Informe Tarea 4 - Métodos Numéricos: "Cálculo de órbitas planetarias relativistas"

Ignacio Andrés Sánchez Barraza
Rut: 18933808-2

October 17, 2015

1 Pregunta 1

1.1 Introducción

- En esta pregunta se pide responder a preguntas teóricas sobre el método propuesto por SOftware Carpentry para mejorar las habilidades al desarollar programas y optimizar su diseño.

1.1.1 Preguntas

- Describa la idea de escribir el main driver primero y llenar los huecos luego. ¿Por qué es buena idea?

  R: La idea de escribir primero el main driver es dar la estructura al programa para darle un orden y secuencia, definiendo las funciones a usar de manera independiente, y así, rellenar luego los codigos faltantes a las demas partes del programa. Esta es una buena idea debido a que se vuelve mas fácil la tarea de optimizar el programa o encontrar errores o bugs en el código en el futuro y asi poder trabajar con trozos del programa y no con el código por completo.

- ¿Cuál es la idea detrás de la función $mark\_filled$? ¿Por qué es buena idea crearla en vez del código original al que reemplaza?

  R: La idea detras de esta función es la de reemplazar el error que detecta python al colocar indices que no estan dentro de una lista. Hay tres razones para esto. La primera es que cumple el rol de dejar de mejor forma que un comentario el valor que se espera para los indices. La segunda es que el mensaje de error que se crea tiene más significado y contenido que el $IndexError$ de python que sólo arroja dicho error para algún índice pero no explicita para cual (x o y). Y finalmente, la tercera razon es que con esta función es posible encontrar indices que podrían haber sido omitidos creyendo que no son posibles en el programa, y que al ejecutarlos si funcionan, creando errores o resultados inesperados.

- ¿Qué es $refactoring$?

  R: Es un método para verificar y validar si el código escrito esta libre de errores y/o si sirve para resolver el problema propuesto o el modelo usado es el correcto.
  Lo primero entonces, es ver si el programa puede ser probado en distintas condicione para encontrar errores. En esto, los casos mas difíciles son aquellos que incluyen escenarios que trabajan con aleatoriedad, donde para ello, se procede a probar dicho programa con una suerte de aleatoriedad "controlada" en la cual se sabe el resultado esperado, y se pueden ver fácilmente los errores si los hay.
  Luego de analizar dichos casos, lo que sigue consiste en reorganizar y delegar tareas a funciones especificas para evitar errores en distintas partes del programa. En síntesis, el $refactoring$ consiste en dividir el programa en funciones o partes fáciles de analizar y que realicen tareas específicas para facilitar el hallazgo de errores y su reparación.

- ¿Por qué es importante implementar tests que sean sencillos de escribir? ¿Cuál es la estrategia usada en el tutorial?

R:La razón de implementar pruebas que sean fáciles de escribir es para poder realizar pruebas variadas y poder ver y entender el resultado de dicha prueba de manera fácil y rápida, para detectar así errores y bugs en el programa.
La estrategia usada en el tutorial es la de "ver", es decir, obtener una interfaz gráfica del problema y lo que está haciendo el programa, de manera que sea entendible de manera fácil para el usuario y poder detectar rápidamente los errores si los hay. Esto sumado a una forma sencilla y rápida de escribir tests en el programa, que en el tutorial toma la forma de comandos asociados a strings entregados por el usuario al programa que luego retornan la interfaz gráfica para su análisis.

- El tutorial habla de dos grandes ideas para optimizar programas, ¿cuáles son esas ideas? Descríbalas.

  R:La primera de ellas es la llamada *AsymptoticAnalysis* que corresponde a atacar el problema que se quiere resolver de manera asintótica, valga la redundancia. Esto quiere decir que se busca reducir la cantidad de variables que toma el programa para procesar, disminuyendo el tiempo de procesamiento de información dado que la cantidad de datos es menor. Más que nada, consiste en actuar de forma recursiva, utilizando los datos de procesos anteriores para poder seguir con los futuros, almacenando menos información y ahorrando tiempo al procesar distintos casos a partir de otros anteriorres, sin tener que recorrer todo el camino desde cero.
  La segunda idea es la del *BynarySearch*. Esta consiste en un método de bisección, que consta de reducir el número de variables a procesar, dividiendo en dos la cantidad total de datos, segun determinados parámetros de búsqueda. Es un método recursivo, que toma datos, los divide a la mitad, procesa una mitad y luego la divide nuevamente repetir el procesos hasta obtener el resultado esperado (como ejemplo está el método de bisección para buscar raíces o ceros de una función. El cual toma dos puntos el dominio de la función para los que la función posee signos contrarios. La función luego toma el punto medio entre ellos. Si la función evaluada en ese valor posee signo contrario a la evaluada en los puntos anteriores, elige el punto medio entre los dos puntos que hacen que los signos sean distintos y repite el proceso). Con este método, el tiempo de procesamiento disminuye enormemente comparado con el de buscar iterativamente el resultado esperado (en el caso de los ceros de una función, evaluar iterativamente la función hasta encontrar que se hace cero).

