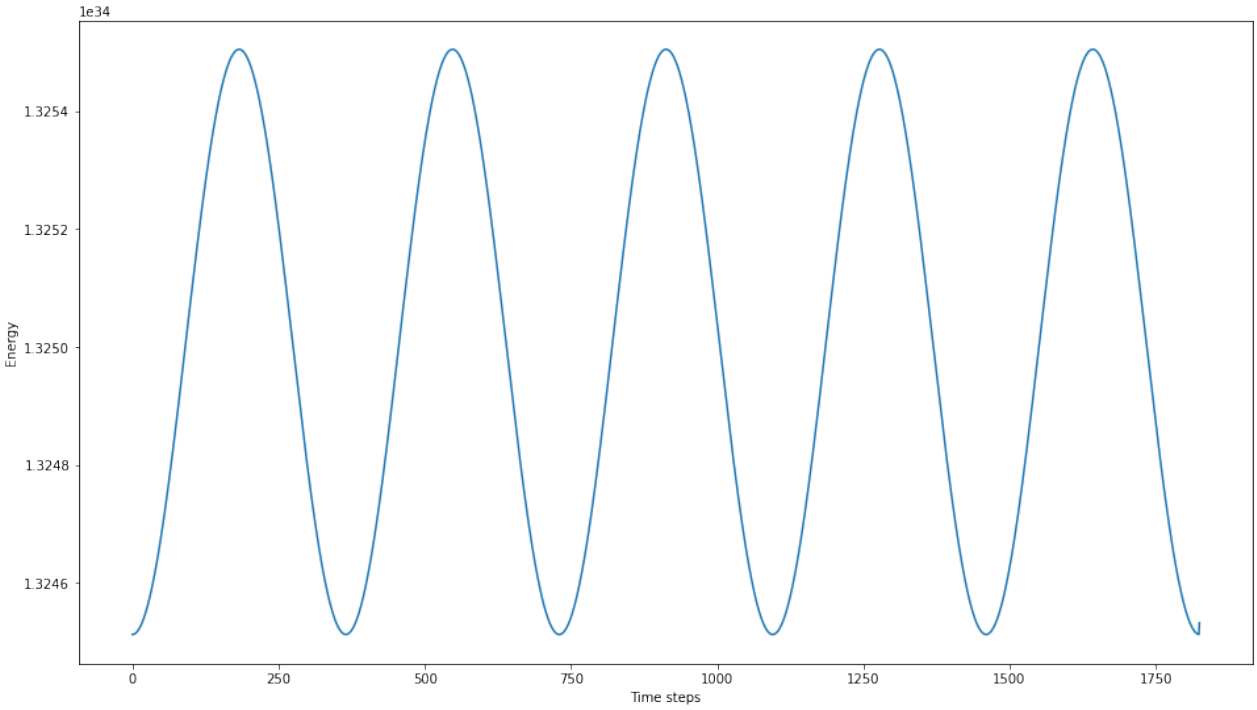

Figura 2: Energía total con dt de 1 día.

Usando un dt de un día, se tendrá una desviación de la energía con un valor de $3.506 \times 10^{30} J$ y la desviación relativa de la energía será del $0.074\,\%$. Se puede apreciar como la energía se conserva y compensa.

