Programación: métodos de un paso para sistemas de ecuaciones diferenciales ordinarias

Objetivos. Programar algunos métodos de un paso (el método de Euler y algunos métodos de Runge–Kutta explícitos o implícitos) para resolver sistemas de ecuaciones diferenciales ordinarias con condiciones iniciales. Hacer comprobaciones y comparar la eficiencia de varios métodos.

Requisitos. Haber programado varios métodos de un paso para EDO con valores iniciales.

1. Ejemplo. Por simplicidad, consideramos un sistema de dos ecuaciones diferenciales con una condición inicial (para ambas componentes), cuya solución es muy conocida e importante:

$$y'(t) = -z(t), \qquad z'(t) = y(t), \qquad y(0) = 1, \qquad z(0) = 0.$$

Escribimos el sistema en la forma vectorial:

$$\begin{bmatrix} y'(t) \\ z'(t) \end{bmatrix} = \begin{bmatrix} -z(t) \\ y(t) \end{bmatrix}, \qquad \begin{bmatrix} y(0) \\ z(0) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

En el programa guardamos el vector con componentes y, z como un renglón. Programamos la función `ftrig` que corresponde al lado derecho de la ecuación.

```
function [p] = ftrig(t, u),
   p = [ -u(2), ??? ];
end
```

Denotemos por `xtrig` a la solución exacta del problema. Dada una columna `t`, la función `xtrig` regresa una matriz de dos columnas:

```
function [v] = xtrig(t),
   v = [ unafuncionmuyconocida(t), otrafuncionmuyconocida(t) ];
end
```

2. Varios esquemos para hacer un paso. En las clases pasadas programamos varios métodos de un paso. Por ejemplo,

```
defun eulerstep(f, t, v, h):
   v + ??? * f(???, ???)
```

También tenemos varios métodos de Runge–Kutta explícitos e implícitos. Se recomienda tener disponibles las funciones correspondientes. Vamos a usarlas sin cambios.

3. Versión vectorial del esquema para métodos de un paso. Ahora `x0` es un vector; denotamos su longitud por d. El resultado `v` de la función debe ser una matriz de tamaño $(n+1) \times d$. El arreglo `t` sigue siendo unidimensional.

```
defun onestepmethodvec(f, tmin, tmax, x0, onestepformula, n):
   d <- length(???);  v <- zeroarray(???, ???); v[0, :] <- x0
   h <- ???; t <- ???; # igual que antes
   for j in range(n):
      v[j + 1, :] <- onestepformula(???, ???, ???, ???)
   (t, v)
```

4. Pruebas.

```
defun testmethodvec(onestepformula, n):
   tmin <- 0; tmax <- 7; x0 <- [1, 0]
   starttime <- currenttime()
   (t, xapprox) <- onestepmethodvec(ftrig, ???, ???, ???, ???, ???)
   elapsedtime <- currenttime() - starttime
   xexact <- xtrig(t)
   maxerror1 <- max(abs(xexact[:, 1] - xapprox[:, 1]))
   maxerror2 <- max(abs(xexact[:, 2] - xapprox[:, 2]))
   maxerror <- max(maxerror1, maxerror2)
   (elapsedtime, maxerror)
```

Se recomienda ejecutar esta función con varios valores de n ($n = 10, 100, \ldots$) y observar el comportamiento del error y del tiempo:

```
testmethodvec(eulerstep, 10)
```

Luego haga pruebas similares con otros métodos de un paso: Heun y Ralston, Runge–Kutta de órdenes 3, 4, 5, y Runge–Kutta implícitos.

5. Gráficas. Graficamos las dos componentes de la solución, como funciones de la variable t, en la sintaxis de Matlab/Octave:

```
function [] = plotsolution1(),
   tmin = 0; tmax = 7; x0 = [1, 0]; n = 100;
   [t, xapprox] = onestepmethodvec(@ftrig, ???, ???, ???, @rk41formula, n);
   plot(t, xapprox(:, 1), t, xapprox(:, 2));
end
```

También hacemos una gráfica en el plano fase:

```
function [] = plotsolution2(),
   tmin = 0; tmax = 7; x0 = [1, 0]; n = 100;
   [t, xapprox] = onestepmethodvec(@ftrig, ???, ???, ???, @rk41formula, n);
   y = xapprox(:, 1); z = xapprox(:, 2);
   plot(y, z);
end
```