- ¿Qué es *lazy evaluation*?

  R: Corresponde a evaluar o realiar un procesamiento de datos, sólo cuando es requerido y no previamente. En el caso del tutorial, se realizar mediante la creación de un diccionario o stack, donde se utiliza como coordena inicial la casilla en verde, y luego se crea si no existen en el diccionario, las demas casillas requeridas, y asi sucesivamente, para disminuir la cantidad de datos al comienzo y la creación innecesaria de datos que podrían no ser usados en los procesos siguientes (en el caso del tutorial, las casillas rojas).

- Describa la *other moral* del tutorial (es una de las más importantes a la hora de escribir un buen código).

  R: La *other moral* del tutorial u otra moraleja del tutorial es que para escribir un programa que sea rápido, primero hay que escribir uno que sea simple. Luego de probarlo y ver que funciona, se puede ir mejorando parte por parte para hacerlo más rapido, probando, en cada una de esas partes, el programa para mejorar su rapidez y funcionamiento.

2 Pregunta 2

2.1 Pregunta 2.1

2.1.1 Introducción

- Para planetas que orbitan cerca del Sol, el potencial gravitatorio relativista puede ser escrito como:

$$U(r) = -\frac{GMm}{r} + \alpha\frac{GMm}{r^2}$$

con α un número pequeño- Con α distinto de cero las órbitas siguen siendo planas, pero ya no son cerradas, sino que precesan, es decir el afelio (punto más lejano de la órbita con respecto al sol) se mueve alrededor del Sol. Dicho esto se pide implementar una clase $Planeta(condicion_inicial, \alpha)$ que utilice tres métodos numéricos (Euler explícito, Runge-Kutta orden 4 y Verlet) para calcular la órbita y energía de un planeta dado, con condiciones iniciales determinadas.

2.1.2 Procedimiento

- Para esto, se procede primero a crear dos funciones $f_x = F_x/m = \frac{d^2x}{dt^2}/m = -\frac{dU}{dx}/m = -\frac{dU}{dr}\frac{dr}{dx}/m$ y $f_y = F_y/m = \frac{d^2y}{dt^2}/m = -\frac{dU}{dy}/m = -\frac{dU}{dr}\frac{dr}{dy}/m$, con $r^2 = x^2 + y^2$, ie, las aceleraciones, que describirán las ecuaciones de movimiento en x e y respectivamente. Dichas funciones entonces quedan la forma siguiente:

$$f_x = \frac{2\alpha GMx}{(x^2+y^2)^2} - \frac{GMx}{(x^2+y^2)^{3/2}}$$
$$f_y = \frac{2\alpha GMy}{(x^2+y^2)^2} - \frac{GMy}{(x^2+y^2)^{3/2}}$$

Ahora para comenzar la clase se definió la funcion inicialización:

```
def __init__(self, condicion_inicial, alpha=0):
        self.y_actual = condicion_inicial
        self.t_actual = 0
        self.alpha = alpha
```

Que redefine la variable $self.y_actual$ como la condicón inicial (una tupla de 4 variables $(x_0, y_0, v_{x_0}, v_{y_0})$) entregada a la clase, $self.t_actual$ como el tiempo actual y $self.alpha$ como el parámetro α de la ecuación de potencial gravitatorio relativista. Dado esto entonces, se define la función $ecs_de_movimiento()$ como:

```
def ecuacion_de_movimiento(self):
        fx=lambda x,y,t: (2*self.alpha*G*M*x)/((x**2 + y**2)**2) - (G*M*x)/((np.sqrt(x**2
        fy=lambda x,y,t: (2*self.alpha*G*M*y)/((x**2 + y**2)**2) - (G*M*y)/((np.sqrt(x**2
        return [vx, vy, fx, fy]
```

