Ecuaciones diferenciales y en derivadas parciales en Matlab

PROBLEMAS DE VALOR INICIAL EN ECUACIONES DIFERENCIALES ORDINARIAS (ODEs)

Matlab tiene muchas funciones internas para aproximar las soluciones de ecuaciones diferenciales ordinarias con problemas de valor inicial (PVIs):

$$\frac{d}{dt}y(t) = f\big(t, y(t)\big), \qquad y(t_0) = y_0$$

donde t es un escalar, $y(t)$ es de dimensión m y la función derivada f también es de dimensión m.

La forma más simple de usarlas consiste en crear una función Matlab que evalúe f y luego llamar a una de las rutinas de Matlab para resolver el PVI.

Argumentos necesarios para las rutinas: al menos hay que darle el nombre de la función *fun*, el rango de valores de t para los que se quiere la solución y el vector inicial y_0:

[T,Y] = *rutina (@fun*, intervt, y0)

donde:

- *fun* tiene que estar definida en su fichero-m como
  function f = fun(t,y)
  con f e y vectores de dimensión m y t escalar.
- El intervalo de integración se ha de definir como
  intervt = [t0 tf]
  Si se pone intervt=[t0,t1,t2,…,tf] (ordenados) la rutina dará soluciones en dichos puntos específicos.
- y0 es un vector de la misma dimensión que la de f, que almacena el valor inicial del PVI.
- Si intervt=[t0 tf]: T = vector columna que almacena los valores de t que toma el algoritmo durante toda la integración a lo largo del intervalo.
  Si intervt=[t0,t1,t2,…,tf]: T almacena sólo los puntos pedidos [t0,t1,t2,…,tf].
- Y = matriz cuya n-ésima fila contiene la solución numérica obtenida en el tiempo T(n).

Argumentos opcionales: especifican más datos sobre el problema y la precisión a la que debe calcularse la solución.

[T,Y] = *rutina (@fun*, intervt, y0, opciones)

opciones = odeset ('opc1',valor1,'opc2',valor2,…)

donde ‘opc1’, ‘opc2’,… son los nombres de las opciones que queremos cambiar y valor1, valor2,… son los valores que les vamos a dar a las nuevas opciones.
Estas opciones dependen de cada rutina pero las dos opciones que vamos a usar nosotros son comunes a todas: la elección de una tolerancia para el error absoluto y una tolerancia para el relativo. En cada paso, la rutina estima el error local err en la componente i-ésima de la solución. Este error debe ser menor o igual a un error aceptable que viene dado en función de una tolerancia específica para el error relativo RelTol y una para el error absoluto AbsTol, de tal manera que

$$|err(i)| < \max \{ RelTol\ |y(i)|, AbsTol \}$$

Por defecto, se tiene

$$AbsTol = 10\text{-}6, \qquad RelTol = 10\text{-}3.$$

Para cambiarlos hay que poner por ejemplo

opciones = odeset (‘AbsTol’, 1e-8, ’RelTol’, 1e-6);

Para más opciones ver la ayuda del programa (**doc odeset**).

Rutina	Problemas	Precisión	Tipo de algoritmo
ode45	No stiff	Media	Basado en el par explícito Runge-Kutta DOPRI(4,5). Se recomienda como primer intento: si falla se pasa a uno de los otros
ode23	No stiff	Baja	Basado en un par RK explícito (2,3) de Bogacki y Shampine. Suele ser mejor que ode45 el tolerancias grandes con presencia de cierta stiffness.
ode113	No stiff	Baja-alta	Basado en los MLMs explícitos de Adems-Bashforth-Moulton PECE, con estrategia de orden variable de 1 a 13. Es más eficiente que ode45 en tolerancias bajas y cuando la función derivada es muy costosa de evaluar.
ode15s	Stiff	Baja-media	Basado en MLMs implícitos de tipo BDF con órdenes variables de 1 a 5. Útil cuando se sospecha que el PVI es stiff o cuando el ode45 falla.
ode23s	Stiff	Baja	Basado en un RK semi-implícito de tipo Rosenbrock de orden 2. Suele ser más eficiente que ode15s en tolerancias altas, y resuelve algunos tipos de problemas stiff con los que ode15s no es bueno.
ode23t	Stiff moderado	Baja	Es una implementación de la regla trapezoidal (RK implícito de 2 etapas). Se usa sólo cuando el problema es moderadamente stiff.
ode23tb	Stiff	Baja	Una implementación que combina un RK implícito y un BDF de orden 2. Como el ode23s es más eficiente que ode15s con tolerancias altas en muchos problemas.

