Métodos Numéricos para la Ciencia e Ingeniería
Informe Tarea

Benjamín Guerrero
18.862.370-0
20 de Octubre, 2015

1. Pregunta 1

1.1. Introducción

Se pide ver un tutorial desarrollado por la compañía Software Carpentry, que busca ayudar a las personas a mejorar el diseño de software que estas desarrollen.

El tutorial se encuentra en: http://swcarpentry.github.io/v4/invperc/index.html o en https://www.youtube.com/playlist?list=PL5859017B018F03F4 (como una lista de videos en YouTube). Cabe señalar que el tutorial se encuentra en inglés.

Una vez visto el tutorial, se pide responder las siguientes preguntas:

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

2) ¿Cuál es la idea detrás de la función *mark_filled*? ¿Por qué es buena idea crearla en vez del código original al que reemplaza?

3) ¿Qué es *refactoring*?

4) ¿Por qué es importante implmentartests que sean sencillos de escribir? ¿Cuál es la estrategia usada en el tutorial?

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

6) ¿Qué es *lazyevaluation*?

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

1.2. Procedimiento

El tutorial usa un ejemplo para mostrar cómo mejorar los programas. Este es calcular cómo se propagará la contaminación en una roca fracturada, usando un grid para representar la roca, con números naturales que representan lo que esa parte de la roca resiste a la contaminación. Se contamina la casilla en el centro de la roca, y luego se identificará el vecino con menor resistencia. Este se contamina, y se repite el proceso hasta que se llega al borde de la roca.

Se responderán las preguntas en este informe, a continuación.

1.3. Resultados

1) La idea es la siguiente: Creas primero el núcleo de tu programa (la parte principal), usando funciones inexistentes para ejecutar lo pedido, y luego defines las funciones, teniendo claro para qué se van a usar estas. Esto es útil porque es más fácil de corregir si el programa no funciona correctamente, ya que está construido de forma modular (uno sabe más fácilmente donde se encuentra el bug).
2) La idea detrás de la función *mark_filled* es que define la casilla del grid como "marcada", es decir, que esa parte se contamina. Esto reemplaza a definir el centro del grid como "-1", para después correr *fill_grid*.
   La función asume que la casilla se encuentra en el grid. Si no lo está, la función *assert* tira un error que dice que no se encuentra dentro del grid. Esto es mucho mejor porque ayuda en la documentación (dice exactamente que se espera de las coordenadas, mejor que simples comentarios), da mensajes de error más significativos que los genéricos IndexError de Pyhton (el usuario sabe exactamente donde fue el error), y asegura que los valores sean positivos, previniendo errores que sucederían en el programa si las coordenadas fueran negativas.
3) *Refactoring* se define, en el tutorial, como cambiar la estructura de un programa, sin modificar su comportamiento o funcionalidad, paramejorar su calidad.
4) Es importante porque, mientras más simple el test, menos es la posibilidad de que el test tenga bugs que corregir. En el tutorial, la estrategia usada es escribir los grids como strings multiíneas, lo que permite tener una variedad de casos para testear. Luego, se crean funciones que convierten el string en una lista de listas, para usar *fill_grid* en esta. Luego, se compara el resultado obtenido con el resultado predicho, para ver si el test funciona correctamente.
5) La primera gran idea es intercambiar memoria por tiempo, es decir, grabar información redundante para no tener que calcularla cada vez. Esto hace que los algoritmos funcionen más rápido. La otra es *lazy evaluation*, que se explicará a continuación.
6) *Lazy evaluation* es una técnica de optimización en la que no genera los valores hasta que los necesita (En el caso del grid, este se genera vacío, y sólo se generan los vecinos a las casillas marcadas). Si bien esto hace el código más

complejo, ayuda a que se ejecute más rápido. Básicamente, intercambiamos tiempo de programación por tiempo de ejecución.

7) La otra moral del tutorial es que, si uno quiere escribir un programa rápido, tiene que empezar escribiendo un programa simple, luego testear ese programa, y después lo mejora parte por parte, reutilizando los tests para comprobar que el programa aún funciona.

