Ecuaciones en Derivadas Parciales y Análisis Numérico

Prácticas

Capítulo 2. Ecuaciones diferenciales ordinarias (EDOs).

2.1 Resolución de una ecuación diferencial ordinaria.

Vamos a resolver numéricamente la siguiente ecuación diferencial:

$$\frac{dy}{dt} = y(1-y)$$

con condición inicial *y(0) = 0.2*. Para empezar vamos a utilizar el *método de Euler explícito*. Este algoritmo, que estudiaréis en teoría con más detalle, se basa en la aproximación:

$$y(t+dt) \approx y(t) + dt \cdot y'(t) = y(t) + dt \cdot y(t)(1-y(t))$$

para un valor de *dt* pequeño. Para empezar, tomamos un conjunto de instantes de tiempo $t_n = t_0 + n\,dt$ y definimos $y_0 = y(0)$. A continuación, definimos el valor de los puntos y_n de forma recursiva, mediante el esquema numérico:

$$y_{n+1} = y_n + dt \cdot y_n(1-y_n)$$

Lo implementamos del siguiente modo

```
tsteps=1000;          % Subdivisiones del intervalo
t_f=10;               % valor final de tiempo
dt=t_f/tsteps;        % el paso de tiempo
t=0:dt:t_f;           % definimos los diferentes tiempos
y=zeros(1,tsteps+1);% inicializamos y
y(1)=0.2;             % definimos dato inicial
for n=1:tsteps
    y(n+1)=y(n)+dt*y(n)*(1-y(n));
end                   %metodo de Euler explicito
```

Dibujamos los resultados

```
plot(t,y);
```

Matlab tiene implementados varios métodos numéricos que podemos aplicar directamente. Crea un fichero de Matlab (File/New/M-file) que contenga las siguientes líneas:

```
function deriv=edo(t,y)
deriv=y*(1-y);
```

Guarda el archivo con el nombre ***edo.m*** y establece el directorio donde lo has guardado como directorio de trabajo. Vuelve a la línea de comandos y teclea:

```
t_0=0;
t_f=10;
y0=0.2;
[t,y]=ode23('edo',[t_0,t_f],y0);
plot(t,y)
```

La function Matlab **ode23** es una implementación de un método Runge-Kutta de orden (2,3). En este caso no es necesario poner el paso de tiempo porque la misma función ode23 lo va reduciendo sobre la marcha si es necesario. Otra function Matlab es **ode45** basado en un método explícito Runge-Kutta de orden (4,5). En general, **ode45** es la mejor función para aplicar en un primer momento a muchos de los problemas:

```
[t,y]=ode45('edo',[t_0,t_f],y0);
plot(t,y)
```

En este caso, es fácil comprobar que la función:

$$y(t)=\frac{e^t}{4+e^t}$$

es la solución exacta del problema de valor inicial anterior. Podemos usar esta referencia para comprobar qué tal precisión hemos obtenido con los distintos métodos.

```
y_exacta=exp(t)./(4+exp(t));
plot(t,abs(y-y_exacta)); %repetir cuando y se ha calculado con
                         % distintos metodos
```

2.2 Errores.

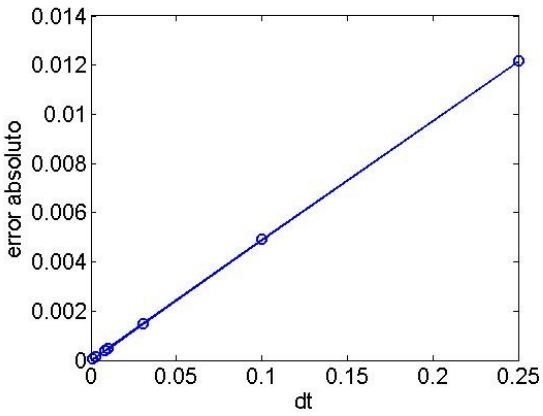
Vamos a estudiar la precisión que obtenemos en función del paso de tiempo utilizado. Para empezar creamos una función en matlab que devuelva sólo el valor aproximado de *y(1)* tomando como argumento el número de pasos temporales:

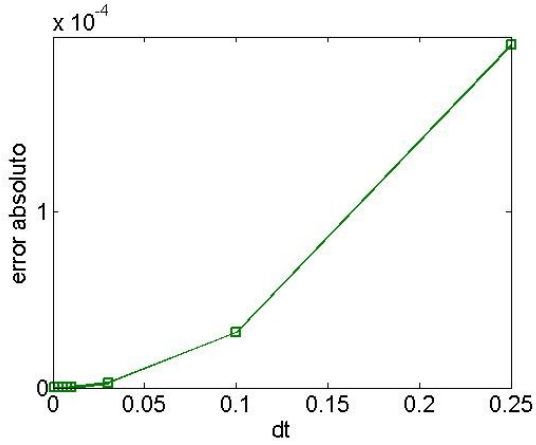
```
function y=euler1(tsteps)
t_f=1;               % valor final de tiempo
dt=t_f/tsteps;       % el paso de tiempo
t=0:dt:t_f;          % definimos los diferentes tiempos
y=0.2;               % definimos dato inicial
for n=1:tsteps
    y=y+dt*y*(1-y);
end                  %metodo de Euler explicito
```

