Práctica 2: Problema de los 2 Cuerpos

Jose Antonio Lorencio Abril

Dado el problema de 2 cuerpos Sol-Tierra, resolverlo con el Método de Euler, el Método de Euler Modificado, y el Método de Runge Kutta de orden 4 (RK4), con paso $h = 1$ *hora*. Además, se pide calcular la duración del año terrestre y el perihelio.

Este problema es un caso particular del problema de los n-cuerpos, que consiste en determinar la evolución de un sistema gravitatorio compuesto por n cuerpos con masa que se atraen gravitatoriamente, dadas las posiciones y velocidades iniciales relativas a un sistema de referencia. Este problema se conoce que no siempre puede solucionarse analíticamente, pues se deben tener en cuenta colisiones entre los cuerpos y diversos efectos físicos que complican mucho el problema, además de la complejidad intrínseca de tener una cantidad ingente de ecuaciones diferenciales (recordemos que todo cuerpo atrae a todos los demás).

Sin embargo, el problema de los 2 cuerpos sí es resuelto de forma analítica (por Johann Bernoulli), y, dependiendo de las condiciones iniciales, se obtienen soluciones elípticas, parabólicas o hiperbólicas. Esto se corresponde con la realidad que observamos: las órbitas estables son elipses (velocidad orbital menor que la velocidad de escape), para las soluciones hiperbólicas podemos pensar en un asteroide lejano que curva su trayectoria al pasar cerca de un astro, y sigue su camino por el espacio (velocidad orbital mayor que la velocidad de escape). Las trayectorias parabólicas son aquellas que quedan en el filo hilo entre las órbitas elípticas y las trayectorias hiperbólicas, o sea, con velocidad orbital exactamente igual a la velocidad de escape.

El problema general de los n cuerpos se vale del cálculo numérico para hacer aproximaciones lo más precisas posibles, es muy usual usarlo para calcular trayectorias de objetos celestes. Por ejemplo, las predicciones de posibles impactos de asteroides en la superficie terrestre se vale de la observación de las condiciones actuales del mismo, y la estimación del futuro del cuerpo, para ver si corremos peligro o no.

Así, si bien este ejemplo puede parecer anodino en un principio, sirve para ver el potencial del cálculo numérico para aplicaciones reales y útiles. Estamos mirando el futuro ni más ni menos.

Tras esta introducción, paso al problema en cuestión.

En clase discutimos algunas simplificaciones que pueden introducirse en la formulación del problema, y obtuvimos un problema final equivalente (bajo las condiciones de las simplificaciones):

$$\begin{cases} \frac{d^2x(t)}{dt^2} = -G\left(M_1 + M_2\right)\frac{x(t)}{\sqrt{x^2(t)+y^2(t)}} & \\ \frac{d^2y(t)}{dt^2} = -G\left(M_1 + M_2\right)\frac{y(t)}{\sqrt{x^2(t)+y^2(t)}} & \\ x\left(t_0\right) = 152.100533 & v_x\left(t_0\right) = 0 \\ y\left(t_0\right) = 0 & v_y\left(t_0\right) = 0.105444 \\ M_1 = 5.9724 \cdot 10^{-3} & M_2 = 1.9885{\cdot}10^3 \\ G = 8.6498928 \cdot 10^{-4} & \end{cases}$$

donde las condiciones iniciales son las correspondientes al afelio, el punto de mayor distancia al Sol de nuestra órbita. Las unidades han sido normalizadas: la unidad de masa es $1um = 10^{27}\ kg$, la de distancia es $1\ ud = 10^{9}\ m$ y la de tiempo $1\ ut = 1\ hora$.

0. Definición del problema en Java

Para resolver esto, como venimos haciendo con los problemas anteriores, he creado un InitialValueProblem que codifica este problema en concreto.