2. Pregunta 2

2.1. Introducción

Se define, para planetas cercanos al Sol, la energía potencial gravitatoria como:

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

En este caso, G es la cte de gravitación universal, M es la masa del sol, m es la masa del planeta, r es la distancia entre los centros del Sol y del planeta, y α es una cte muy pequeña. Si α fuera cero, tendríamos la energía potencial gravitatoria en la física newtoniana, pero como es distinto de 0, las órbitas de los planetas precesan, es decir, el afelio gira alrededor del Sol. Este caso se acerca a la teoría de la relatividad general de Einstein.

El repositorio contiene un archivo llamado *planeta.py*, que contiene la clase *Planeta*. Esta clase tiene algunos métodos vacíos, que deberían ser una función *ad hoc*, integraciones en Euler explícito, RK4, y Verlet, y una ecuación para calcular la energía. También contiene el archivo *solución_usando_euler.py*, que muestra como importar una clase de otro archivo.

Primero, se pide implementar los métodos de la clase *Planeta*. Se pueden expandir los docstrings, y añadir los métodos y parámetros que uno estime conveniente.

Usando la clase, después se pide resolver la ecuación de movimiento planetario con las siguientes condiciones iniciales:

$$x_0 = 10$$

$$y_0 = 0$$

$$v_x = 0$$

Para v_y, se puede escoger lo que se quiera, pero se tiene que asegurar que la energía total sea menor a 0 (potencial + cinética). Se debe integrar la ecuación de movimiento por más o menos 5 órbitas usando Euler explícito, RK4, y Verlet, y se debe plotear la trayectoria y la energía en función del tiempo para cada método.

Luego, considere el caso $\alpha = 10^{-2.XXX}$, en el que XXX son los últimos 3 dígitos del RUT antes del dígito verificador (en este caso, es 370). Se debe integrar la ecuación de movimiento usando Verlet por 30 órbitas. Luego, se debe calcular la velocidad angular del afelio. Se debe especificar cómo se hizo esto y cómo se calculó el afelio. Una vez hecho esto, se debe graficar la trayectoria, y la energía en función del tiempo.

2.2. Procedimiento

Primero, calculemos la ecuación de movimiento. Para esto, tenemos que sacar la aceleración de la partícula. Se aprecia que la única fuerza que se ejerce sobre la partícula es la gravitatoria. Usando un plano de 2 dimensiones, y recordando que la fuerza de gravedad se encuentra en función de un vector unitario, asumamos que:

$$r = \sqrt{x^2 + y^2}$$

$$|F|\frac{\vec{r}}{r} = |F_x|\frac{\vec{x}}{r} + |F_y|\frac{\vec{y}}{r}, (\vec{r} = r\hat{r})$$

Ahora, $F = -\nabla U$. Por lo tanto, primero hay que calcular el negativo de la derivada de U.

$$-\nabla U = -\left(-1 * \frac{-GMm}{r^2} + -2 * \alpha * \frac{GMm}{r^3}\right)$$

$$F = GMm * \left(-\frac{1}{r^2} + \frac{2\alpha}{r^3}\right)\frac{\vec{r}}{r}$$

$$\frac{F_x}{m} = a_x = \frac{x}{r}GM\left(-\frac{1}{r^2} + \frac{2\alpha}{r^3}\right) = xGM\left(-\frac{1}{r^3} + \frac{2\alpha}{r^4}\right)$$

Así, tenemos a_x. Esto es análogo para a_y. Por lo tanto, podemos definir el método *ecuación_de_movimiento* en la clase *Planeta* para que retorne (v_x, v_y, a_x, a_y).

