

CÓDIGOS EN OCTAVE/MATLAB
SEMINARIOS Y PRÁCTICAS
MÉTODOS MATEMÁTICOS APLICADOS
A LA INGENIERÍA

ASIGNATURAS:
MÉTODOS MATEMÁTICOS APLICADOS A LA ING. DE
MATERIALES
MÉTODOS MATEMÁTICOS APLICADOS A LA ING. DE LA
ENERGÍA
MÉTODOS NUMÉRICOS (MÓDULO I) EN EL MÁSTER EN ING.
INDUSTRIAL

A. I. Muñoz Montalvo, A. Nolla y E. Schiavi
Septiembre 2022

©2022. Autores: A. I. Muñoz Montalvo, A. Nolla, E. Schiavi.
Algunos derechos reservados.

Este documento se distribuye bajo la licencia internacional
Creative Commons Attribution-ShareAlike 4.0 International
License.

Disponible en: <http://creativecommons.org/licenses/by-sa/4.0/>

Publicado en: <https://burjcdigital.urjc.es>

Índice

1. CÓDIGOS EN OCTAVE/MATLAB	2
1.1. Códigos para resolver ecuaciones no lineales	2
1.1.1. Método de bisección	2
1.2. Método del punto fijo	3
1.2.1. Método Aitken	4
1.2.2. Método de Newton Raphson para una ecuación de una incógnita	5
1.2.3. Método de la secante	6
1.2.4. Método de regulafalsi	6
1.2.5. Método de Newton Raphson para un sistema de ecuaciones	7
1.3. Códigos para resolver problemas de valor inicial	9
1.3.1. Método de Euler explícito	9
1.3.2. Método de Euler implícito	9
1.3.3. Método de Crank-Nicolson	10
1.3.4. Método de Heun	11
1.3.5. Método de Simpson	12
1.3.6. Ejemplo de resolución de un sistema de edos con E. explíci- to. Ejercicio 3.30 del libro de problemas	12
1.4. Códigos para resolver problemas de contorno	13
1.4.1. Método para resolver un problema de contorno 1D con condiciones Dirichlet	13
1.4.2. Método para resolver un problema de contorno 1D con condiciones mixtas	13
1.4.3. Método para resolver la ecuación del calor 1D	17
1.4.4. Método para resolver la ecuación de Poisson 2D	19
2. SEMINARIOS: EJERCICIOS RESUELTOS	21
3. EJERCICIOS DE PRÁCTICAS	30

1. CÓDIGOS EN OCTAVE/MATLAB

Observaciones: Todo aquello escrito en cursiva y que no vaya después del símbolo prompt del sistema, `>`, debe ser interpretado como un comentario o explicación, y no como una línea de comandos.

La mayor parte de los códigos son adaptaciones de las funciones del libro “Cálculo científico con MATLAB y Octave” de A. Quarteroni, F. Saleri, que se pueden obtener <https://mox.polimi.it/qs/>.

1.1. Códigos para resolver ecuaciones no lineales

1.1.1. Método de bisección

```
> function[sol,itera]=metbiseccion(fecu,a,b,errorper,maxitera)
```

Este código encuentra una aproximación de una raíz en el intervalo $[a,b]$, de la ecuación $fecu=0$. Los argumentos son:

- *errorper: el error permitido.*
- *maxitera: el número máximo de iteraciones que se permiten realizar.*
- *sol: la solución numérica aproximada obtenida.*
- *itera: el número de iteraciones que ha realizado para obtener sol.*

```
> x = [a, (a+b)*0.5, b]; fx = fecu(x);
```