También se puede escribir

sol = *rutina (@fun*, intervt, y0, opciones)

La rutina devuelve lo que se llama una *estructura* de Matlab a la que se le pone el nombre **sol**. Sin entrar en mucho detalle en lo que es una estructura la idea básica es que es un array cuyos elementos son de distintos tipos y tamaños: vectores, matrices, cadenas de caracteres, etc. Así, **sol** es una estructura que que en este caso almacena el vector de los pasos en t y la matriz con las correspondientes soluciones numéricas. Para obtener dichos datos:

sol.x	vector fila que almacena los pasos en t (= T traspuesto)
sol.y	matriz que almacena en cada columna sol.y(:,n) la solución en el tiempo T(n) (= Y traspuesta)

La ventaja de esta alternativa es que a partir de ella podemos usar el comando **deval** que dada **sol** nos dará una aproximación a la solución en cualquier punto intermedio:

ysol = deval(sol,ts)

donde ts es un punto o un rango de puntos de t en los que quiero evaluar la solución. Tiene que caer en el intervalo [sol.x(1),sol.x(end)]. También se puede poner opcionalmente

ysol = deval(sol,ts,ind)

donde ind=1,2,…,m, con lo que sólo devuelve la componente ind de la solución numérica.

PROBLEMAS DE VALORES DE CONTORNO EN ECUACIONES DIFERENCIALES ORDINARIAS (BVPs)

La función **bvp4c** de Matlab integra problemas de valores de contorno en dos puntos del tipo

$$\frac{d}{dt}y(t) = F\big(t, y(t)\big), \qquad g(y(a), y(b)) = 0$$

mediante un método de colocación. Al igual que en el caso anterior t es un escalar que varía en *[a, b], y(t)* es de dimensión m y la función derivada F también es de dimensión m. Estos problemas son bastante más complejos de aproximar que los PVIs, principalmente porque no se puede garantizar la existencia de solución en general ni su unicidad cuando la hay. Por eso esta rutina necesita que se le aporte previamente una aproximación inicial a la solución. Esta solución aproximada inicial y la solución final que obtiene el método se almacena en estructuras.

Se llama a esta función mediante la sentencia

sol = **bvp4c** *(@fun*, *@funcont*, **solinic**, opciones)

donde

- En la función *fun* se le aporta la función derivada F. Tiene que estar definida igual que con los PVIs

  function f = fun(t,y)

  con f e y vectores columna de dimensión m y t escalar.
- *funcont* es la función que computa las condiciones de contorno y devuelve el vector columna *g* con la sintaxis:

  function g=funcont(ya,yb)

siendo ya e yb vectores columna que se corresponden con $y(a)$ e $y(b)$ respectivamente. Hay que tener en cuenta que las condiciones de contorno hay que escribirlas igualadas a 0. Por ejemplo, si las condiciones fueran y(0)=1, y(1)=-1, tendríamos que escribir

g=[ya(1)-1; yb(1)+1];

- **solinic** es una estructura que almacena la solución inicial que se necesita usando el comando

  solinic=**bvpinit**(linspace(a,b,n),@*funinic*);

  Con esta sentencia se llama a la rutina **bvpinit** que evalúa la función *funinic* en n puntos igualmente espaciados del intervalo $[a,b]$ y los almacena en la estructura **solinic**. La función *funinic* es la que aporta la aproximación inicial a la solución y tiene que escribirse como

  function yinic=*funinic*(t)

  La elección de esta solución inicial es muy importante para el buen funcionamiento del algoritmo. Hay otras formas de computar dicha estructura **solinic** pero por ahora nos quedaremos con la más básica que es la que usa **bvpinit**.
- Con opciones se introducen otros datos para afinar el comportamiento de la rutina en cada problema. Se establecen previamente con la sentencia

  opciones = **bvpset** ('opc1',valor1,'opc2',valor2,…)

  La opción de cambiar la tolerancia del error relativo y del absoluto es igual que con **odeset**. Para conocer todas las demás opciones posibles se puede ver la ayuda del programa (**doc bvpset**).