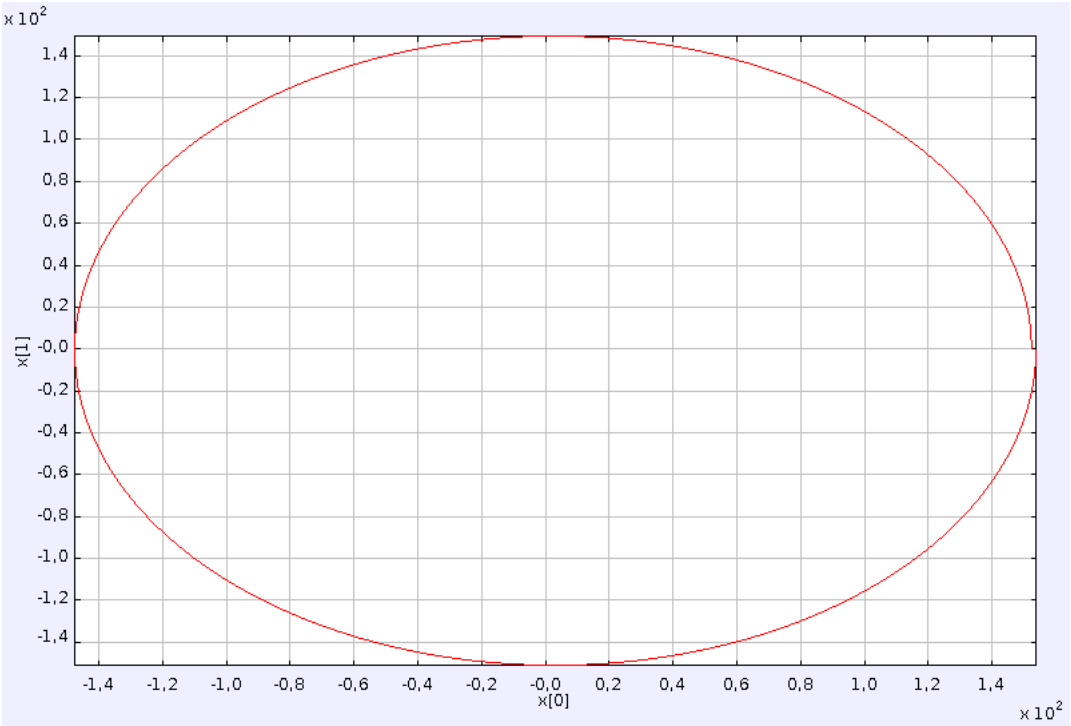
```
public class Problem2Bodies implements InitialValueProblem {
        static double h = 0.5*1.0e-0, x0=152.100533,
                                y0=0.0, vx0=0.0, vy0=0.105444,
                                G=8.6498928e-4, M1=5.9724e-3,
                                M2=1.9885e3, cte = G*(M1+M2);
        @Override
        public double getInitialTime() {
                return 0;
        }

        @Override
        public double[] getInitialState() {
                return new double[] {x0,y0,vx0,vy0};
        }

        @Override
        public double[] getDerivative(double time, double[] state) {
                double mod =
                        Math.sqrt(state[0]*state[0]+state[1]*state[1]);
                mod = mod*mod*mod;

                return new double[] {
                                state[2],
                                state[3],
                                -cte*state[0]/mod,
                                -cte*state[1]/mod};
        }

        public static void main(String[] args) {
                //métodos
        }

}
```

1. Método de Euler

Este método lo hemos usado anteriormente, por lo que no voy a extenderme en su explicación. Solo recalcar la condición de parada, halfWay es una variable booleana que indica si hemos recorrido ya la mitad del camino o no. Es una observación directa que habremos recorrido la mitad del camino cuando la compontente y sea negativa, momento en el que la ponemos a True. Es decir, el bucle termina cuando hemos recorrido más de la mitad de la órbita, e y vuelve a hacerse positiva.

```
//Regular Euler method

FixedStepMethod method = new FixedStepEulerMethod(problem,1.0e-2);

boolean halfWay = false;
if (false) {
        method.solve(2400);
        currentPoint = eulerM.getSolution().getLastPoint();
        currentPoint.println();
}
else {
    previousPoint = currentPoint = method.getSolution().getLastPoint();
    while (!(currentPoint.getState(1) > 0 && halfWay) ) {
        previousPoint = currentPoint;
        currentPoint = method.step();
        if(currentPoint.getState(1)<0 && !halfWay) halfWay = true;
    }
    previousPoint.println();
    currentPoint.println();
}
DisplaySolution.statePlot(method.getSolution(), 0, 1);
```

La gráfica ploteada es:



Que, ciertamente, tiene pinta de ser una elipse. Salvo por el detalle de que no cierra de forma precisa por el principio.

Y obtenemos los dos últimos puntos:

$$x_{n-1} = 153.48586548764547, \quad y_{n-1} = -0.030656332990161295$$
$$v_{x_{n-1}} = 1.9037562581738032 \cdot 10^{-5}, \; v_{y_{n-1}} = 0.10496100753748344$$
$$t_{n-1} = 8824$$

y

$$x_n = 153.48588452520806, \quad y_{n-1} = 0.07430467454732215$$
$$v_{x_{n-1}} = -5.397554343549696 \cdot 10^{-5}, \; v_{y_{n-1}} = 0.10496102212067704$$
$$t_n = 8825$$

Estos puntos son los que vamos a usar para obtener la duración del año, haciendo interpolación mediante bisección, tal como hicimos para el problema del péndulo:

```
StateFunction interpolator =
        new EulerMethodInterpolator(problem, previousPoint);
double zeroYAt = BisectionMethod.findZero
                (interpolator, previousPoint.getTime(),
                currentPoint.getTime(), 1.0e-8, 1);
if (Double.isNaN(zeroYAt)) {
    System.out.print ("Zero not found!!!");
}
else{
    System.out.println
("Regular Euler method gives us a hitting time of t="+zeroYAt/24.);
}
```

Obteniendo una duración del año de 367.6788363953431 días.