*Comprobamos que el intervalo elegido satisface el teorema de Bolzano, $fecu(a)*fecu(b)<0$, si no lo verifica, aparece un mensaje de error. También puede ocurrir que a o b sean ya raíces.*

```
> if fx(1)*fx(3) >0
> error([' El signo de la función en los extremos',...
' del intervalo (a,b) tiene que ser distinto']);
> elseif fx(1) == 0
> sol = a; itera = 0;
> return
> elseif fx(3) == 0
> sol = b; itera = 0;
> return
```

```

> end
> itera = 0; I = (b - a)*0.5;
> while I >= errorper && itera <maxitera
> itera = itera + 1; > if fx(1)*fx(2) <0
> x(3) = x(2);
> x(2) = x(1)+(x(3)-x(1))*0.5;
> fx = fecu(x);
> I = (x(3)-x(1))*0.5;
> elseif fx(2)*fx(3) <0
> x(1) = x(2);
> x(2) = x(1)+(x(3)-x(1))*0.5;
> fx = fecu(x);
> I = (x(3)-x(1))*0.5;
> else
> x(2) = x(find(fx==0)); I = 0;
> end
> end
> if(itera==maxitera && I >errorper)
> fprintf(['El método no converge,'...
> 'se ha llegado al número máximo de iteraciones ',...
> 'sin estar por debajo del error máximo permitido']);
> end
> sol = x(2);

```

1.2. Método del punto fijo

```

> function[sol,itera]=metpuntofijo(g,x0,errorper,maxitera)

```

Este método iterativo encuentra una aproximación del punto fijo de la función g . Nótese que $f(x)=0 \Leftrightarrow g(x)=x$. Los argumentos son:

- *g : función que define el esquema numérico.*
- *x_0 : semilla para iniciar el esquema iterativo.*
- *sol : solución numérica obtenida.*
- *$errorper$: el error máximo permitido. El error se mide mediante el valor absoluto de la diferencia de dos valores obtenidos en iteraciones consecutivas.*
- *$maxitera$: el número máximo de iteraciones permitido.*
- *$itera$: el número de iteraciones realizado para obtener sol , cumpliendo con el error máximo permitido.*

```

> x = x0; diferencia= errorper +1; itera =0;
> while itera <= maxitera && diferencia >= errorper
> gx = feval(g,x);
> diferenciait=x-gx;
> diferencia=abs(diferenciait);
> xnueva=gx;
> x=xnueva;
> itera=itera+1;
> end
> if itera >= maxitera
> fprintf('Se ha llegado al número máximo de iteraciones',...
> 'sin estar por debajo del máximo error permitido');
> end
> sol = x;
> return
> endfunction

```

1.2.1. Método Aitken

```

> function [sol,itera]=metodoaitken(g,x0,errorper,maxitera)

```

Este código utiliza el método de Aitken para obtener una aproximación numérica al punto fijo de una función g , utilizando como semilla x_0 . Los argumentos son:

- *g : función que define el esquema numérico.*
- *x_0 : semilla para iniciar el esquema iterativo.*
- *sol : solución numérica obtenida.*
- *$errorper$: el error máximo permitido. El error se mide mediante el valor absoluto de la diferencia de dos valores obtenidos en iteraciones consecutivas.*
- *$maxitera$: el número máximo de iteraciones permitido.*
- *$itera$: el número de iteraciones realizado para obtener sol , cumpliendo con el error máximo permitido.*

```

> x = x0; difer = errorper + 1; itera = 0;
> while itera <maxiter && difer >= errorper
> gx = g(x); ggx = g(gx);xnew =(x*ggx-gx^2)/(ggx-2*gx+x);
> difer = abs(x-xnew);
> x = xnew;
> itera = itera + 1;
> end

```

```

> if
> (itera==maxitera && difer>errorper)
> fprintf(['El método no converge en el número',...
> 'máximo de iteraciones fijado']);
> end
> sol = x;
> return

```

1.2.2. Método de Newton Raphson para una ecuación de una incógnita

```

> function[sol,itera]=metnewton1ec(fecu,dfecu,x0,errorper,maxitera)

```

Este código obtiene una solución aproximada de una raíz de la ecuación $fecu=0$, con el método de Newton-Raphson, tomando como semilla x_0 . Los argumentos son:

- *x_0 : semilla para arracar el esquema iterativo.*
- *fecu: es la función que define la ecuación de la cual queremos obtener una raíz.*
- *dfecu: derivada primera de fecu.*
- *sol: la solución numérica obtenida.*
- *errorper: el máximo error permitido en la aproximación.*
- *maxitera: el número máximo de iteraciones permitidas.*
- *itera: el número de iteraciones realizadas por el esquema para alcanzar un error por debajo del error cometido. El error se mide con el valor absoluto de $fecu(x)/dfecu(x)$.*

```

> x = x0;
> fx = fecu(x); dfx = dfecu(x); itera = 0;medidaerror = errorper+1;
> while medidaerror >= errorper && itera <maxitera
> itera = itera + 1;
> medidaerror = - fx/dfx;
> x = x + medidaerror;
> medidaerror= abs(medidaerror);
> fx = fecu(x);
> dfx = dfecu(x);
> end
> if (itera==maxitera && medidaerror >errorper)
> fprintf(['El método no converge por alcanzar el número máximo',...
> 'de iteraciones permitidas',...
> 'sin llegar a un error por debajo del error máximo permitido']);
> end