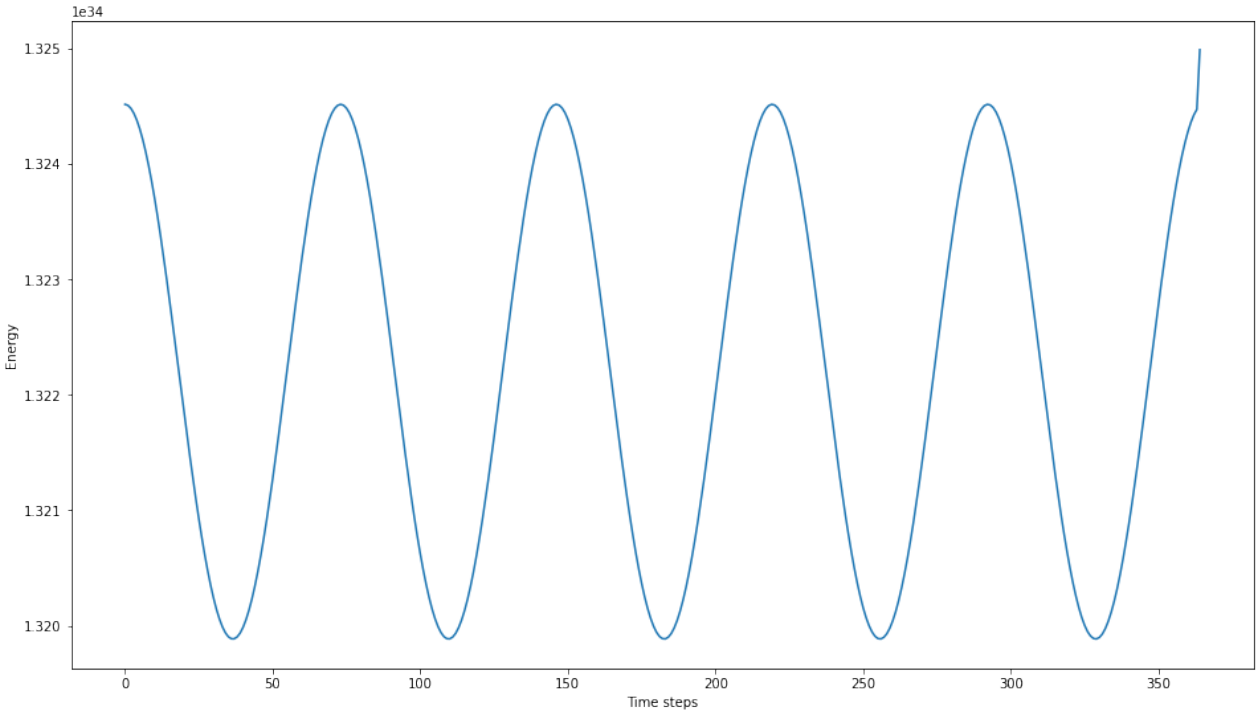

Figura 3: Energía total con dt de 5 días.

Usando un dt de 5 días, se tendrá una desviación de la energía con un valor de $1.637 \times 10^{31} J$ y la desviación relativa de la energía será del $0.386\,\%$. Se puede ver que la variación de la energía sigue siendo mínima aunque ya se empieza a ver ciertas asimetrías al final de la gráfica.