La duración del año sideral (lo que tarda la Tierra en volver a un mismo punto de su órbita alrededor del Sol), es $365,256363$ días.

Por lo que obtenemos un error relativo de 0.006632255152097389, lo cual, a priori, podría parecer que no es mucho. Sin embargo, el error absoluto es de unos 2 días y medio, esto quiere decir que, si estableciésemos un calendario con esta duración (haciendo, por ejemplo, un febrero con 30 días), al cabo de 10 años, estaríamos tomando las uvas en lo que debería ser el 25 de enero.

Para solucionar este problema, sabemos que podemos disminuir el paso, o mejorar el método. Y vimos en clase que, por lo general, es mejor utilizar un método más preciso, en términos de rendimiento. Eso es lo que vamos a hacer en los siguientes apartados.

Antes de seguir, vamos a calcular el perihelio. Para ello, vamos a interpolar con los puntos que tenemos en el momento en el que halfWay se pone a True.

```
if(currentPoint.getState(1)<0 && !halfWay) {
        halfWay = true;
        perihelionPrev = (NumericalSolutionPoint) previousPoint.clone();
        perihelionPost = (NumericalSolutionPoint) currentPoint.clone();
}
```

Nótese que he añadido el método clone a la clase NumericalSolutionPoint para poder hacer esto (Java pasa los objetos por referencia, no por copia). Y las variables perihelionPrev y perihelionPost son dos NumericalSolutionPoint previamente definidos.

Finalmente, interpolamos:

```
StateFunction interpolatorPeri =
        new EulerMethodInterpolator(problem, perihelionPrev);
double perihelionAt = BisectionMethod.findZero
        (interpolatorPeri, perihelionPrev.getTime(),
        perihelionPost.getTime(), 1.0e-8, 1);
if (Double.isNaN(perihelionAt)) {
    System.out.print ("Zero not found!!!");
}
else{
    double perihelion = interpolatorPeri.getState(perihelionAt,0);
    System.out.println
        ("Regular Euler method gives us a perihelion ="+perihelion);
}
```

Obteniendo un resultado de -147.72577971444696 ud, es decir, una distancia de $147.72577971444696 \cdot 10^6$ km, frente a la que ofrecen los datos de la NASA, de $152.0977 \cdot 10^6 km$, obteniendo un error relativo de 0.0287.

De nuevo, nos acercamos considerablemente, pero errores relativos bajos, en términos astronómicos suponen grandes errores absolutos. Por este motivo vamos a repetir este ejercicio con el método de Euler modificado y con el método RK4.