```

```
> sol = x;  
> return
```

1.2.3. Método de la secante

```
> function [sol,itera]=metsecante(fecu,a,b,errorper,maxitera)
```

Este código encuentra una aproximación de una raíz de la ecuación dada por $fecu=0$, utilizando el método de la secante. Los argumentos son:

- *a y b: semillas.*
- *sol: la solución numérica obtenida.*
- *errorper: el error máximo permitido. El error se mide sobre el valor absoluto de la diferencia de dos valores obtenidos en iteraciones consecutivas.*
- *maxitera: el número máximo de iteraciones permitido.*
- *itera: el número de iteraciones realizadas para obtener sol, cumpliendo con el error máximo permitido.*

```
> fx0=feval(fecu,a); fx1=feval(fecu,b); x0=a; x1=b;  
> r=fx1*(x1-x0)/(fx1-fx0); x2=x1-r; c=1;  
> while (abs(r)>errorper && c <maxitera)  
> x0=x1; x1=x2;  
> fx0=feval(fecu,x0); fx1=feval(fecu,x1);  
> r=fx1*(x1-x0)/(fx1-fx0);  
> x2=x1-r;  
> c=c+1;  
> end  
> if c >= maxitera  
> fprintf('No converge en el máximo',...  
> 'número de iteraciones');  
> end  
> sol = x2;  
> itera = c;  
> end
```

1.2.4. Método de regulafalsi

```
> function[sol,itera]=metregulafalsi(fecu,a,b,errorper,maxitera)
```

Este código encuentra una aproximación de una raíz de la ecuación dada por $fecu=0$, utilizando el método de regulafalsi. Los argumentos son:

- *fecu*: función que define la ecuación.
- *a* y *b* semillas.
- *sol*: la solución numérica obtenida tomando como semillas *a* y *b*.
- *errorper*: el error máximo permitido. El error se mide sobre el valor absoluto de la diferencia de dos valores obtenidos en iteraciones consecutivas.
- *maxitera*: el número máximo de iteraciones permitido.
- *itera*: el número de iteraciones realizadas para obtener *sol*, cumsubpliendo con el error máximo permitido.

```

> fx0=feval(fecu,a); fx1=feval(fecu,b); x0=a; x1=b;
> r=fx1*(x1-x0)/(fx1-fx0); x2=x1-r; c=1;
> fx2=feval(fecu,x2);
> while (abs(r)>errorper && c <maxitera)
> if fx2*fx1<0
> x0=x2; x1=x1;
> else
> x0=x0; x1=x2;
> end
> fx0=feval(fecu,x0); fx1=feval(fecu,x1);
> r=fx1*(x1-x0)/(fx1-fx0);
> x2=x1-r; fx2=feval(fecu,x2);
> c=c+1;
> end
> if c >= maxitera
> fprintf(' No converge en el máximo',...
> 'número de iteraciones ');
> end
> sol = x2; itera = c;
> end

```

1.2.5. Método de Newton Raphson para un sistema de ecuaciones

```

> function [vectorsol,itera] = metnewtonsisistema(fecusistema,...
> jacobiana,vectorx0,errorper,maxitera)

```

Este código obtiene una solución aproximada del sistema no lineal dado por el vector de funciones fecusistema, F , igualado a cero, $F=0$, partiendo de la semilla vectorx0. Los argumentos son:

- *fecusistema*: funciones que definen el sistema de ecuaciones.