Por conveniencia, se van a usar np.arrays para los métodos, lo que hará calcular las integrales más fácil. Se le añade el argumento *datos=np.array([0,0,0,0])*, para hacer más fácil calcular RK4 (se suman a cada uno de los respectivos valores de y_actual)

Luego, se definen cada uno de los métodos de integración:

Euler explícito, para $y(x)$:

$$y_n = y_{n-1} + h * f(x, y)$$

RK4, para $y(x)$:

$$y_{i+1} = y_i + \frac{1}{6}h\left(k_1 + 2k_2 + 2k_3 + k_4\right),$$

$$\begin{cases} k_1 &= f\left(x_i, y_i\right) \\ k_2 &= f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}hk_1\right) \\ k_3 &= f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}hk_2\right) \\ k_4 &= f\left(x_i + h, y_i + hk_3\right) \end{cases}$$

Verlet, para $x(t)$:

$$\vec{x}_{n+1} = 2\vec{x}_n - \vec{x}_{n-1} + A(\vec{x}_n)\,\Delta t^2. \qquad \ddot{\vec{x}}(t) = \vec{A}(\vec{x}(t))$$

$$\vec{v}(t) = \frac{\vec{x}(t+\Delta t) - \vec{x}(t-\Delta t)}{2\Delta t} + \mathcal{O}(\Delta t^2).$$

Una vez hecho esto, se define la energía total como la suma de la energía cinética y la energía potencial, o sea:

$$\left(v_y^2 + v_x^2\right)\frac{m}{2} + U(r)$$

2) Ahora, vamos a crear 2 archivos llamados *solución_usando_rk4.py* y *solución-_usando_verlet.py*.

En *solución_usando_euler.py*, ya incluido, primero se inicia el programa llamando a la clase *Planeta* recién definida como *p*. Luego, se definen tiempo final, dt, y número de pasos. El tiempo final va a ser 900 (lo que genera 5 órbitas), y dt es 0.1. Por lo tanto, van a haber 9000 pasos.

Se definen *x, y, vx, vy*, y energía como np.arrays vacíos (o con 9000 ceros), y se definen los primeros términos como las condiciones iniciales. En este caso, vy_inicial es 0.25. Se usa *p.energia_total* para calcular la energía inicial.

Ahora, usando *p.avanza_euler*, se itera hasta el paso final, añadiendo los valores de *x, y, vx, vy*, y calculando la energía en cada paso.

Luego, se crean los gráficos pedidos usando *matplotlib.pyplot*. Los gráficos se guardan en un archivo llamado *Euler.jpg*.

En *solución_usando_rk4.py*, se realiza lo mismo. La única diferencia aquí es que se usa *p.avanza_rk4* para integrar. Los gráficos se guardan en un archivo llamado *RK4.jpg*.

Luego, en *solución_usando_verlet.py*, se realiza lo mismo. Eso sí, el primer paso se integra usando RK4 porque el método de Verlet necesita el paso previo para funcionar. Luego, de los pasos 2 a 9000, se usa *p.avanza_verlet*. Los gráficos se guardan en un archivo llamado *Verlet.jpg*

3) Ahora, veamos la velocidad angular de precesión de la órbita. Para esto, tenemos que calcular las posiciones del afelio de la órbita (en x,y), y el tiempo que toma en trasladarse, para calcular la velocidad angular de éste.

Para hacer el cálculo, creamos un archivo que se llama *precesión.py*. Importamos las librerías necesarias (aquí se incluye *scipy.stats*). Esta vez, cuando se llama a la clase Planeta, se agrega el término $\alpha = 10^{-2.370}$. Luego, se integra usando Verlet (con tiempo final 5400), pero esta vez se le añade el término $r = \sqrt{x^2 + y^2}$. También se abre una lista de 3 listas llamada *afelio,* que definirá el tiempo, posición *x* y posición *y* de los afelios encontrados.

Viendo los resultados de la parte anterior, se aprecia que el afelio según Verlet es más o menos 10, así que para cada iteración, se ve si r es afelio o no, usando un valor esperado, una tolerancia y la función auxiliar *es_afelio* (Si el radio es más o menos 10, entonces es afelio).

