Programación: un par de métodos de Runge–Kutta de tercer order para resolver EDO's con valores iniciales

Objetivos. Programar un par de métodos de Runge–Kutte de tercer order para resolver ecuaciones diferenciales ordinarias con condiciones iniciales. Hacer comprobaciones y comparar la eficiencia de los métodos.

Requisitos. Haber programado y probado el método de Euler.

1. Ejemplo. Se puede usar el ejemplo de la clase anterior: $x'(t) = \dfrac{2x(t)}{3+t}$, $x(0) = 1$. Se sabe que la solución exacta es $x(t) = \dfrac{1}{9}(3+t)^2$. Ya hemos programado una función `f1` de dos argumentos `t`, `x` que regresa el valor del lado derecho de la ecuación, y una función `x1` que calcula los valores de la solución exacta.

2. Un paso del método RK31 y un paso del método RK32. Programamos funciones que calculan una aproximación del valor x_{j+1} a partir de t_j, x_j y h, usando la función `f`:

```
defun rk31step(f, t, v, h):
   k1 <- h * f(???, ???) // lo mismo que en el método de Euler
   k2 <- h * f(t + h / 2, v + k1 / 2)
   k3 <- h * f(t + h, v - k1 + 2 * k2)
   v + (k1 + 4 * k2 + k3) / 6

defun rk32step(f, t, v, h):
   k1 <- ??? // lo mismo que antes
   k2 <- h * f(t + h / 3, v + k1 / 3)
   k3 <- h * f(t + 2 * h / 3, v + 2 * k2 / 3)
   v + (k1 + 3 * k3) / 4
```

3. Esquema general de los métodos de un paso. Si no lo han hecho antes, sugiero programar ahora el **esquema general** de los métodos de un paso:

```
defun onestepmethod(f, tmin, tmax, x0, onestepformula, n):
   v <- zeroarray(n + 1); v[0] <- ???; h <- ???
   t <- tmin + h * range(n + 1)
   for j in range(n):
      v[j + 1] <- onestepformula(???, ???, ???, ???)
   (t, v)
```

Se supone que el argumento `onestepformula` es el apuntador a una función del mismo tipo que las funciones `rk31step` y `rk32step`. Se supone que la función interna `range` construye y devuelve el arreglo de los números $0, 1, \ldots, n-1$. En algunos lenguajes hay otras funciones para generar progresiones aritméticas.

4. Pruebas.

```
defun testmethod(onestepformula, n):
   (tmin, tmax, x0) <- (0, 1, 1)
   starttime <- currenttime()
   (t, xapprox) <- onestepmethod(f1, ???, ???, ???, onestepformula, n)
   elapsedtime <- currenttime() - starttime
   maxerror <- max(abs(x1(t) - xapprox))
   (elapsedtime, maxerror)
```

Se recomienda ejecutar esta función con varios valores de n ($n = 10, 100, \ldots, 1000000$) y observar el comportamiento del error y del tiempo:

```
testmethod(rk31step, 10)
testmethod(rk31step, 100)
...
```

En algunos lenguajes se utiliza un operador especial para crear el apuntador a una función. Por ejemplo, en Matlab y GNU Octave se puede escribir `testmethod(@rk31step, 10)`. Luego haga pruebas similar para el método RK32.

5. Análisis del tiempo de ejecución. En este ejercicio elegimos n de tal manera que el tiempo de ejecución sea al menos 0.1 segundos. ¿Cómo se cambia el tiempo de ejecución cuando n se multiplica por 10?

6. El número de evaluaciones de la función f. ¿Cuántas evaluaciones de f se necesitan en un paso del método RK31? ¿Y en un paso del método de RK32? ¿Cuántas veces se evalúa f, si el intervalo está dividido en n partes?

7. El comportamiento del error para los métodos RK31 y RK32. Calcule los errores ε_n para $n = 10, 10^2, \ldots, 10^5$:

```
defun testerror():
   ns <- [10, 100, 1000, 10000, 100000]; eps <- zeroarray(6)
   for j in range(6):
      eps[j] <- testmethod(rk31step, ns[j])[1]
   eps
```

También calcule y muestre los cocientes $\dfrac{\varepsilon_{10}}{\varepsilon_{100}}$, $\dfrac{\varepsilon_{100}}{\varepsilon_{1000}}$, $\ldots$, $\dfrac{\varepsilon_{10000}}{\varepsilon_{100000}}$.
¿Cómo se cambia el error cuando n se multiplica por 10?
Haga pruebas similares para el método RK32.