- *jacobiana*: la función matriz jacobiana J .
- *fecusistema* y *jacobiana* deben ser definidas como funciones en scripts *.m*
- *vectorx0*: vector semilla.
- *errorper*: el máximo error permitido, que se mide tomando el módulo del vector $-[J(\text{vectorx})]^{-1}F(\text{vectorx})$.
- *vectorsol*: es la aproximación del vector raíz que hemos obtenido.

```

> itera = 0; medidaerror = errorper + 1; vectorx= vectorx0;
> while medidaerror >= errorper && itera <maxitera
> J = fecusistema(vectorx(1),vectorx(2));
> F = jacobiana(vectorx(1),vectorx(2));
> incremento = - inv(J)*F;
> vectorx = vectorx + incremento;
> medidaerror = norm(incremento);
> itera = itera + 1;
> end
> vectorsol=vectorx;
> if (itera==maxitera && medidaerror>errorper)
> fprintf(['El método no converge en el número máximo',...
> 'de iteraciones',...
> 'por no alcanzarse un error inferior al permitido'],F);
> end
> return

```

Función fecusistema.m. Ejemplo de implementación

```

> function F=fecusistema(x,y)
> F(1,1)=(x+y)^2-2;
> F(2,1)=x^2+1-y;
> return

```

Función jacobiana.m. Ejemplo de implementación

```

> function J=jacobiana(x,y)
> J(1,1)=2*(x+y);
> J(1,2)=2*(x+y);
> J(2,1)=2*x;
> J(2,2)=-1;
> return

```

1.3. Códigos para resolver problemas de valor inicial

1.3.1. Método de Euler explícito

```
> function[solt,soly]=eulerexplicito(f,intiempo,valorini,npasos)
```

*Este código resuelve el problema de valor inicial dado por la ecuación diferencial $y' = f(t,y)$ y el valor inicial, $y(t_0)$, denotado por *valorini*, en el intervalo temporal $[t_0, T]$ (definido en *intiempo*), con el método de Euler explícito. El resto de argumentos son:*

- *npasos*: el número de pasos que hay que dar, tomando un tamaño de discretización constante h , para llegar a T , es decir, $(T-t_0)/h$.
- *solt*: el vector que contiene los nodos temporales, $t_0, t_1, \dots, t_n = T$.
- *soly*: el vector que contiene los valores numéricos aproximados de $y(t)$, obtenidos para los distintos nodos temporales.

```
> h=(intiempo(2)-intiempo(1))/npasos;  
> solt=ones(npasos+1,1);  
> soly=ones(npasos+1,1);  
> solt(1)=intiempo(1);  
> soly(1)=valorini;  
> for i=2:npasos+1  
> solt(i)=solt(i-1)+h;  
> soly(i)=soly(i-1)+h*f(solt(i-1),soly(i-1));  
> end  
> return
```

1.3.2. Método de Euler implícito

```
> function [solt,soly]=eulerimplicito(f,intiempo,valorini,npasos)
```

Este código resuelve un problema de valor inicial $y' = f(t,y)$, $y(t_0) = y_0$, utilizando el método de Euler implícito. Los argumentos son:

- *f*: la función que define y' .
- *intiempo*: el intervalo de tiempo donde se quiere resolver el problema.
- *valorini*: el dato inicial.
- *npasos*: el número de pasos que hay que dar para llegar al tiempo final y depende del tamaño de discretización h .
- *solt* y *soly*: los vectores solución, e indican los tiempos intermedios considerados y los correspondientes valores aproximados de y .
- *Obs*: utilizamos el comando *fsolve* para resolver la posible ecuación no lineal en y que pueda aparecer al aplicar el esquema implícito.

```

> h=(intiempo(2)-intiempo(1))/npasos;
> solt=(intiempo(1):h:intiempo(2));
> soly=ones(npasos+1:1); soly(1)=valorini;
> global glob_h glob_t glob_y glob_f;
> glob_h = h; glob_y=valorini; glob_f=f;
> for i=2:npasos+1
> glob_t=solt(i);
> w = fsolve(@(w) beulerfun(w),glob_y);
> soly(i)=w;
> glob_y=w;
> end
> clear glob_h glob_t glob_y glob_f;
> end
> function[z]=beulerfun(w)
> global glob_h glob_t glob_y glob_f;
> z=w-glob_y-glob_h*feval(glob_f,glob_t,w);
> end

```

1.3.3. Método de Crank-Nicolson

```

> function[solt,soly]=cranknicolson(f,intiempo,valorini,npasos)

```

Este código resuelve un problema de valor inicial $y'=f(t,y)$, $y(t_0)=y_0$, utilizando el método de Crank Nicolson. Los argumentos son:

- *f: función que define y' .*
- *intiempo: el intervalo de tiempo donde se quiere resolver el problema.*
- *valorini: el dato inicial.*
- *npasos: el número de pasos que hay que dar para llegar al tiempo final y depende del tamaño de discretización h .*
- *solt y soly: los vectores solución, e indican los tiempos intermedios considerados y los correspondientes valores aproximados de y .*
- *Obs: utilizamos el comando fsolve para resolver la posible ecuación no lineal en y que pueda aparecer al aplicar el esquema implícito.*

```

> h=(intiempo(2)-intiempo(1))/npasos;
> solt=(intiempo(1):h:intiempo(2));
> soly=ones(npasos+1:1); soly(1)=valorini;
> global glob_h glob_t glob_t glob_y glob_f;

> glob_h = h; glob_y=valorini; glob_f=f;

> for i=2:npasos+1

```

```

> glob_t=solt(i);
> glob_t=solt(i-1);
> w = fsolve(@(w) crank(w),glob_y);
> soly(i)=w;
> glob_y=w;
> end
> clear glob_h glob_t glob_t glob_y glob_f;
> end
> function [z]=crank(w)
> global glob_h glob_t glob_t glob_y glob_f;
> z=w-glob_y-0.5.*glob_h*feval(glob_f,glob_t,w)- ...
> 0.5.*glob_h*feval(glob_f,glob_t,glob_y);
> end

```

1.3.4. Método de Heun

```

> function [solt,soly]=heun(f,intiempo,valorini,npasos)

```

Este código resuelve un problema de valor inicial $y'=f(t,y)$, $y(t_0)=y_0$, utilizando el método de Heun visto como un caso de método Runge Kutta. Los argumentos son:

- *f: la función que define y' .*
- *int tiempo: el intervalo de tiempo donde se quiere resolver el problema.*
- *valorini: el dato inicial.*
- *npasos: el número de pasos que hay que dar para llegar al tiempo final y depende del tamaño de discretización h .*
- *solt y soly: son los vectores solución, e indican los tiempos intermedios considerados y los correspondientes valores aproximados de y .*

```

> h=(int tiempo(2)-int tiempo(1))/npasos;
> soly=ones(npasos+1:1);
> solt=linspace(int tiempo(1),int tiempo(2),npasos+1);
> soly(1)=valorini;
> for i=2:npasos+1
> tn1=solt(i-1);
> tn2=solt(i);
> yn1=soly(i-1);
> yn2=soly(i-1)+h*f(tn1,yn1);
> soly(i) =soly(i-1)+ 0.5*h*(f(tn1,yn1)+f(tn2,yn2));
> end

```

1.3.5. Método de Simpson

```
> function [solt,soly]=rungekuttao3(f,intiempo,valorini,npasos)
```

Este código resuelve un problema de valor inicial $y'=f(t,y)$, $y(t_0)=y_0$, utilizando el método Runge Kutta de orden, que denominamos Simpson por la fórmula de integración numérica que usa. Los argumentos son:

- *f: la función que define y' .*
- *intiempo: el intervalo de tiempo donde se quiere resolver el problema.*
- *valorini: el dato inicial.*
- *npasos: el número de pasos que hay que dar para llegar al tiempo final y depende del tamaño de discretización h .*
- *solt y soly: los vectores solución, e indican los tiempos intermedios considerados y los correspondientes valores aproximados de y .*

```
> solt=linspace(intiempo(1),intiempo(2),npasos+1);
> h=(intiempo(2)-intiempo(1))/npasos;
> soly=ones(npasos+1:1); soly(1)=valorini;
> for i=2:npasos+1
> ti1=solt(i-1);
> ti2=solt(i-1)+0.5.*h;
> ti3=solt(i);
> yi1=soly(i-1);
> yi2=soly(i-1)+h.*f(ti1,yi1);
> yi3=soly(i-1)+h.*(-f(ti1,yi1)+2.*f(ti2,yi2));
> soly(i)=soly(i-1)+(1/6).*h.*(f(ti1,yi1)+4.*f(ti2,yi2)+...
> f(ti3,yi3));
> end
```