Luego, se calcula el ángulo $\phi = \arctan\left(\frac{x}{y}\right)$, para cada afelio (x,y), en radianes. Después, se calculan $d\phi$ y dt (intervalos de ángulo y tiempo), y después, la velocidad angular para cada intervalo: $\omega = \frac{d\phi}{dt}$. Estas velocidades se guardan en una lista llamada *vel_angulares*, y usando *scipy.stats.tmean*, se calcula la velocidad angular promedio. Luego, la trayectoria y la energía en función del tiempo se grafican, y se guardan en un archivo llamado *precesionVerlet.jpg*.

2.3. Resultados

Una vez ejecutados los programas, se obtienen los siguientes gráficos:

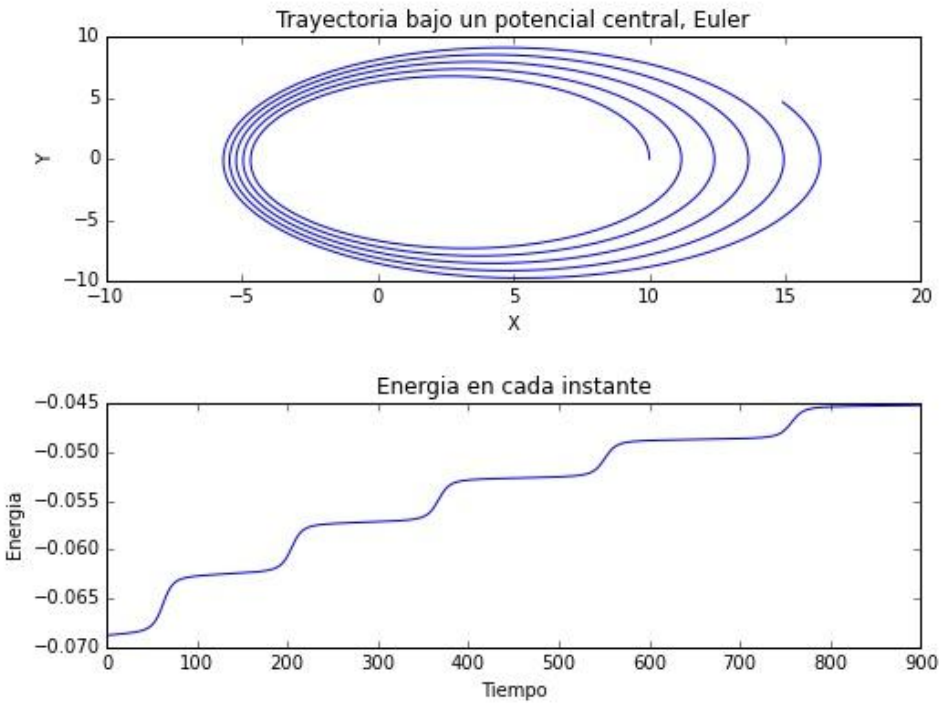

Figura 1: Gráficos usando Euler

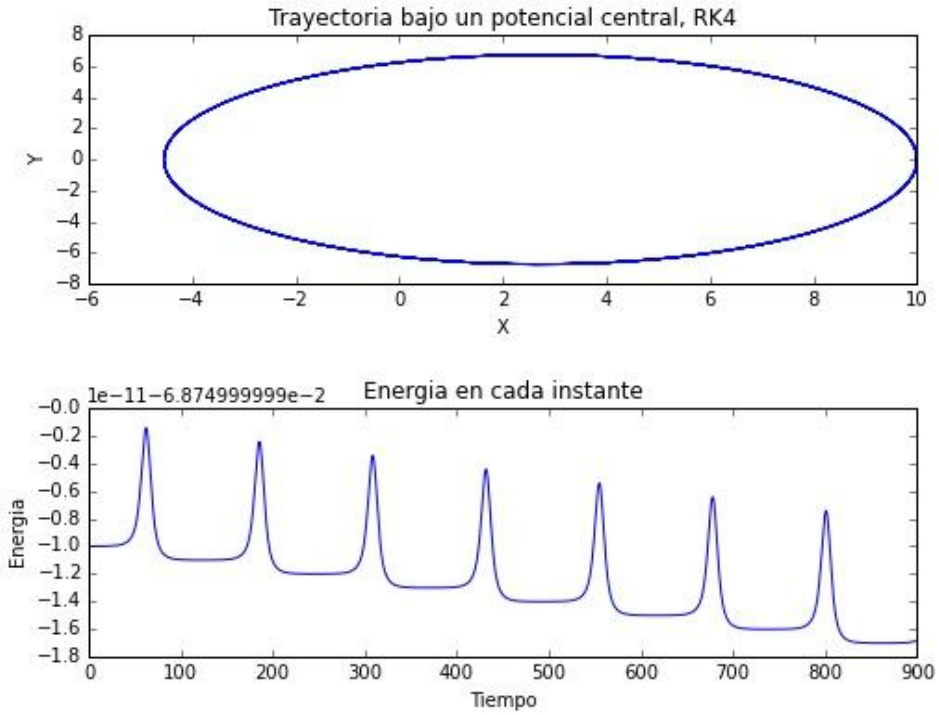

Fig 2: Gráficos usando RK4.