Que establece las ecs. de movimiento para x e y y retorna las velocidades iniciales y dichas funciones f_x y f_y. Terminado esto se procede a implementar el código para el método de Euler Explícito, que consiste en definir una función $avanza_euler(dt)$ como:

```
def avanza_euler(self, dt):
        t0=self.t_actual
        x0,y0,vx0,vy0=self.y_actual
        fx=self.ecuacion_de_movimiento()[2]
        fy=self.ecuacion_de_movimiento()[3]
        vxn=vx0+dt*fx(x0,y0,t0)
        vyn=vy0+dt*fy(x0,y0,t0)
        xn=x0+dt*vxn
        yn=y0+dt*vyn
        self.y_actual=xn,yn,vxn,vyn
        pass
```

Este método consiste básicamente en calcular la derivada de la función posición y velocidad mediante la definición de derivada, despejar la variación de cada función en función del paso dt y su derivada y con esto avanzaz en un tiempo dt las condiciones iniciales y actualizar la variable $self.y_actual$ con ellas.
Ahora lo mismo pero con el método de Runge-Kutta de orden 4, se define la función $avanza_rk4(dt)$, con $v = \frac{dx}{dt}$ para formar un sistema de EDO's de primer orden, y toma la forma:

```
def avanza_rk4(self, dt):
        t0=self.t_actual
```

```
        x0,y0,vx0,vy0=self.y_actual
        fx=self.ecuacion_de_movimiento()[2]
        fy=self.ecuacion_de_movimiento()[3]
        k1x=dt*vx0
        k1y=dt*vy0
        l1x=dt*fx(x0,y0,t0)
        l1y=dt*fy(x0,y0,t0)
        k2x=dt*(vx0+l1x/2.0)
        k2y=dt*(vy0+l1y/2.0)
        l2x=dt*fx(x0+k1x/2.0,y0+k1y/2.0,t0+dt/2.0)
        l2y=dt*fy(x0+k1x/2.0,y0+k1y/2.0,t0+dt/2.0)
        k3x=dt*(vx0+l2x/2.0)
        k3y=dt*(vy0+l2y/2.0)
        l3x=dt*fx(x0+k2x/2.0,y0+k2y/2.0,t0+dt/2.0)
        l3y=dt*fy(x0+k2x/2.0,y0+k2y/2.0,t0+dt/2.0)
        k4x=dt*(vx0+l3x)
        k4y=dt*(vy0+l3y)
        l4x=dt*fx(x0+k3x,y0+k3y,t0+dt)
        l4y=dt*fy(x0+k3x,y0+k3y,t0+dt)
        xn=x0+(k1x+2*k2x+2*k3x+k4x)/6.0
        vxn=vx0+(l1x+2*l2x+2*l3x+l4x)/6.0
        yn=y0+(k1y+2*k2y+2*k3y+k4y)/6.0
        vyn=vy0+(l1y+2*l2y+2*l3y+l4y)/6.0
        self.y_actual=xn,yn,vxn,vyn
        pass
```

y para el método de Verlet, se define la función $avanza_verlet(dt)$ que toma la forma:

```
def avanza_verlet(self, dt):
        t0=self.t_actual
        x0,y0,vx0,vy0=self.y_actual
        fx=self.ecuacion_de_movimiento()[2]
        fy=self.ecuacion_de_movimiento()[3]
        xn=x0+vx0*dt+(fx(x0,y0,t0)*(dt**2))/2.0
        yn=y0+vy0*dt+(fy(x0,y0,t0)*(dt**2))/2.0
        vxn=vx0+((fx(x0,y0,t0)+fx(xn,yn,t0+dt))*dt)/2.0
        vyn=vy0+((fy(x0,y0,t0)+fy(xn,yn,t0+dt))*dt)/2.0
        self.y_actual=xn,yn,vxn,vyn
        pass
```

Ahora para calcular la energía total de la órbita para ciertas condiciones iniciales, se define la función $energia_total()$ como:

```
def energia_total(self):
        x0,y0,vx0,vy0=self.y_actual
        E=0.5*m*(vx0**2 + vy0**2) + (self.alpha*G*M*m)/(x0**2 + y0**2) - (G*M*m)/(np.sqrt
        return E
```

y con esto la clase *Planeta* queda definida y lista para ser usada en la siguiente parte de la pregunta.