1.3.6. Ejemplo de resolución de un sistema de edos con E. explícito. Ejercicio 3.30 del libro de problemas

```
> odefun1=@(t,x,y) y; odefun2=@(t,x,y)-6.54.*x-0.8.*y;
> inicial1=0 inicial2=1; h=0.05; tspan=10; n=1+(tspan/h);
> t=ones(n); x=ones(n); y=ones(n);
> x(1)=inicial1; y(1)=inicial2;
> t(1)=0;
> for i=2:n;
> t(i)=h.*(i-1);
> x(i)=x(i-1)+h*odefun1(t(i-1),x(i-1),y(i-1));
> y(i)=y(i-1)+h*odefun2(t(i-1),x(i-1),y(i-1));
> end
> plot(t,x,'r'); legend('z');
```

1.4. Códigos para resolver problemas de contorno

1.4.1. Método para resolver un problema de contorno 1D con condiciones Dirichlet

```
> function [xh,uh]=bvmdirichlet(a,b,meronodos,D,V,Q,f,ua,ub)
Este código resuelve un problema de contorno unidimensional con condiciones de contorno Dirichlet:
-D u''+Vu'+Qu=f en (a,b)
con condiciones de contorno u(a)=ua, u(b)=ub, utilizando diferencias centradas. El resto de argumentos son:
```

- *meronodos*: el número de nodos incluidos los extremos del intervalo.
- *xh*: el vector que contiene los nodos.
- *uh*: es la solución numérica.

```
> h = (b-a)/(meronodos-1);
> xh = (linspace(a,b,meronodos));
> nodosinteriores=meronodos-2;
> hD = D/ h^2 ; hV = V/ (2*h);
> e=ones(nodosinteriores,1);
> A = spdiags([-hD*e-hV (2*hD+Q)*e -hD*e+hV],...
> -1:1,nodosinteriores,nodosinteriores);

> xi = xh(2:end-1); > ff =feval(f,xi); ff(1) = ff(1)+ua*(hD+hV);
> ff(end) = ff(end)+ub*(hD-hV);
> uh = A \ ff; uh=[ua; uh; ub];
> return
```

1.4.2. Método para resolver un problema de contorno 1D con condiciones mixtas

```
>function[xh,uh]=bvp2cvrobinup(a,b,N,D,V,q,bvpfun,c11,...
>c12,c21,c22,ua,ub,esquema,varargin)
Este código resuelve problemas de contorno unidimensionales de la forma
```

$$-D(x) * (d^2U/dX^2) + V(x) * dU/dX + q(x) * U = BVPFUN$$

en el intervalo (A,B) con las condiciones de contorno

$$c11 * U'(A) + c12 * U(A) = UA$$

$$c21 * U'(B) + c22 * U(B) = UB$$

mediante diferencias finitas en N nodos internos equiespaciado en (A,B) . Notación:

- *BVPFUN*, D , V , q : funciones inline o anónimas.

- *esquema = 'C', se usan fórmulas centradas.*
- *esquema = 'U', se usan formulas descentradas a contracorriente (üpwind") para el término convectivo.*
- *varargin: permite pasar parámetros adicionales a la función BVPFUN.*
- *xh: contiene los nodos de la discretización.*
- *uh: contiene la solución numérica.*
- *N: es el número de nodos interiores.*
- *c11,c12,c21,c22: coeficientes en las condiciones de contorno.*
- *ua, ub: valores de las condiciones de contorno.*

```

> h = (b-a)/(N+1);
> xh = (linspace(a,b,N+2))'; xi = xh(2:N+1);
> if (c11 == 0 && c12 == 0)
> disp('Condición en la frontera izquierda incorrecta')
> return
> end
> if (c21 == 0 && c22 == 0)
> disp('Condición de contorno en la frontera derecha incorrecta')
> return
> end
> if c11 = 0
> TrIzq=[-D(a)/(h^2) 2*D(a)/(h^2)-V(a)*c12/c11+q(a) -D(a)/(h^2)];
> FlIzq = [-c11/(2*h) c12 c11/(2*h)];
> end
> if c21 = 0
> TrDcha=-D(b)/(h^2) 2*D(b)/(h^2)-V(b)*c22/c21+q(b) -D(b)/(h^2)];