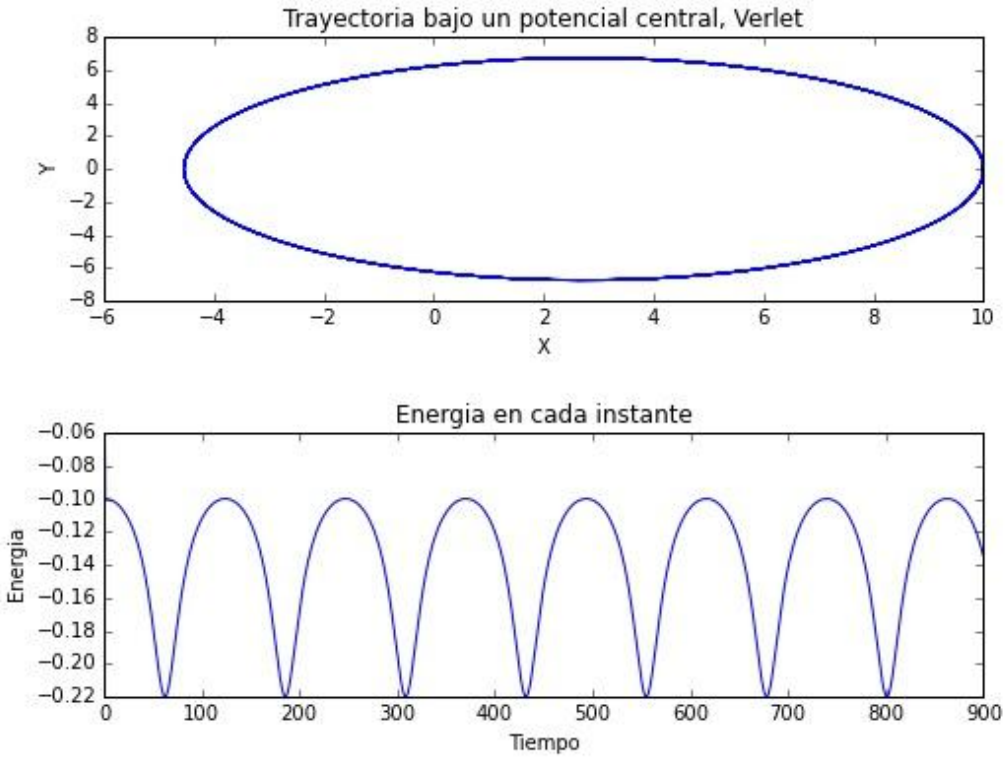

Fig 3: Gráficos usando Verlet

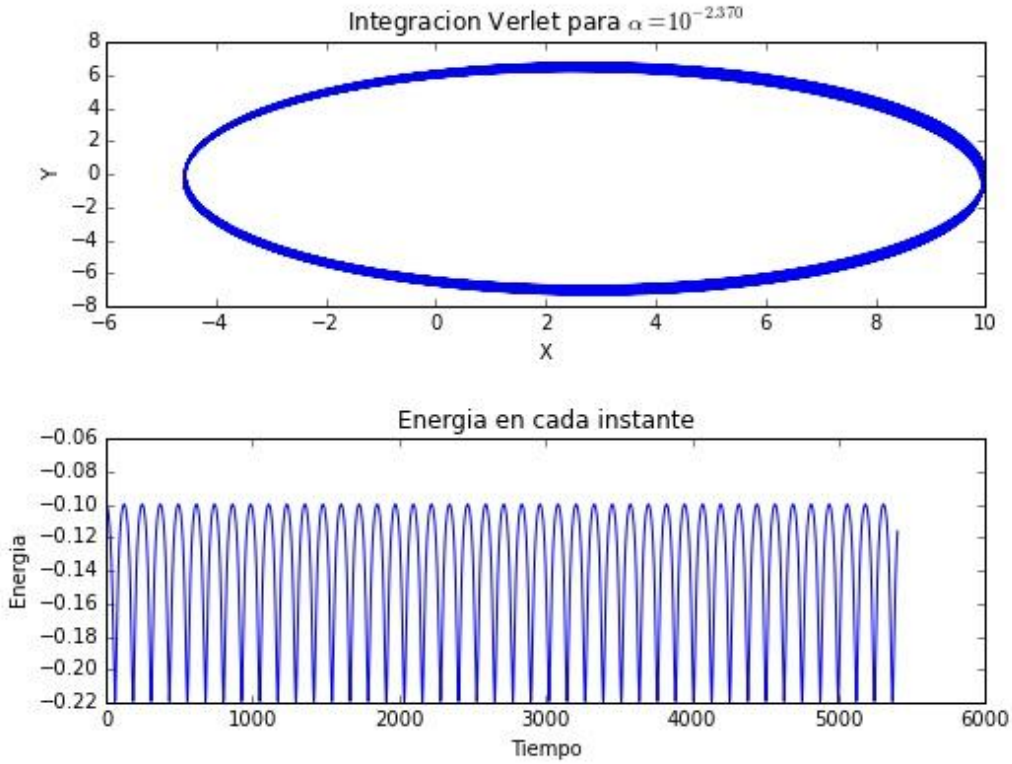

Fig 4: Precesión usando Verlet

La velocidad angular promedio del afelio es, aproximadamente:

$$\omega = -3.51 * 10^{-5} \left(\frac{rad}{s}\right)$$

3. Conclusiones

De la pregunta 1, se puede concluir que es importante crear programas desde su núcleo, y que otras personas entiendan que hace el programa al mirarlo. También es importante hacerlo fácil de probar, para evitar errores. Esto salva esfuerzo y tiempo a la hora de programar.

En la pregunta 2, cada método de integración se comporta de forma diferente. En el método de Euler, la órbita del planeta se aleja cada vez más del centro de gravedad, debido al error asociado al método (es muy grande), y la energía total tiende a cero mientras aumenta la distancia.

En RK4, el planeta orbita de forma constante alrededor del centro de gravedad, pero la energía desciende más y más, haciendo pensar que el planta está perdiendo velocidad.

En Verlet, la órbita es igual a la obtenida con RK4, pero la energía se vuelve periódica, es decir, se conserva mejor en el tiempo (tal como se espera de Verlet).

Ahora, calculando la precesión, se nota que, si bien es apreciable en la trayectoria, no es muy significativa. Es posible que esto se deba a la ecuación misma, o al valor de α definido por el RUT. También se aprecia que la velocidad angular del afelio es negativa. Esto quiere decir que el afelio se traslada en sentido contrario al planeta.

En conclusión, se ve que el método de Verlet es el más apropiado para calcular EDOs relacionadas con energía, puesto que la conserva mejor que RK4 o Euler. Esto permite un cálculo más preciso de variables como la precesión de la órbita.