2.2 Pregunta 2.2

2.2.1 Introducción

- Se pide estudiar el caso para $\alpha = 0$ y para las condiciones iniciales:

$$x_0 = 10$$
$$y_0 = 0$$
$$v_x = 0$$

escogiendo por cuenta propia $v_y = 0.14$ (como una suerte de velocidad de escape) para una redefinición de $GMm = 1$ y $m = 1$, y graficar las órbitas y energías de éstas por los tres métodos definidos en la clas *Planeta*.

2.2.2 Procedimiento

- Para poder realizar esto, usamos entonces la clase *Planeta*, definiendo el objeto de clase como $Planeta(condicion_inicial, \alpha = 0)$ con $condicion_inicial = [10, 0, 0, 0.14]$. Entonces solo basta ahora iterar cada una de las funciones predefinidas en la clase para obtener una lista con las posiciones y velocidad al avanzarlas un paso de tiempo dt para aproximadamente 5 órbitas, que corresponden según la ley de Kepler sobre el periodo de una órbita a $t \approx 1000[s]$.

2.2.3 Resultados

- A raíz del código descrito y creado, se obtienen los siguientes gráficos para las órbitas y para las energías asociados a los distintos métodos usados:

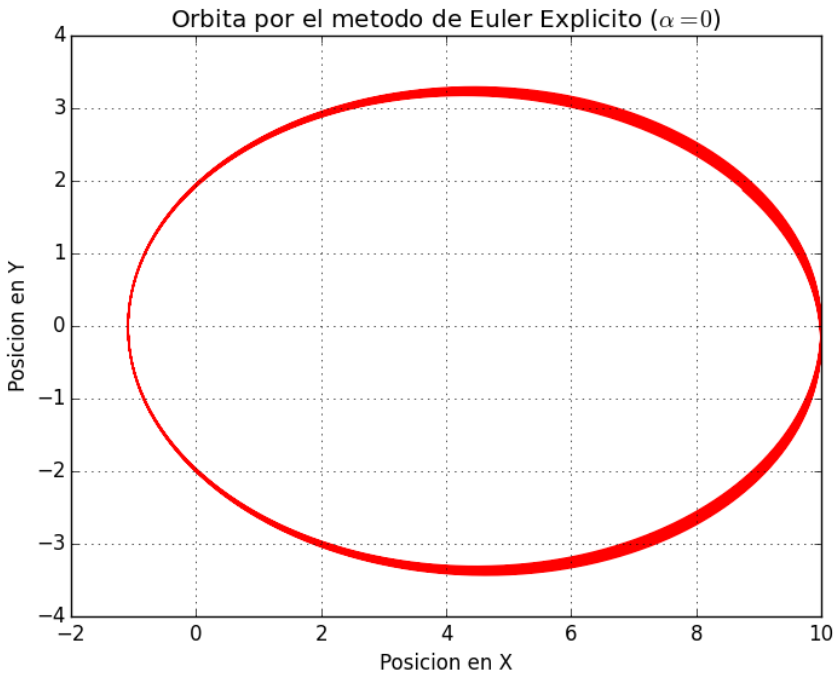

Figure 1: Gráfico de órbita por el método de Euler Explícito.