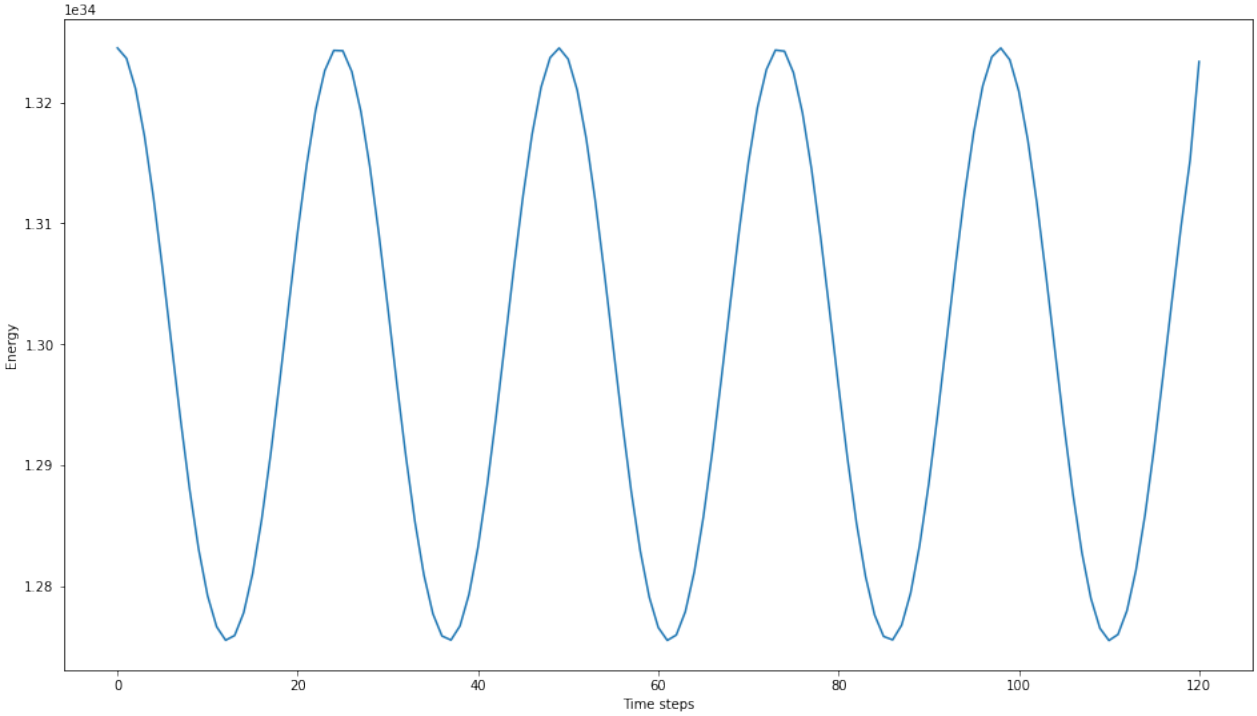

Figura 4: Energía total con dt de 15 días.

Usando un dt de 15 días, se tendrá una desviación de la energía con un valor de $1.729 \times 10^{32} J$ y la desviación relativa de la energía será del $3.844\,\%$. Se puede ver ya una notable variación de la energía, ya que se ve que su valor aumenta a medida que aumenta el dt. le desviación relativa ya empieza a ser considerable. Sin embargo aún trata de mantener la convergencia.

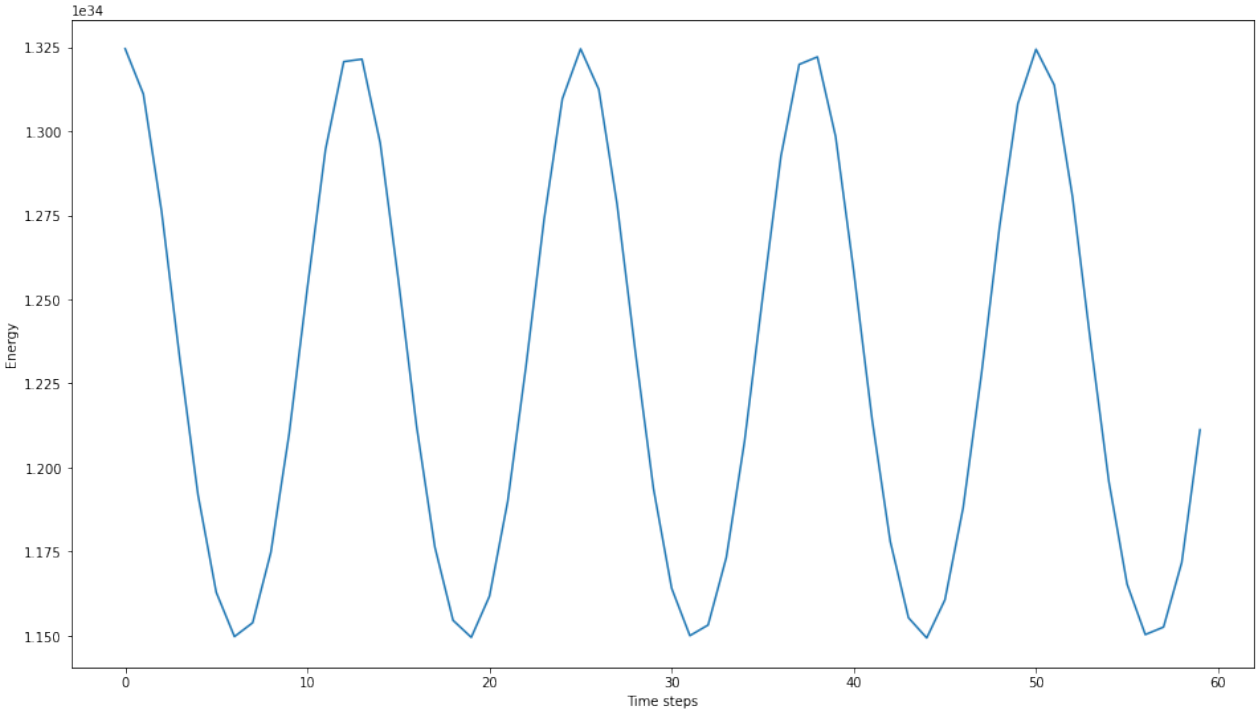

Figura 5: Energía total con dt de 30 días.