```

```

> FlDcha = [-c21/(2*h) c22 c21/(2*h)];
> end
> if esquema == 'C'
> nW = -D(xi)/(h^2)-V(xi)/(2*h);
> nC = 2*D(xi)/(h^2)+q(xi);
> nE = -D(xi)/(h^2)+V(xi)/(2*h);
> K = [nW nC nE];
> elseif esquema == 'U'
> for i=2:N+1
> if V(a+(i-1)*h) = 0
> rho(i-1) = 1/2+abs(V(a+(i-1)*h))./(2*V(a+(i-1)*h));
> else
> rho(i-1) = 1/2;
> end
> end
> nW = -D(xi)/(h^2)-V(xi).*rho'/h;
> nC = 2*D(xi)/(h^2)+V(xi).*(2*rho'-1)/h+q(xi);
> nE = -D(xi)/(h^2)+V(xi).*(1-rho')/h;
> K = [nW nC nE];
> else
> disp('Error en el tipo de esquema. Las opciones son:')
> disp('C = centrado')
> disp('U = upwind')

```



```

> return

> end

> if c11 == 0

> if c21 == 0

> A = full([spdiags(FlIzq,[0 1 2],1,N+4);...
> spdiags(TrIzq,[0 1 2],1,N+4);...
> spdiags(K,[1 2 3],N,N+4);...
> spdiags(TrDcha,[N+1 N+2 N+3],1,N+4);...
> spdiags(FLDcha,[N+1 N+2 N+3],1,N+4)]);

> B = [ua; bvpfun(a)-V(a)*ua/c11;...
> feval(bvpfun,xi,varargin:);...
> bvpfun(b)-V(b)*ub/c21; ub];

> uh = A \ B;

> uh = [uh(2:N+3)];

> else

> cD = zeros(1,N+3); cD(1,end) = 1;

> A = full([spdiags(FlIzq,[0 1 2],1,N+3);...
> spdiags(TrIzq,[0 1 2],1,N+3);...
> spdiags(K,[1 2 3],N,N+3); cD]);

> B = [ua; bvpfun(a)-V(a)*ua/c11;feval(bvpfun,xi,varargin:); ub/c22];

> uh = A \ B;

> uh = [uh(2:N+2); ub];

> end

```

```

> else

> if c21 == 0

> cI = zeros(1,N+3); cI(1,1) = 1;

> A = full([cI; spdiags(K,[0 1 2],N,N+3);...

> spdiags(TrDcha,[N N+1 N+2],1,N+3);...

> spdiags(FlDcha,[N N+1 N+2],1,N+3)]);

> B = [ua/c12; feval(bvpfun,xi,varargin:);...

> bvpfun(b)-V(b)*ub/c21; ub];

> uh = A \ B;

> uh = [ua; uh(2:N+2)];

> else

> cI = zeros(1,N+2); cI(1,1) = 1;

> cD = zeros(1,N+2); cD(1,end) = 1;

> A = full([cI; spdiags(K,[0 1 2],N,N+2); cD]);

> B = [ua/c12; feval(bvpfun,xi,varargin:); ub/c22];

> uh = A \ B;

> uh = [ua; uh(2:N+1); ub];

> end

> end

> return

```

1.4.3. Método para resolver la ecuación del calor 1D

```

> function[x,uf]=ecucalor(C,intespacio,intiempo,...

```

```
> pasosespacio,pasost tiempo,theta,u0,cc,f)
```

Este código resuelve la ecuación del calor: en una dimensión espacial utilizando un theta-método para la discretización temporal:

$$\frac{du}{dt} = C \frac{d^2u}{dx^2} + f, \quad (t, x) \in (t_0, T) \times (a, b),$$

$$u(x, t) = cc(x, t), \quad x = a, \quad x = b,$$

$$u(x, t_0) = u_0(x), \quad x \in [a, b],$$

en una dimensión espacial utilizando un theta-método para la discretización temporal.