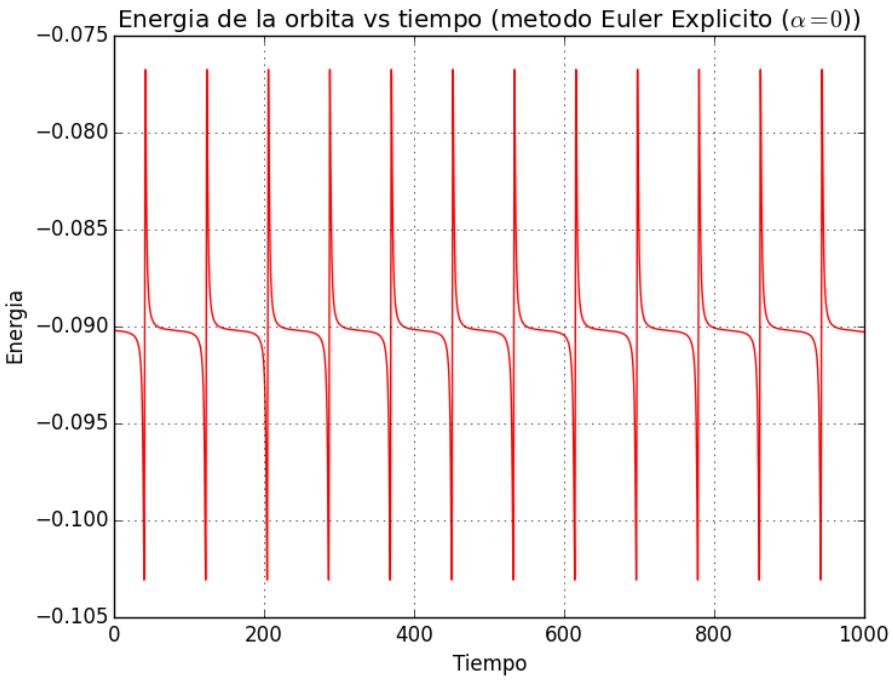

Figure 2: Gráfico de energía de órbita por el método de Euler Explícito.

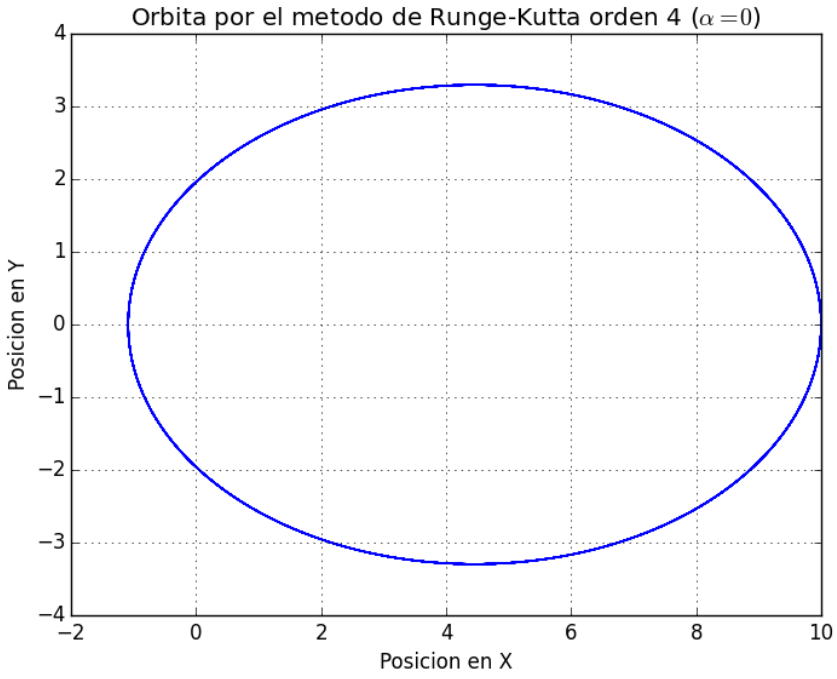

Figure 3: Gráfico de órbita por el método de Runge-Kutta orden 4.