2. Método de Euler modificado

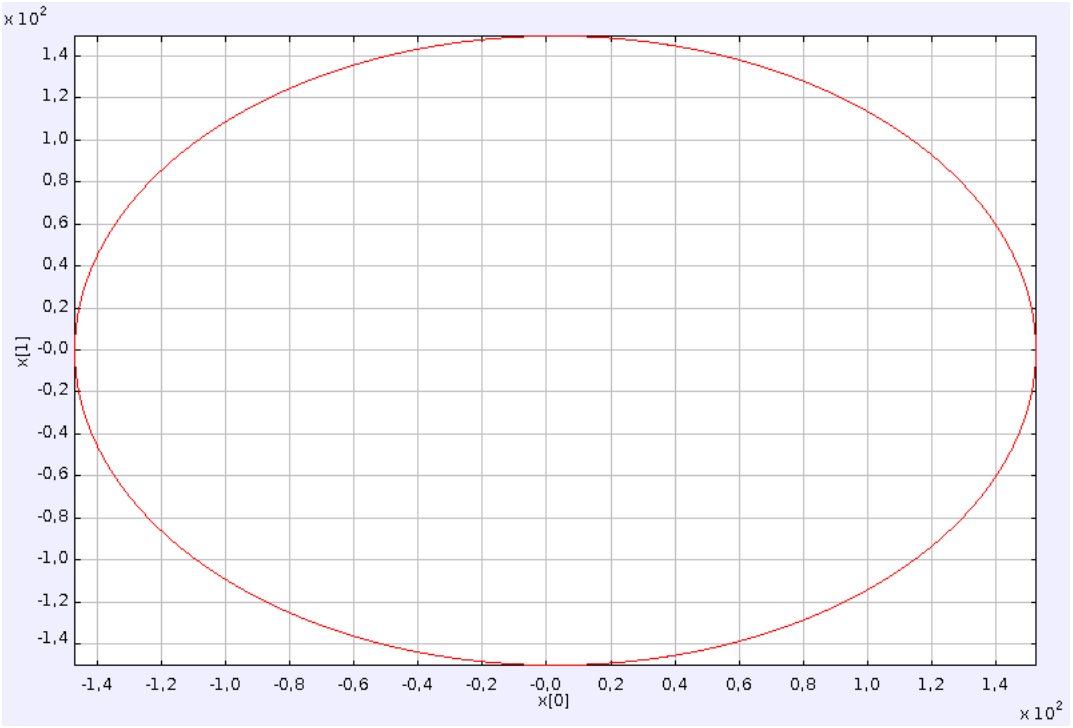
Lo primero es crear la clase que define el propio método:

```
1  public class FixedStepModifiedEulerMethod extends FixedStepMethod{

   public FixedStepModifiedEulerMethod
                  (InitialValueProblem problem, double step){
         super(problem, step);
6  }

   @Override
   protected double doStep(double deltaTime, double time, double[] state) {
         double[] derivative1 = mProblem.getDerivative(time, state);
11       double[] state2 = state.clone();
         for (int i=0; i<state2.length; i++) {
                  state2[i] += deltaTime*derivative1[i];
         }
         double[] derivative2 =
16                        mProblem.getDerivative(time+deltaTime, state2);
     super.addToEvaluationCounter(2);
     for (int i=0; i<state.length; i++) {
         state[i] = state[i] +
                        deltaTime/2 * (derivative1[i] + derivative2[i]);
21   }
     return time+deltaTime;
   }
   }
```

Donde solo hemos modificado la clase FixedStepEulerMethod para que haga los pasos como el método modificado de Euler. Ahora, repetimos los pasos del apartado anterior, pero usando este método.

Dado que el código es el mismo, salvo los nombres de las variables, no lo voy a incluir, voy a exponer y comentar los resultados.

El plot del conjunto de estados obtenido es:



Cuyas diferencias con el diagrama del apartado anterior son inapreciables a simple vista, excepto que vemos que esta parece cerrarse bastante bien. Esto es buena señal, en el sentido de que, dado que el método anterior ya había sido validado en las prácticas anteriores, unas discrepancias excesivas entre ambas soluciones deberían hacer sonar nuestras alarmas.

Los últimos puntos de la solución son:

$$x_{n-1} = 152.10050319847062, \quad y_{n-1} = -0.09471514426855587$$
$$v_{x_{n-1}} = 6.677834563171233 \cdot 10^{-5}, \; v_{y_{n-1}} = 0.10544397913728978$$
$$t_{n-1} = 8764$$

y

$$x_n = 152.10053280223588, \quad y_{n-1} = 0.010728858017873266$$
$$v_{x_{n-1}} = -7.570803823826711 \cdot 10^{-6}, \; v_{y_{n-1}} = 0.10544399966421797$$
$$t_n = 8765$$

Interpolamos con estos valores, pero en este caso vamos a hacerlo usando interpolación mediante polinomios de Hermite de orden 2. Para ello, he programado la clase *hermiteInterpolation*:

```
public class HermiteInterpolation {
private NumericalSolutionPoint p1, p2;

public HermiteInterpolation
        (NumericalSolutionPoint p, NumericalSolutionPoint pp) {
        p1=p;
        p2=pp;
}

public void interpolate
        (double[] sol, int variableObj, int derivada, double objetivo) {
        double y1=p1.getState(variableObj),y2=p2.getState(variableObj),
                vy=p1.getState(derivada),t1=p1.getTime(), t2=p2.getTime()
        //coefficients of the 2nd order equation obtained from Hermite
        double a=(y2-y1-vy*(t2-t1))/((t2-t1)*(t2-t1)),
                        b=vy, c=y1-objetivo;
        //now, we check for solutions
        double discriminant = b*b-4*a*c;
        if(discriminant<0) sol[0]=Double.NaN;
        //If there are no solutions, we return NaN
        //else we get both solutions avoiding catastrophic cancellation
        if(b>=0) {
                sol[0] = -b-Math.sqrt(b*b-4*a*c)/(2*a);
                sol[1] = c/(a*sol[0])+t1;
                sol[0] += t1;
        //we sum t1 because we solved the eq a*(t-t1)^2+b*(t-t1)+c=0
        }
        else {
                sol[0] = -b+Math.sqrt(b*b-4*a*c)/(2*a);
                sol[1] = c/(a*sol[0])+t1;
                sol[0] += t1;
        }
}
}
```