El argumento de salida **sol** es una estructura que contiene la solución numérica obtenida:

sol.x	vector fila que almacena los puntos t en los que se ha computado la solución numérica
sol.y	matriz que almacena en cada columna **sol.y**(:,i) la solución en el tiempo **sol.x**(i)
sol.yp	matriz que almacena en cada columna **sol.yp**(:,i) una aproximación a la derivada primera de la solución en el tiempo **sol.x**(i)

Si se quiere evaluar la solución en otros puntos t's distintos a los de **sol.x** se debe usar **deval** como con los PVIs.

También se pueden resolver problemas de la forma

$$\frac{d}{dt}y(t) = F(t, y(t), p), \qquad g(y(a), y(b), p) = 0$$

donde p es un vector de parámetros que aparece en ciertos problemas de valores de contorno como los problemas de autovalores pero no lo veremos por ahora.

ECUACIONES EN DERIVADAS PARCIALES (PDEs)

Matlab tiene una rutina que integra ecuaciones en derivadas parciales de tipo parabólico (por ejemplo, la ecuación del calor) y de tipo elíptico (por ejemplo, la ecuación de Laplace) unidimensional:

$$c\left(x,t,u,\frac{\partial u}{\partial x}\right)\frac{\partial u}{\partial t}=\frac{1}{x^m}\frac{\partial}{\partial x}\left(x^m f\left(x,t,u,\frac{\partial u}{\partial x}\right)\right)+\ s\left(x,t,u,\frac{\partial u}{\partial x}\right)$$

donde $a \le x \le b$, $t_0 \le t \le t_f$. La incógnita u(x,t) puede ser vectorial. El entero m sólo puede ser 0, 1 ó 2 (simetría del problema). La función c es una matriz diagonal cuyos elementos son o ceros o positivos. Cada elemento nulo nos da una ecuación elíptica, mientras que un elemento positivo nos aporta una ecuación parabólica. Debe haber al menos una ecuación parabólica ($c \neq 0$).

Los valores iniciales y de frontera tienen que darse de la siguiente forma:

Valores iniciales	$u(x,t_0) = u_0(x)$ con u_0 dada en una función aparte *pdeini*
Valores frontera	para $x=a$ y $t_0 \le t \le t_f$, la solución debe verificar: $$p_a(x,t,u)+q_a(x,t)f\left(x,t,u,\frac{\partial u}{\partial x}\right)=0$$ para $x=b$ y $t_0 \le t \le t_f$, la solución debe verificar: $$p_b(x,t,u)+q_b(x,t)f\left(x,t,u,\frac{\partial u}{\partial x}\right)=0$$ con funciones p_a, q_a, p_b, q_b dadas en una función aparte *pdefron*

Para llamar a esta rutina **pdepe** tenemos que escribir

sol=**pdepe**(m,@pdefun,@pdeini,@pdefron,xmesh,intervt)

donde

- m= potencia de x que aparece, que puede tomar los valores 0, 1 ó 2.
- pdefun es la función que computa las funciones c, f y s de la ecuación, que tiene que tener la forma

  function [c,f,s] = pdefun(x,t,u,DuDx)

  en la que solo se expresa c, f y s en función de x, t, u y DuDx que representa a ∂u/∂x, y que él computará cuando lo necesite.
- pdeini es la función que almacena la función inicial para $t=t_0$, que tiene que ponerse como

  function u0 = pdeini(x)
- pdefron almacena los datos de frontera, es decir, devuelve las funciones p_a, q_a, p_b y q_b del problema

  function [pa,qa,pb,qb] = pdefron(xa,ua,xb,ub,t)
- xmesh es el vector que almacena la discretización espacial ordenada en [a,b] que queramos, a condición que xmesh(1)=a y xmesh(end)=b. Para ello se suele usar

  xmesh = linspace(a,b,n)

  donde n lo elegimos según el problema. Tiene mucha influencia en la eficiencia y la precisión de la integración. Este algoritmo usa una discretización espacial de segundo orden. Cuando el problema varía rápidamente respecto a x es necesario que n sea grande, pero cuando

no es así es mejor elegir valores de n moderados para evitar un coste computacional excesivamente alto.
- intervt es el vector que especifica los valores de t en los que se debe computar la solución, entendiéndose siempre que intervt(1)=t_0 e intervt(end)=t_f y intervt(i)<intervt(i+1), al igual que en las rutinas odes.