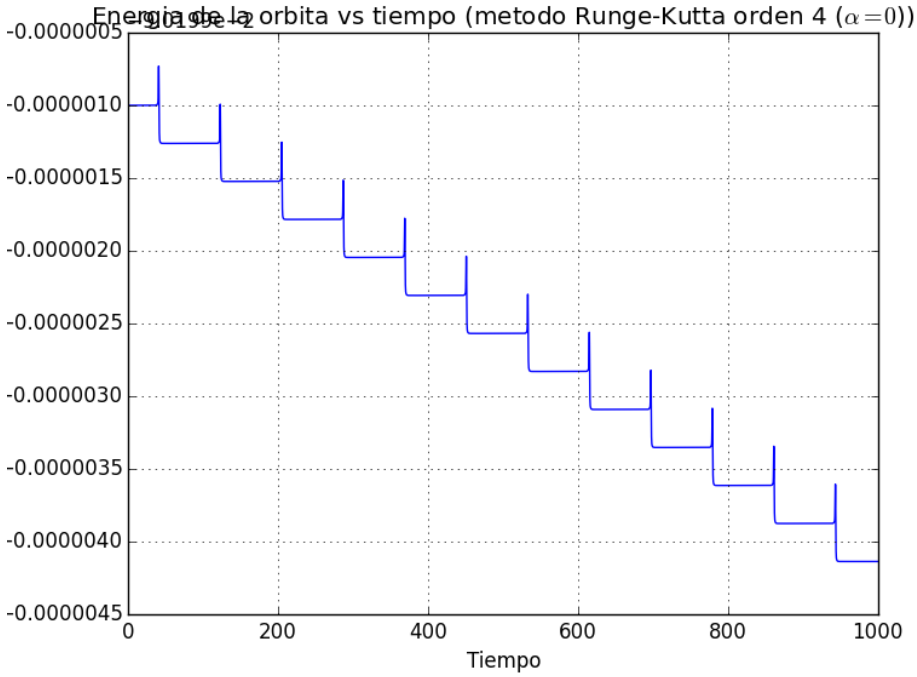

Figure 4: Gráfico de energía de órbita por el método de Runge-Kutta orden 4.

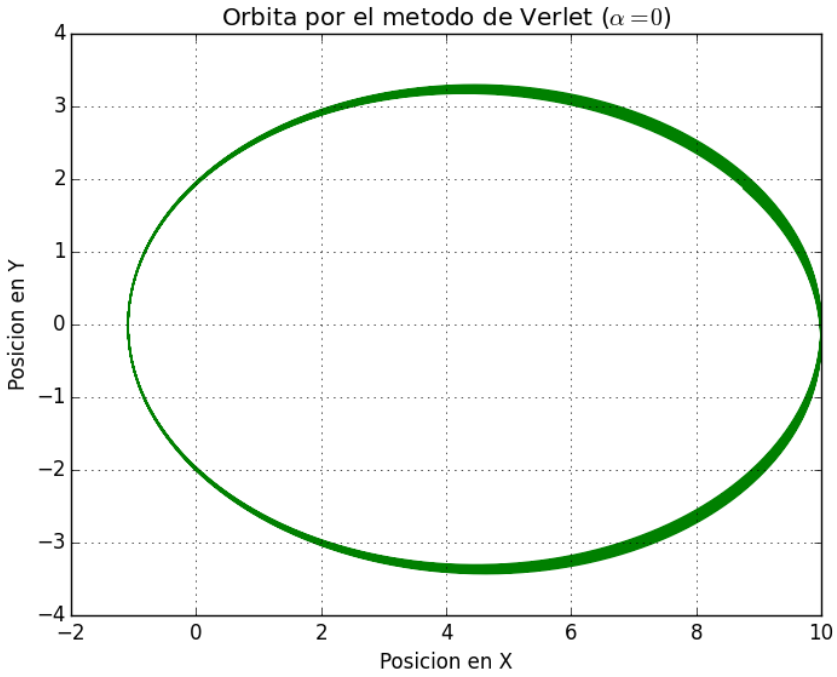

Figure 5: Gráfico de órbita por el método de Verlet.

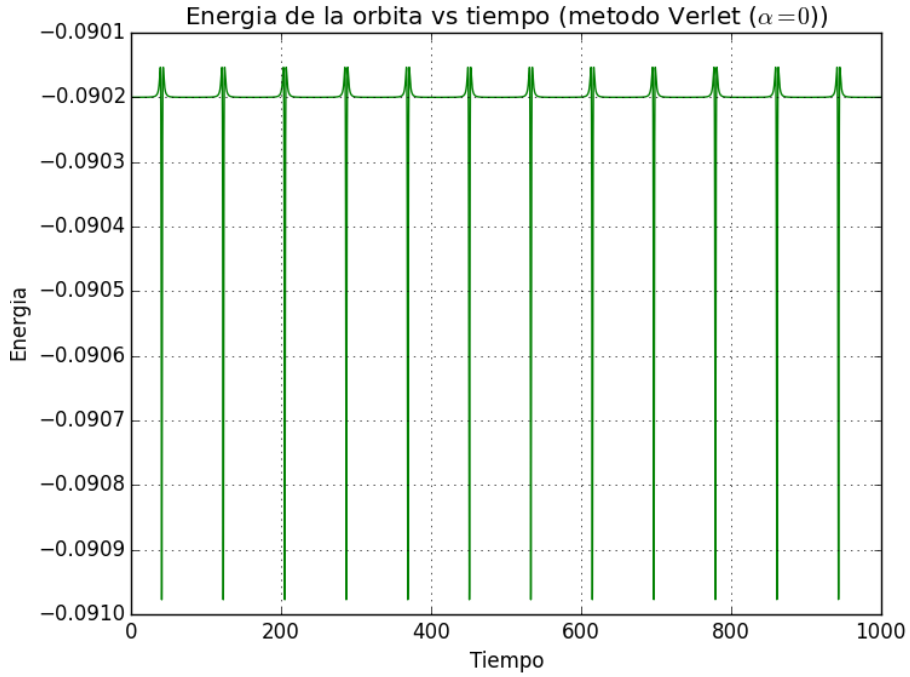

Figure 6: Gráfico de energía de órbita por el método de Verlet.