Y ahora usamos esta clase para interpolar, con los resultados anteriores:

```
HermiteInterpolation hinter =
        new HermiteInterpolation(previousPoint, currentPoint);
double[] hittingTime = new double[2];
hinter.interpolate(hittingTime, 1,3,0);
double interpolatedTime;

if(hittingTime[0]*hittingTime[1]<0)
        interpolatedTime = Math.max(hittingTime[0], hittingTime[1])/24.;
else
        interpolatedTime = Math.min(hittingTime[0], hittingTime[1])/24.;
System.out.println
        ("Modified Euler method gives us a
        hitting time of "+interpolatedTime+" days");
```

Obteniendo un período orbital de 365.24152087147945 días, lo que proporciona un error relativo de $4.06348 \cdot 10^{-5}$, un par de órdenes de magnitud mejor que el anterior.

En esta ocasión el error absoluto que obtenemos es de unos 0.015 días, o sea, 20 minutos, aproximadamente. Esto quiere decir, que para acumular el error de 25 días, obtenido con la estimación anterior en 10 años, necesitaríamos 1800 años. Observamos así la considerable mejora de nuestra nueva estimación.

No obstante, más adelante observaremos que nos vamos a alejar del valor exacto al disminuir el paso. ¿Cómo es esto posible si al disminuir el paso estamos aumentando la precisión? La respuesta es que no estamos resolviendo el mismo problema que la NASA, por lo que nuestras estimaciones se ajustan cada vez más a una solución diferente a la real. Cercana, pero distinta.

Para el cálculo del perihelio, hacemos igual que antes. Nótese que no podemos usar Hermite para este propósito, ya que Hermite no trabaja con el vector completo y va avanzando, sino que construye el polinomio que interpola a una variable del vector y su derivada. Es decir, para obtener en qué momento se produce un valor intermedio, debemos conocer de antemano qué valor buscamos (este no es el caso). Por tanto, vamos a usar, de nuevo, bisección.

El resultado obtenido es -147.07117416011727 *ud*, con un error relativo de 0.03305. Por lo que, en este caso, la estimación empeora. Esto puede tener diversas explicaciones, una que considero muy probable es que este perihelio sea realmente mejor que el anterior, pero en términos de nuestro problema simplificado, que no es la misma forma en que la NASA obtiene estas magnitudes, al igual que hemos comentado que ocurre con el período orbital.

3. Método RK4

Por último, repetimos lo ya explicado para el método de Runge-Kutta de orden 4, el cual, por supuesto, hemos programado:

```
public class FixedStepRK4 extends FixedStepMethod {

public FixedStepRK4(InitialValueProblem problem, double step) {
        super(problem, step);
}

@Override
protected double doStep(double deltaTime, double time, double[] state) {
        double[] k1 = mProblem.getDerivative(time, state);
        for(int i=0; i<k1.length; i++) k1[i] *= deltaTime;

        double[] stateAux = state.clone();
        for (int i=0; i<stateAux.length; i++)   stateAux[i] += 0.5*k1[i];
        double[] k2 =
                mProblem.getDerivative(time + 0.5*deltaTime, stateAux);
        for(int i=0; i<k1.length; i++) k2[i] *= deltaTime;

        stateAux = state.clone();
        for (int i=0; i<stateAux.length; i++)   stateAux[i] += 0.5*k2[i];
        double[] k3 =
                mProblem.getDerivative(time + 0.5*deltaTime, stateAux);
        for(int i=0; i<k1.length; i++) k3[i] *= deltaTime;

        stateAux = state.clone();
        for (int i=0; i<stateAux.length; i++)   stateAux[i] += k3[i];
        double[] k4 = mProblem.getDerivative(time + deltaTime, stateAux);
        for(int i=0; i<k1.length; i++) k4[i] *= deltaTime;

    super.addToEvaluationCounter(4);
    for (int i=0; i<state.length; i++) {
        state[i] = state[i] + 1./6. * (k1[i] + 2*k2[i] + 2*k3[i] + k4[i])
    }
    return time+deltaTime;
}
}
```

El plot de la solución es visualmente indistinguible del anterior.