Ahora podemos hacer una gráfica de la precisión obtenida (comparamos con el valor exacto, $\frac{e}{4+e}\approx 0{,}404609675$) en función del paso temporal *dt*.

```
yexacto=exp(1)/(4+exp(1));
pasos=[];errores=[];      %vectores con los pasos temporales y los
                          %errores
for k=[4 10 30 100 300 1000]
    pasos=[pasos, 1/k];   %engordamos los vectores sobre la marcha
    errores=[errores, abs(euler1(k)-yexacto)];
end
plot(pasos, errores)
```

En la siguiente gráfica a la izquierda puedes ver el resultado, mientras que a la derecha se muestra la gráfica correspondiente a un método de Runge-Kutta de orden 2 (el método del *punto medio*). Como puedes ver, la convergencia del método de Euler es lineal mientras que la del método del punto medio es cuadrática (la gráfica es aproximadamente una parábola).



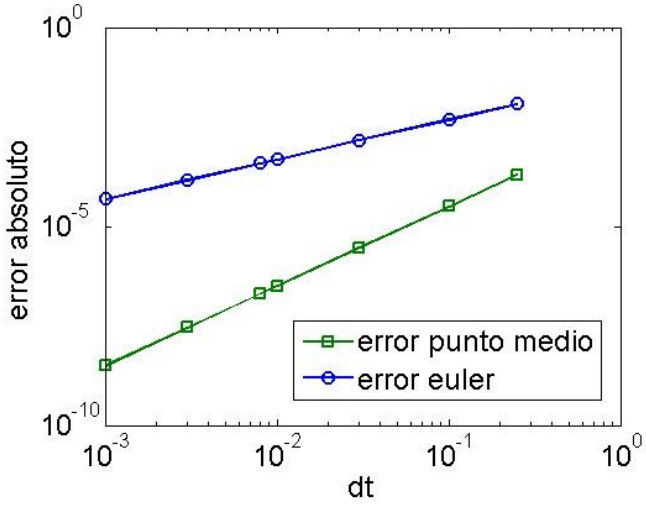


En general, para estimar el orden de convergencia de un método, se escribe el error como una potencia del paso de tiempo:

$$error = C \cdot dt^{\alpha}$$

y se toma logaritmos para obtener una ecuación lineal:

$$\log(error) = \alpha \cdot \log(dt) + \log(C)$$



Al aplicar este procedimiento al error del método de Euler, se obtiene una recta (a la derecha, en azul) con pendiente prácticamente igual a1. Es decir, $\alpha \approx 1$, mientras que si usamos el error del método de Runge-Kutta, se obtiene una recta (a la derecha, en verde) con pendiente casi igual a 2. En este caso $\alpha \approx 2$, lo que significa que el error depende del paso de forma cuadrática.

2.3 Resolución de un sistema de EDOs.

Vamos a intentar ahora resolver aproximadamente las ecuaciones de Lorentz. Se trata de un sistema no lineal caótico con lo que se denomina un *atractor extraño*. Sin embargo, es posible obtener soluciones aproximadas con cualquiera de los métodos anteriores. Estudiamos el sistema de ecuaciones:

$$\frac{dx}{dt} = \sigma(y - x)$$

$$\frac{dy}{dt} = x(\rho - z) - y$$

$$\frac{dz}{dt} = xy - \beta z$$

para los valores de los parámetros $\sigma = 10,\ \rho = 28,\ \beta = 8/3$ y dato inicial *x(0)=y(0)=z(0)=1*. Como ejercicio, puedes intentar resolver la ecuación aproximadamente con el método de Euler. Para utilizar uno de los métodos numéricos de Matlab, creamos un fichero de Matlab (File/New/M-file) con el nombre *lorentz.m* y escribimos:

```
function v=lorentz(t,x)
sigma=10;      %parametros
rho=28;
beta=8/3;
v=[sigma*(x(2)-x(1));x(1)*(rho-x(3))-x(2);x(1)*x(2)-beta*x(3)];
```

y después escribimos en la terminal:

```
t_0=0;
t_f=40;
v0=[1;1;1];
[t,v]=ode45('lorentz',[t_0,t_f],v0);
plot3(v(:,1),v(:,2),v(:,3))
```