2.2.4 Conclusiones

- De los gráficos anteriores se puede ver claramente que las energías totales para las condiciones iniciales, oscilan (debido a que las órbitas van cambiando a medida que las funciones hacen avanzar las condiciones iniciales, no por una precesión en sí, sino por el error asociado a cada método) pero son siempre negativas, esto porque en un tiempo dt ("instantáneo") las órbitas se comportan como órbitas cerradas, es decir, que poseen energía negativa debido a el potencial gravitatorio producido por la fuerza central que apunta en sentido negativo hacia el centro en coordenadas polares. También se puede ver que los gráficos de Euler Explícito y Verlet presentan errores numéricos debido a que las órbitas no deberían precesar, pero lo hacen en una proporción muy pequeña debido a estos errores, en cambio el método de Runge-Kutta de orden 4, que posee un error mucho menos, se ve una órbita mas prolija y estática.

2.3 Pregunta 2.3

2.3.1 Introducción

- En esta pregunta se pide ahora estudiar el caso para $\alpha = 10^{-2.808}$ por el método de verlet para el caso de 30 órbitas y graficar la órbita y energía en dicho caso. En esta pregunta se verá como precesa la órbita bajo ese parámetro, que a pesar de ser casi 3 órdenes de magnitud menor que 1, cambia bastante la situación de la órbita en el tiempo.

2.3.2 Procedimiento y resultados

- Para esto entonces procedemos como en el caso anterior pero para un tiempo $t \approx 6000[s]$, bajo las mismas condiciones iniciales pero para $\alpha = 10^{-2.808}$. Dicho esto, entonces se obtienen los siguientes gráficos:

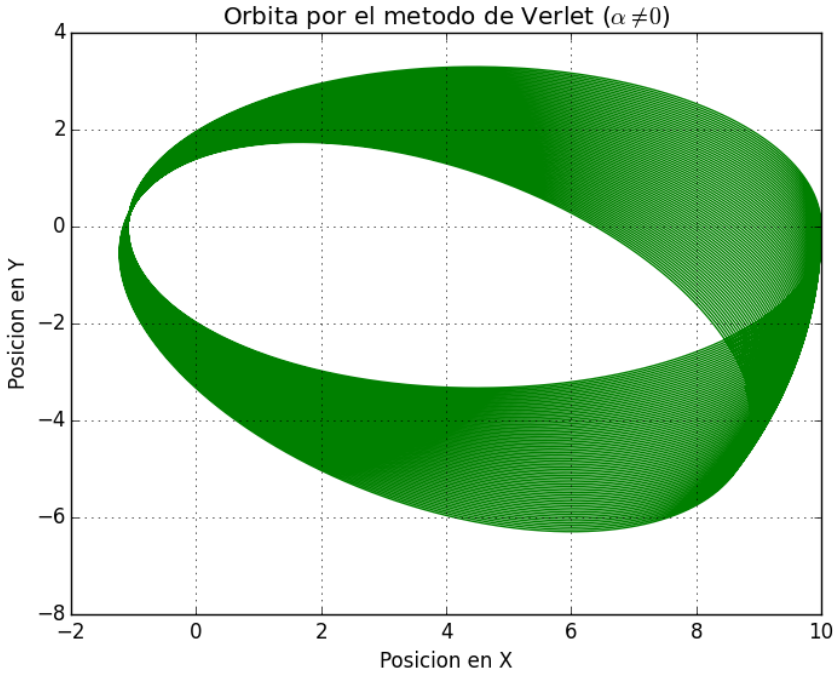

Figure 7: Gráfico de la órbita por el método de Verlet para $\alpha \neq 0$.