Pasamos a los dos últimos puntos de la misma:

$$x_{n-1} = 152.10050340538947, \quad y_{n-1} = -0.09408168098686828$$
$$v_{x_{n-1}} = 6.633753816243651 \cdot 10^{-5}, \; v_{y_{n-1}} = 0.10544397948347972$$
$$t_{n-1} = 8764$$

y

$$x_n = 152.10053256834337, \quad y_{n-1} = 0.011362312900475513$$
$$v_{x_{n-1}} = -8.011632608189955 \cdot 10^{-6}, \; v_{y_{n-1}} = 0.10544399970075616$$
$$t_n = 8765$$

Vemos que estos puntos son realmente cercanos a los obtenidos en el apartado anterior. Vamos a ver qué sucede con la interpolación. La vamos a hacer, de nuevo, con bisección, ya que Hermite es un interpolador de orden 2, y preferimos interpolar con orden 4. Nótese que hemos programado un RK4Interpolator, muy similar al interpolador que tenemos para Euler, por lo que omito el código. Obtenemos un período orbital de 365.203843464454 días, lo que da un error relativo de 0.0001438, muy similar al del apartado anterior.

Respecto al perihelio, sacamos -147.0711740584012 *ud*, dando un error relativo de 0.03305, como en el apartado anterior. Esto respalda la conclusión que obtuvimos: estamos aproximando bien el problema simplificado, que no se corresponde exactamente con el problema real.

4. Prueba 1: variando h

Vamos ahora a ver qué sucede al variar el paso, tanto con los errores relativos como con el número de operaciones para cada método.

	Euler		**Euler Modificado**	
h	**Error Relativo**	**NOps**	**Error Relativo**	**NOps**
1	0.006632255152097389	8825	$4.063482535571868 \cdot 10^{-5}$	17530
0.1	$5.324611552720103 \cdot 10^{-4}$	87709	$1.3325180450678818 \cdot 10^{-4}$	175298
0.01	$7.617673105542081 \cdot 10^{-5}$	876549	$1.435321044169764 \cdot 10^{-4}$	1752980
0.001	$1.3702698117328113 \cdot 10^{-4}$	8764952	$1.4376025553110126 \cdot 10^{-4}$	17529786

	RK4	
h	**Error Relativo**	**NOps**
1	$1.4378814680916457 \cdot 10^{-4}$	35060
0.1	$1.4378813973206823 \cdot 10^{-4}$	350596
0.01	$1.437881248405191 \cdot 10^{-4}$	3505960
0.001	$1.43787993485409 \cdot 10^{-4}$	35059572

Como podemos observar en las tablas, las diferencias entre métodos solo son importantes para $h = 1$, para los demás tamaños de paso, nos dan valores muy parecidos. Además, al disminuir el paso, la precisión no mejora prácticamente. Por tanto, de estas observaciones (a posteriori) podemos extraer las siguientes conclusiones:

- Para $h \leq 0.1$, el método a usar debería ser Euler estándar, ya que proporciona prácticamente el mismo resultado, pero con una considerable reducción en la cantidad de evaluaciones.
- Para $h = 1$, no debemos utilizar Euler estándar, pues genera una aproximación mala. Y entre los otros métodos, deberíamos elegir Euler Modificado, puesto que el resultado es similar, pero necesitamos menos evaluaciones.

Por supuesto, esto es algo que podemos ver una vez hecho un estudio de este tipo, pero en un entorno real, debemos seleccionar el método antes de saber los resultados. ¿Podemos extraer alguna conclusión más general de estos datos?

El hecho de que los resultados sean tan parecidos debería hacernos recapacitar sobre qué está pasando. Al fin y al cabo, estamos, en principio, utilizando métodos de distinta potencia para una misma tarea, por lo que esperaríamos resultados mucho mejores por parte de los mejores métodos.

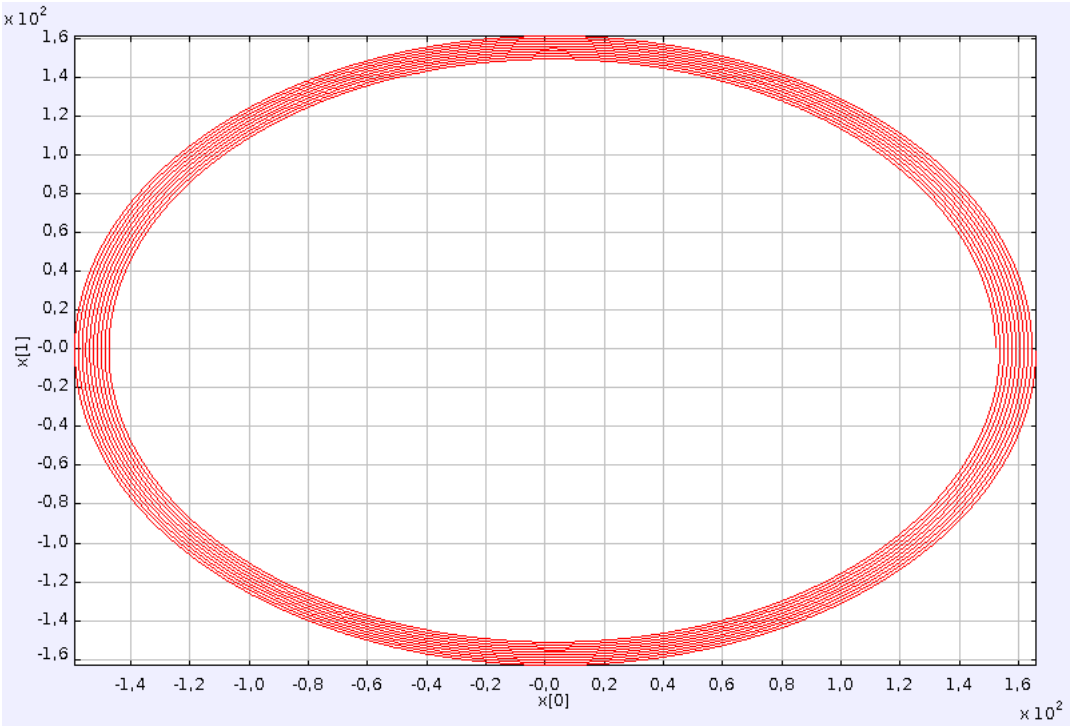
Para ejemplificar el error en nuestro razonamiento, recurro al caso extremo en el que la solución de nuestro problema fuese una recta. En este caso, los tres métodos darían exactamente el mismo resultado, con un error de 0.