Como ya hemos dicho arriba este algoritmo combina una discretización espacial (respecto x) de segundo orden y luego integra temporalmente con el mismo código que la rutina **ode15s** (métodos BDF con orden variable de 1 a 5) a paso variable.

También existe la posibilidad de poner otras opciones con **odeset**, pero en este caso no suele ser necesario ni suele aportar mejores resultados. Hay que tener en cuenta que en el ámbito de las ecuaciones en derivadas parciales no se trabaja en el mismo rango de precisión que con las ecuaciones diferenciales ordinarias: si se encuentran soluciones con errores menores que 10^{-2} ó 10^{-3} ya podemos darnos por satisfechos, y más aún cuando trabajamos con métodos genéricos para un amplio abanico de problemas como es el caso. Normalmente sólo se encuentra una mayor precisión cuando diseñamos un algoritmo específico para un problema particular.

En este caso la rutina nos devuelve **sol** que ahora es un array de tres componentes en el que cada elemento sol(i,j,k) almacena la aproximación a la k-ésima componente de la solución u(x,t) en el punto x=xmesh(j) y t=intervt(i).

Además tenemos el comando **pdeval** que se usa para evaluar u(t,x) y $\partial u/\partial x$ en otros puntos. Para ver cómo funciona ver la documentación (doc pdeval).

ECUACIONES DIFERENCIALES CON RETARDO (DDEs)

Matlab tiene una función propia, **dde23**, que resuelve numéricamente problemas de valor inicial de ecuaciones diferenciales con retardo del tipo

$$\frac{d}{dt}y(t) = f\big(t, y(t), y(t-\tau_1), y(t-\tau_2), \dots, y(t-\tau_k)\big), \quad t > t_0$$

con $\tau_1, \tau_2, \dots, \tau_k$ son los retardos constantes positivos conocidos. En lugar de una condición inicial simple estos problemas necesitan de una *función inicial* dada:

$$y(t) = S(t), \quad t \le t_0$$

La forma de llamar a esta función es

sol = **dde23***(@ddefun*, retardos, historia, intervt, opciones)

donde:

- *ddefun* es la función derivada de la ecuación que tiene que estar definida en su fichero-m como

function f = ddefun(t,y,Z)

con f e y vectores de dimensión m y t escalar. La variable Z es un array mxk cuya columna j contiene al vector $y(t-\tau_j)$.

- retardos es un vector fila que almacena los retardos de la ecuación, es decir

  retardos(j)= τ_j.

- historia es donde le damos la función inicial S(t) del PVI. Como en muchas aplicaciones esta función suele ser constante, Matlab nos permite dos posibilidades para introducir esta función:
  - Si S(t)=S un vector constante, podemos poner directamente S como un vector columna en dde23.
  - Si S(t) es una función de t no constante habrá que definirla en otra función

    function s=historia(t)

  e introducirla luego como dde23(…,@historia,…).
- El intervalo de integración intervt sigue las mismas reglas que con las rutinas odes.
- En este caso, las opciones se introducen mediante el comando:

  opciones = **ddeset**('opc1',valor1,'opc2',valor2,…)

  ddeset tiene opciones iguales que para las odes, como las del error absoluto y relativo, y otras diferentes específicas para estas ecuaciones. A nosotros por ahora nos bastará con las opciones del error.

Con esta sentencia Matlab aplica una implementación del mismo par (2,3) RK explícito de Bogacki y Shampine que usa la rutina **ode23**, usando interpolación para extraer la salida densa que necesita en cada paso, además de otras modificaciones específicas para este tipo de problemas.

Así Matlab nos devuelve en la estructura **sol** los datos de la solución numérica de una forma análoga que con las odes:

sol.x	vector fila que almacena los pasos en t
sol.y	matriz que almacena en cada columna sol.y(:,n) la solución en el tiempo T(n)

Y de la misma forma se puede usar **deval** para obtener valores de la solución en los puntos que deseemos.

Matlab también tiene otra rutina **ddesd** para problemas de valor inicial con retardo cuando dicho retardo no es constante, es decir, $\tau=\tau(t)$, que usa el RK clásico explícito de 4 etapas y orden 4, usando una implementación a paso variable. Pero cuando los retardos son constantes es mejor usar **dde23**.