Usando un dt de 30 días, se tendrá una desviación de la energía con un valor de $6.204 \times 10^{32} J$ y la desviación relativa de la energía será del $15.238\,\%$. Se puede ver ya una notable variación de la energía, el intervalo de valores entre cada ciclo es grande. le desviación relativa es considerable.

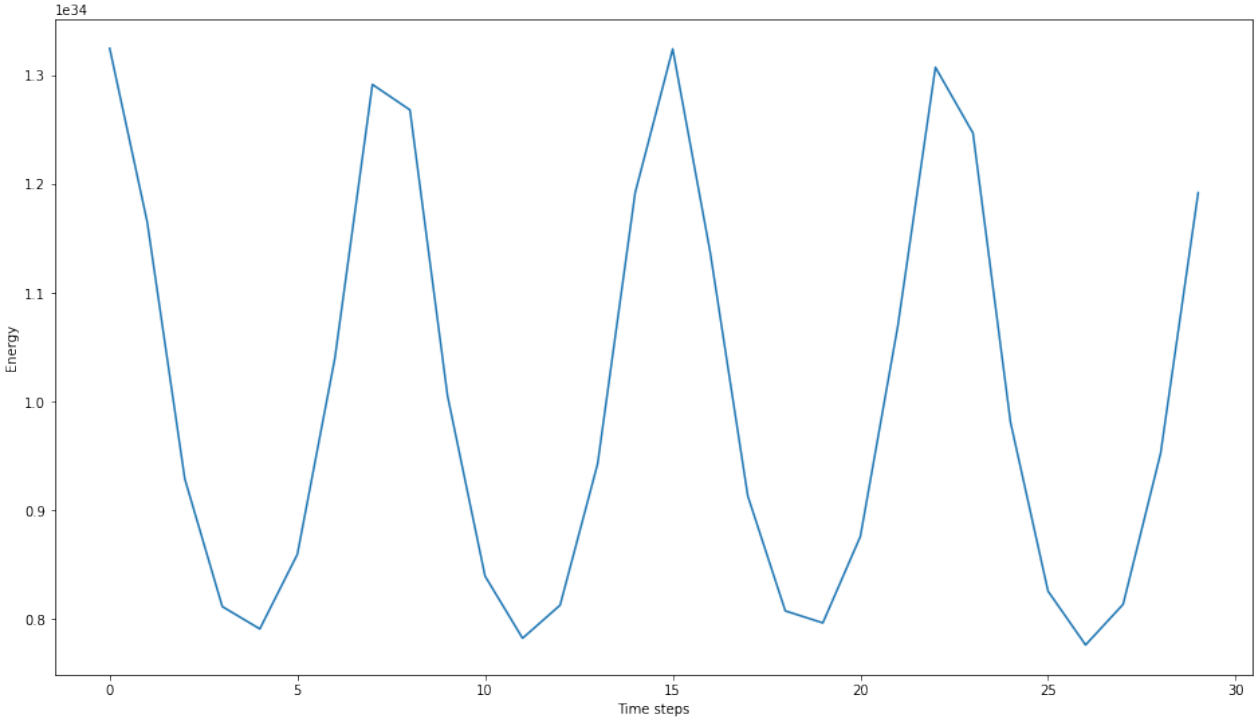

Figura 6: Energía total con dt de 60 días.

Usando un dt de 60 días, se tendrá una desviación de la energía con un valor de $1.899 \times 10^{33} J$ y la desviación relativa de la energía será del $70.624\,\%$. El valor de la energía es bastante impreciso.
Graficando cómo sería la trayectoria con este valor de dt se vería así:

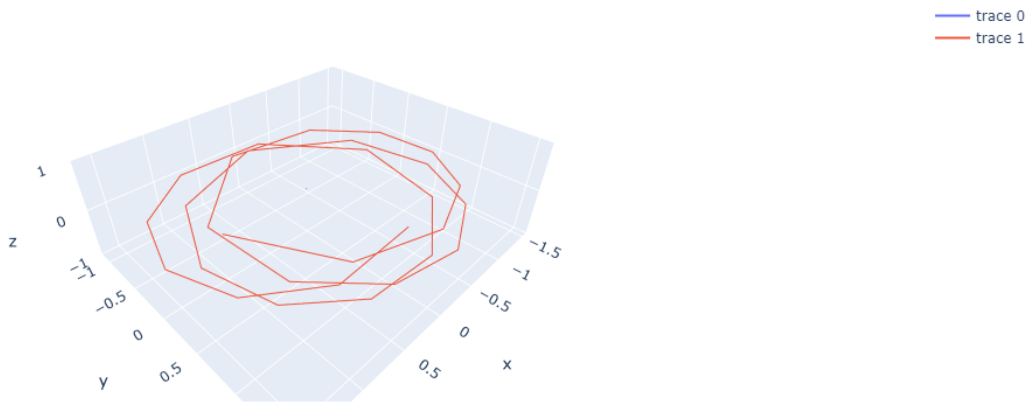

Figura 7: Sistema tierra sol calculado con un dt de 60 días.

Cómo se puede ver, se tendrá una orbita con trayectorias rectilíneas las cuales no tienden a ser una orbita estable.

3.2. Sistema Sgr A* con 13 Estrellas

Ahora se implementará un sistema de 14 cuerpos, lo cual corresponde a los datos de 13 estrellas orbitando alrededor del súper agujero negro ubicado en el centro de la Vía Lactea conocido como Sagitario A*.

Para poder calcula la trayectoria de este sistema primero se implementan las unidades de medida en formato de unidades de Astropy y se inicializa la clase Astrosystem; se cargan los datos con las condiciones iniciales de cada objeto.

```
distance = np.tan(1*u.arcsec)*8*u.kpc

sgrAstar = Astrosystem(distance, distance/u.year, 1.*u.M_sun)
sgrAstar.loadtxt("sgrAstar.csv", delimiter=",")

sgrAstar
```

Se espera que retorne el siguiente texto **Astronomical System with 14 objects**. Finalmente se procede a calcular la trayectoria de cada objeto por un periodo de 100 años con un dt de 5 días. También se medirá cuanto tiempo demora en realizar todos los cálculos usando *%%time*.

```
%%time
sgrAstar.advance_time(100.*u.year, 5.*u.day)
```

El calculo toma un tiempo probado y medido en Google Colab de 19 minutos aproximadamente. Finalmente la gráfica del sistema es la siguiente:

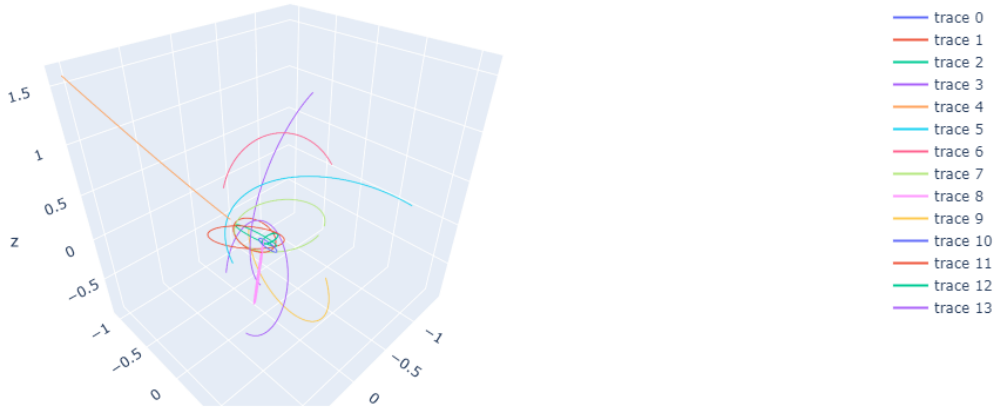

Figura 8: Trayectoria sistema Sgr A* con 13 Estrellas por 100 años.