Aquí sucede algo parecido, los términos de órdenes 3 y superior serán muy 'débiles', de modo que al interpolar con orden mayor que 2 realmente no obtenemos una ganancia sustancial.

5. Prueba 2: 'pasando años'

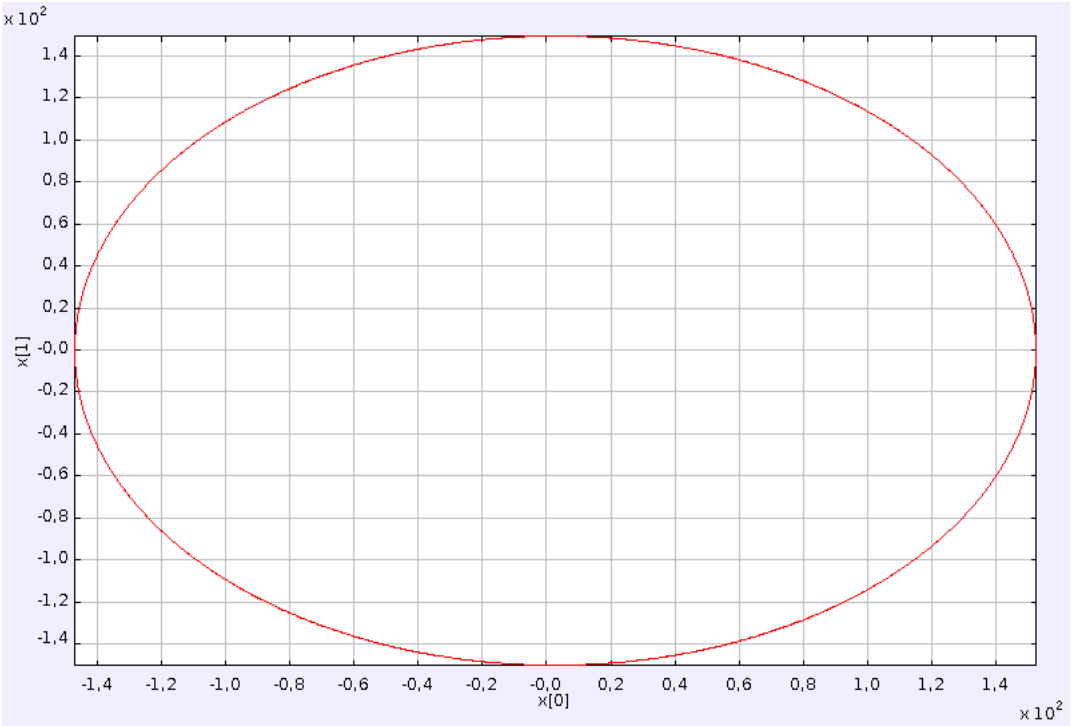
Vamos ahora a ver qué sucede cuando hacemos la Tierra orbitar alrededor del Sol durante varios años, para ver cómo evolucionan las soluciones a largo plazo. Para esto he modificado el código ligeramente, pero no creo necesario mostrarlo, solo he añadido una variable para indicar cuántos años queremos ver pasar.