2.4 Resolución de un sistema *stiff*.

Estudiaremos ahora un sistema lineal mucho más sencillo pero que muestra las debilidades de métodos explícitos como el de Euler:

$$\frac{dx}{dt} = -x$$

$$\frac{dy}{dt} = -10^3 y$$

con dato inicial *x(0)=y(0)=1*. Ponemos las dos variables juntas en un mismo vector **v** para escribir el sistema de ecuaciones en forma matricial:

$$\begin{pmatrix} \frac{dx}{dt} \\ \frac{dy}{dt} \end{pmatrix} = \begin{pmatrix} -1 & 0 \\ 0 & -10^3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

$$\frac{dv}{dt} = M\, v$$

y aplicamos el método de Euler explícito:

```
tsteps=1000;              % Subdivisiones del intervalo
t_f=10;             % valor final de tiempo
dt=t_f/tsteps;                % el paso de tiempo
t=0:dt:t_f;         % definimos los diferentes tiempos
v=zeros(2,tsteps+1);
v(:,1)=[1;1];       % definimos el dato inicial
M=[-1 0 ; 0 -10^3];
A=eye(2)+dt*M;
for n=1:tsteps
   v(:,n+1)=A*v(:,n);
end                 %metodo de Euler
plot(t,v)           %dibuja la posicion y la velocidad
```

Repitiendo lo anterior para distintos valores de **tsteps** observamos comportamientos muy distintos. Para obtener resultados razonables, es necesario que **tsteps** valga al menos 10000, un valor excesivamente grande para una ecuación tan sencilla. Para valores de **tsteps** menores que 5000, el resultado es catastrófico. Sin embargo, la ecuación es tan sencilla que se puede integrar de forma exacta:

$$x(t)=e^{-t}$$
$$y(t)=e^{-10^3 t}$$

Es claro que la ecuación debería converger a *(0,0)*, y bastante deprisa. También es fácil calcular la solución exacta del método de Euler explícito:

$$x_n=(1-dt)^n x_0$$
$$y_n=(1-10^3\,dt)^n y_0$$

Ahora podemos explicar el comportamiento observado: para $dt>2\cdot10^{-3}$: $|y_n|$ tiende a infinito en *n*.
Probamos ahora el método de Euler implícito, que se comporta mucho mejor que los métodos explícitos con ecuaciones de tipo *stiff*. Este método se basa en la aproximación:

$$v(t+dt)\approx v(t)+dt\cdot v'(t+dt)$$

para un valor de *dt* pequeño. Para empezar, tomamos un conjunto de instantes de tiempo $t_n=t_0+n\cdot dt$ y definimos $v_0=v(0)$. A continuación, definimos el valor de los puntos v_n de forma recursiva, mediante el esquema numérico:

$$v_{n+1}=v_n+dt\cdot M\cdot v_{n+1}$$
$$v_{n+1}=(I-dt\cdot M)^{-1}v_n$$

Adaptamos el código del esquema de Euler explícito para que utilice el método implícito:

```
tsteps=1000;          % Subdivisiones del intervalo
t_f=10;               % valor final de tiempo
dt=t_f/tsteps;        % el paso de tiempo
t=0:dt:t_f;           % definimos los diferentes tiempos
v=zeros(2,tsteps+1);
v(:,1)=[1;1];         % definimos el dato inicial
M=[-1 0 ; 0 -10^3];
A=(eye(2)-dt*M)^(-1);
for n=1:tsteps
    v(:,n+1)=A*v(:,n);
end                   %metodo de Euler implicito
plot(t,v)             %dibuja la posicion y la velocidad
```

Ahora los resultados son más razonables, incluso para valores de **tsteps** tan pequeños como 10.
Observemos la solución exacta del método numérico:

$$v_n=(I-dt\cdot M)^{-n}v_0$$

$$x_n=\frac{1}{(1+dt)^n}x_0$$
$$y_n=\frac{1}{(1+10^3\,dt)^n}y_0$$

En este caso para cualquier valor de *dt* la solución converge al vector *(0,0)* de forma exponencial, al igual que la solución exacta. En general, los métodos implícitos son más estables que los explícitos. Incluso métodos numéricos más sofisticados como los de **ode23** u **ode45** no dan buen resultado con ecuaciones de tipo stiff. Al ser métodos adaptativos, reducen su paso de tiempo lo suficiente como para dar un buen resultado, pero a costa de necesitar muchas más operaciones que un método implícito. Por contra, los métodos implícitos tienen en su contra que no son fáciles de aplicar a problemas no lineales. Además, cuando se aplican a problemas lineales pero con un vector de dimensión muy grande, requieren mucho tiempo de cómputo, pues hay que o bien invertir una matriz grande, o al menos resolver un sistema lineal de ecuaciones en cada paso. Para implementar esta última opción en el código anterior, cambiaríamos el código del bucle principal de este modo:

```
A=(eye(2)-dt*M);
for n=1:tsteps
    v(:,n+1)=A\v(:,n);
end                 %metodo de Euler implicito
```

Aunque en este caso no vamos a percibir la diferencia, ésta puede ser importante en sistemas de dimensión alta. Veremos en el siguiente capítulo de las prácticas que este problema surge de forma natural al resolver ecuaciones en derivadas parciales.