Cómo podemos ver, algunas estrellas no alcanzan a completar una orbita y parece que un objeto tendiera a escapar del sistema. Ahora se procede a comprobar la convergencia del algoritmo calculando la energía total del sistema.

```
%%time
sgrAstar.show_energy()
```

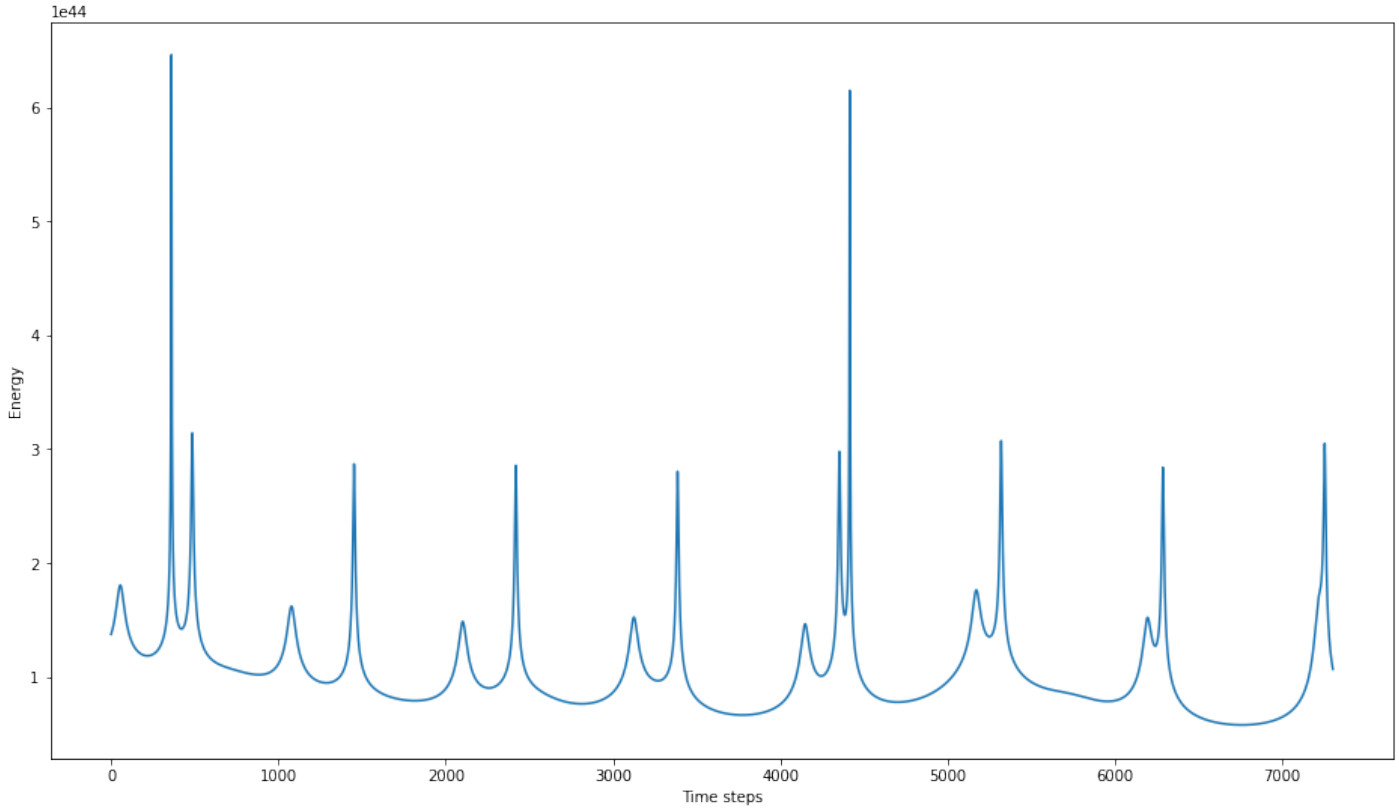

Figura 9: Gráfica Energía total del sistema Sgr A* con 13 Estrellas por 100 años.

Se puede apreciar un comportamiento estable de la energía pero diferente comparado con las anteriores gráficas, esto puede darse debido al acercamiento de las estrellas al agujero negro y las orbitas incompletas de las otras estrellas.
Se desea poder completar las orbitas de las estrellas y así asegurarnos que el algoritmo calcula bien y no hay objetos que escapen del sistema o presenten novedades, para ello se amplia el tiempo de calculo a 500 años conservando el mismo dt. El tiempo de ejecución fue aproximadamente de 1 hora con 30 minutos. El resultado de la trayectoria es:

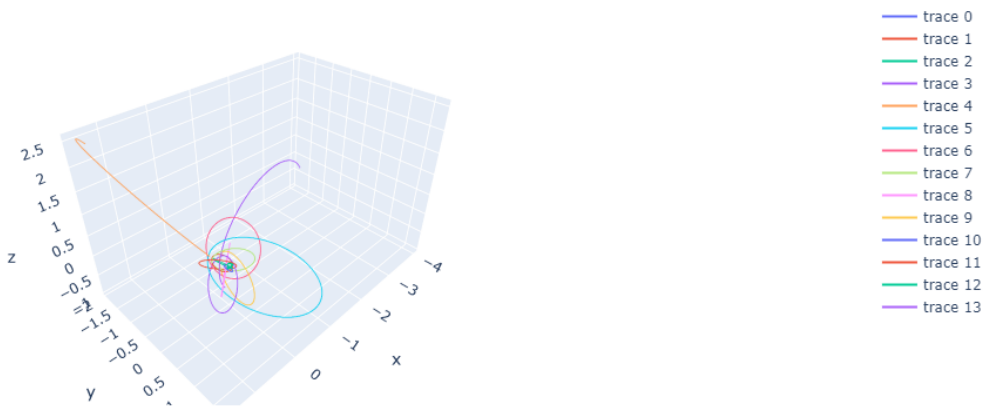

Figura 10: Trayectoria del sistema Sgr A* con 13 Estrellas por 500 años.

Cómo se puede apreciar, algunas estrellas completan sus trayectorias mientras que otras no lo hacen, probablemente se requerirá mucho más tiempo pero se ve que el comportamiento de la orbita es continuo y esperado, con respecto al objeto alejado que parecía escapar en la anterior gráfica, ahora parece empezar la trayectoria de regreso concluyendo que es una órbita demasiado hiperbólica.
Nuevamente se calcula la energía total del sistema:

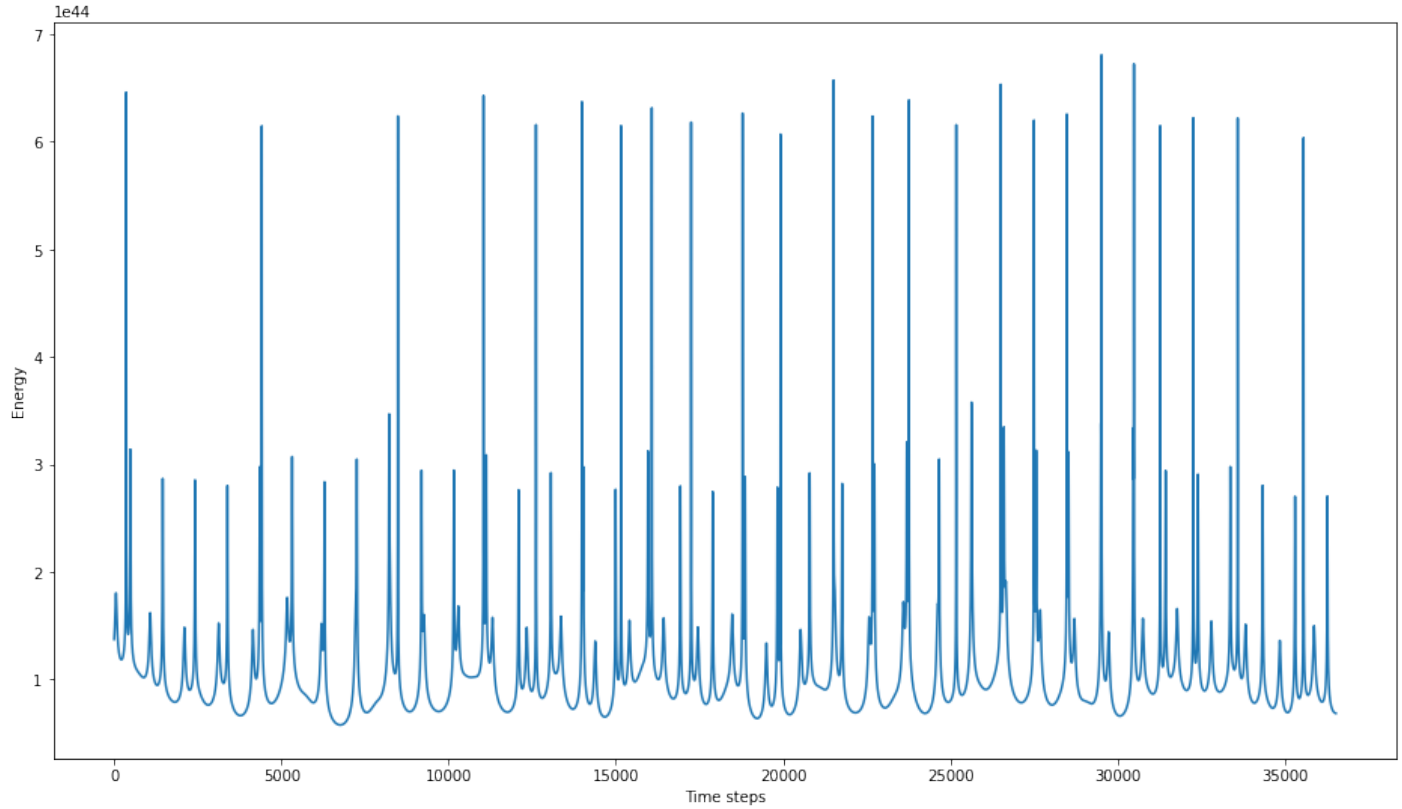

Figura 11: Gráfica Energía total del sistema Sgr A* con 13 Estrellas por 500 años.

El gráfico de la energía sigue el patrón observado con la anterior, no hay mayor variación en la energía, lo cual indica que el algoritmo no está convergiendo.

Conclusiones

1. El método de integración de Stömer-Verlet tiene una mayor eficiencia computacional, ya que no requiere realizar calculo de primera derivada ni hacer más cálculos a diferencia de RK4, los tiempos evaluados al comparar Verlet con RK4 se obtuvo una reducción en tiempos de ejecución hasta del $60\,\%$.

2. El algoritmo de Strömer-Verlet muestra una alta precisión y mínimo riesgo de convergencia ante valores adecuados de dt, esto se pudo ver mediante de calculo de variables conservadas cómo la energía total del sistema.

3. Implementar el algoritmo mediante clases brinda una manera mucho más organizada y eficiente de poder generalizar el código para que pueda aplicarse a cualquier sistema

4. Si bien el uso de Astropy pueda afectar ligeramente el rendimiento, el aprovechamiento del mismo para comprobar sistemas de unidades es una herramienta muy útil la cual permite asegurar que los cálculos realizados son correctos en el sentido de análisis dimensional y además, permite que acepte como entrada los datos en diferentes unidades de medida sin tener que realizar adiciones o más funciones de conversión.

5. Para sistemas con un considerable número de cuerpos y además con periodos de tiempo grandes, requerirán de un tiempo considerable de calculo. Una heurística para optimizar el calculo sería ignorar aquellas interacciones entre objetos la cual sea mínima o inapreciable, este factor puede considerarse y aplicarse para futuras actualizaciones en el código.

6. El uso de Mecánica celeste clásica para este tipo de problemas tiene una buena aproximación y precisión, siendo útil para muchos aún hoy en la actualidad para el cálculo de trayectorias.

7. Usar operaciones de NumPy entre los arreglos en vez de usar los operadores por defecto en Python, mostraron obtener una muy ligera mejora en tiempos de ejecución reduciendo hasta un $5\,\%$

Referencias

[1] Comunidad de desarrolladores de AstroPy. *AstroPy Reference*. en. Ver. 4.3.1. AstroPy Project. 11 de ago. de 2021. URL: https://docs.astropy.org/en/stable/ (visitado 12-08-2021).

[2] Astropy Collaboration y col. «Astropy: A community Python package for astronomy». En: *Astronomy & Astrophysics* 558, A33 (oct. de 2013), pág. 10. DOI: 10.1051/0004-6361/201322068. arXiv: 1307.6212 [astro-ph.IM].

[3] Astropy Collaboration y col. «The Astropy Project: Building an Open-science Project and Status of the v2.0 Core Package». En: *The Astronomical Journal* 156.3, 123 (sep. de 2018), pág. 39. DOI: 10.3847/1538-3881/aabc4f. arXiv: 1801.02634 [astro-ph.IM].

[4] Charles R. Harris y col. «Array programming with NumPy». En: *Nature* 585.7825 (sep. de 2020), págs. 357-362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[5] Comunidad de desarrolladores de NumPy. *NumPy Reference*. en. Ver. 1.21. NumPy. 22 de jun. de 2021. URL: https://numpy.org/doc/1.21/reference/ (visitado 05-08-2021).