Comenzamos con paso $h = 1$, dejando pasar 10 años con el método de Euler estándar:



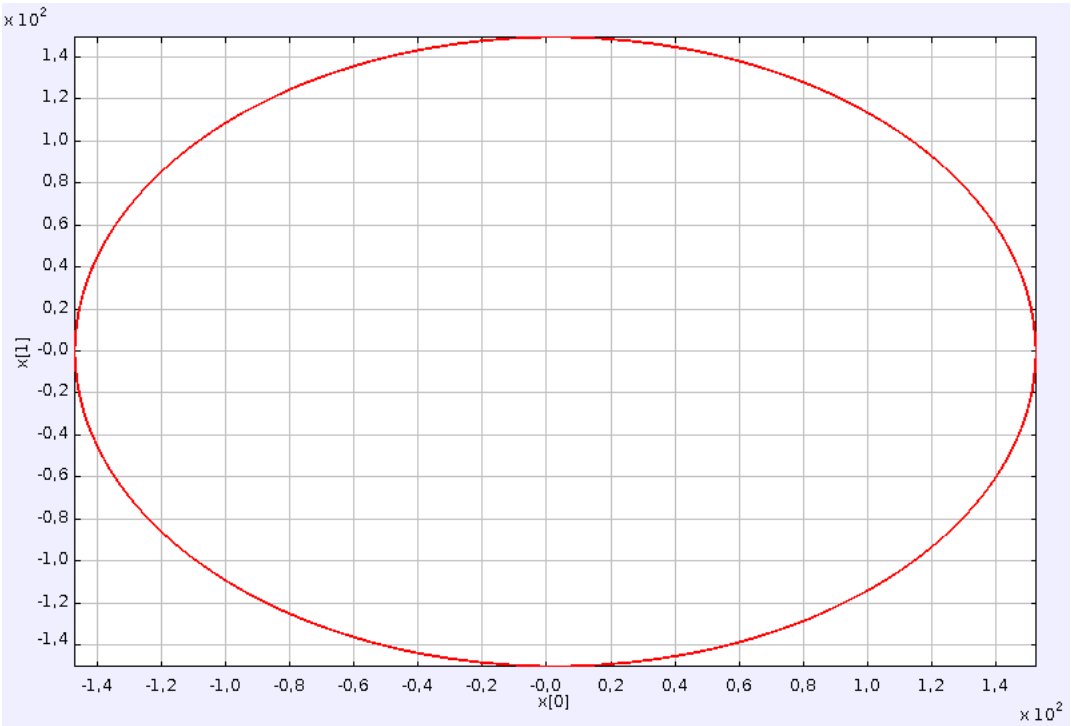
Vemos cómo se aprecia la desviación desde el primer momento, y que se acumula en cada vuelta, dando lugar a grandes errores.

Para Euler modificado y RK4, la verdad es que el resultado es sorprendente:



No se observa desviación alguna para ninguno de los dos métodos. Este experimento nos permite reafirmarnos en nuestra hipótesis anterior: los términos de orden superior a 2 deben ser muy pequeños, ya que se error apenas se acumula.

Vamos a forzar la máquina y dejar pasar 100 años:



Y observamos cómo la línea se engrosa un poco, pero el error es muy pequeño. De hecho, Euler Modificado nos dice que tardaríamos 36520.45 días en dar 100 vueltas alrededor del Sol, tan solo 5 días menos de lo que duran en realidad 100 años.

El método estándar de Euler nos arroja un valor muchísimo peor, de 61225.44 días.

Y RK4 nos dice que tardaríamos 36520.38 días, un valor muy similar al arrojado por Euler Modificado.

Podemos, ahora sí, asegurar con bastante certeza que este problema presenta unos términos de orden superior a 2 muy pequeños, que ni tan siquiera muestran acumulación del error a largo plazo.

6. Conclusiones

Como hemos explicado a lo largo del informe, los términos de orden 3 y superior parecen ser despreciables en este problema, por ello parece una decisión sensata olvidarnos de usar métodos más potentes que orden 2, ya que la mejora en el resultado será mínima, pero gastaremos más recursos.

Además, según los resultados obtenidos, parece que disminuir el paso h tampoco tiene grandes repercusiones en las soluciones. Esto puede deberse a que la solución es bastante *smooth*. Así que tampoco nos conviene usar un h excesivamente pequeño, por la misma razón de eficiencia comentada anteriormente.

Por último, comentar que aún más interesante que este análisis a posteriori de este problema en concreto sería encontrar las propiedades generales que debe verificar un problema para poder descartar métodos demasiado potentes o pasos muy bajos. Si fuésemos capaces de hacer esto, las decisiones posteriores serían menos y, por tanto, más sencillas.

Para este problema en concreto, si hubiéramos sido capaces de asegurar que los términos de órdenes supercuadráticos son despreciables y que la solución es bastante *smooth*, entonces podríamos haber atacado directamente con Euler Modificado con paso $h = 1$ o $h = 0.1$.