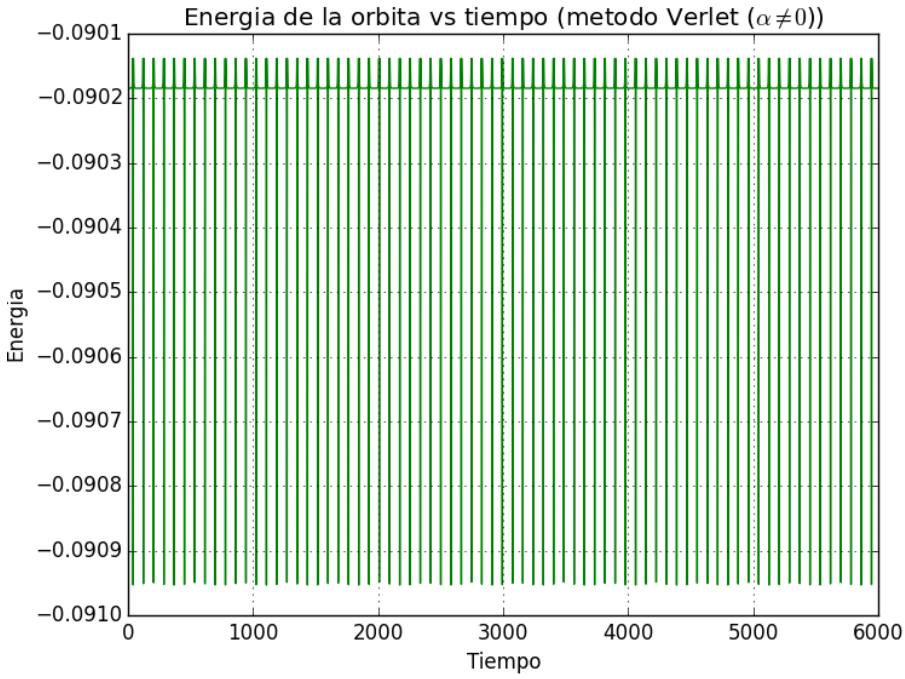

Figure 8: Gráfico de energía de órbita por el método de Verlet para $\alpha \neq 0$.

Ahora también se pide calcular la variación del ángulo de precesión o velocidad angular de precesión. Para esto se debe calcular la posición del afelio al final de la precesión y su variación angular con respecto a la horizontal, esto se hizo mediante el código:

```
...
for h in range(n):
```

```
    x808[h],y808[h],vx808[h],vy808[h]=P.y_actual
    E808[h]=P.energia_total()
    d=np.sqrt(x808[h]**2+y808[h]**2)
    afelio.append(d)
    P.avanza_verlet(dt)
afeliofinal=max(afelio[5700:6001])
indxafeliofinal=afelio.index(afeliofinal)
xafeliofinal=x808[indxafeliofinal]
yafeliofinal=y808[indxafeliofinal]
angprec=np.arctan(yafeliofinal/xafeliofinal)
tiempoafeliofinal=indxafeliofinal/(10.0)
velprec=angprec/tiempoafeliofinal
...
```

que lo que hace es asignar a una lista $afelio$ las distancias a cada punto de la órbita, para luego buscar en un tramo de la lista la distancia máxima, es decir, el afelio (específicamente desde los índices 5700 a 6000, debido a que si cada órbita tiene un periodo aproximado de $t = 200[s]$, entonces para 29 órbitas el tiempo aproximado sería $t = 580$, esto porque pasa por el afelio final al completar la órbita 29 ya que por errores asociados probablemente no alcance a llegar al afelio de la órbita 30 el programa). Ahora para poder obtener el ángulo hay que obtener las posiciones en x e y para luego por la relación trigonométrica de la tangente del ángulo, obtener éste último que correspondería al ángulo de precesión total. Finalmente con este ángulo, obtenemos la variación angular como $\theta_f - \theta_0 = \delta\theta$ y con esto se obtiene para la velocidad angular de precesión el valor de $\omega_{precesion} = -8.87955032316e - 05[rad/s]$.

2.3.3 Conclusiones

- Como se ve en la Figura 7, la órbita precesa en un ángulo determinado despues de 30 órbitas y la energía total de la órbita varía en un orden de magnitud de 10^{-3} siendo siempre negativa.
  Como conclusión final a este informe se puede destacar que a pesar de que el parámetro α sea bastante menor que 1 tiene un efecto en la órbita descrita por el modelo relativista usado no despreciable, haciéndola precesar pero para un periodo de tiempo prolongado.