Los argumentos son:

- *intespacio: (a,b).*
- *int tiempo: (t0,T).*
- *u0: condición inicial.*
- *cc: condición de contorno tipo Dirichlet.*
- *theta: el valor de theta empleado en el theta método, theta=0 es para Euler explícito, theta=1 es para Euler implícito, theta=0.5 para Crank-Nicolson.*
- *pasosespacio: el número de pasos en espacio.*
- *pasost tiempo: el número de pasos en tiempo.*
- *x: el vector que contiene los nodos en espacio.*
- *uf: contiene los valores numéricos obtenidos en t=T.*

```
> h = (intespacio(2)-intespacio(1))/pasosespacio;
```

```
> dt = (int tiempo(2)-int tiempo(1))/pasost tiempo;
```

```
> N = pasosespacio+1;
```

```
> e =ones(N,1);
```

```
> D = spdiags([-e 2*e -e],[-1,0,1],N,N); I = speye(N);
```

```
> M = I+C*dt*theta*D/ h^2; Mn = I-C*dt*(1-theta)*D/h^2;
```

```
> M(1,:) = 0; M(1,1) = 1; M(N,:) = 0; M(N,N) = 1;
```

```
> x =linspace(intespacio(1),intespacio(2),N);
```

```
> x = x'; fn = feval(f,int tiempo(1),x);
```

```
> un =feval(u0,x);
```

```
> [L,U]=lu(M);
```

```
> for t = int tiempo(1)+dt:dt:int tiempo(2)
```

```

> fn1 = feval(f,t,x);
> rhs = Mn*un+dt*(theta*fn1+(1-theta)*fn);
> temp = feval(cc,t,[intespacio(1),intespacio(2)]);
> rhs([1,N]) = temp; > u = L\ rhs;
> u = U\u;
> fn = fn1; un = u;
> end
> uf=u;
> return

```

1.4.4. Método para resolver la ecuación de Poisson 2D

```

>function [u,x,y,error]=ecupoisson(a,b,c,d,dx,dy,f,...
> cc,solexacta)

```

Este código resuelve la ecuación de Poisson bidimensional:

$$-\Delta(u) = f(x, y), \text{ en } (a, b) \times (c, d)$$

con condiciones de frontera Dirichlet, dadas por la función cc(x,y), utilizando el esquema de cinco punto en cruz para aproximar el operador laplaciano Δ . También calcula el error cometido cuando se conoce la solución exacta, definida en solexacta(x,y). En caso de no conocer la solución exacta, no se mete como argumento. Los tamaños de discretización vienen dados por dx y dy.

```

> if nargin == 8
> solexacta=@(x,y) 0.*x;
> end

    Número de intervalos en x, nx, en y, ny
> nx=(b-a)/dx; ny=(d-c)/dy;
> nx1=nx+1;
> dx2=dx^2; dy2=dy^2;
> kii=2/dx2+2/dy2; kix=-1/dx2; kiy=-1/dy2;
> dim=(nx+1)*(ny+1); K=speye(dim,dim);
> rhs=zeros(dim,1);
> rhs1=zeros(dim,1);
> y = c;

```

Calculamos la matriz de coeficientes del sistema

```

> for m = 2:ny
> x = a; y = y + dy;
> for n = 2:nx
> i = n+(m-1)*(nx+1);
> x = x + dx;
> rhs(i) = feval(f,x,y);
> K(i,i) = kii; K(i,i-1) = kix;
> K(i,i+1) =kix; K(i,i+nx1)=kiy;

```

```

> K(i,i-nx1)=kiy;
> end
> end

```

Calculamos el lado derecho del sistema de ecuaciones

```

> rhs1(1:nx1) = feval(cc,x,c);
> rhs1(dim-nx:dim) = feval(cc,x,d);
> y = [c:dy:d];
> rhs1(1:nx1:dim-nx) = feval(cc,a,y);
> rhs1(nx1:nx1:dim) = feval(cc,b,y);
> rhs = rhs - K*rhs1;
> nbound = [[1:nx1],[dim-nx:dim],...
> [1:nx1:dim-nx],[nx1:nx1:dim]];
> ninternal = setdiff([1:dim],nbound);
> K = K(ninternal,ninternal);

```

Resolvemos el sistema para los nodos interiores

```

> rhs = rhs(ninternal);
> utemp = K\ rhs;
> uh = rhs1;

```

Imponemos las condiciones de contorno en los bordes

```

> uh (ninternal) = utemp;
> k = 1; y = c;
> for j = 1:ny+1
> x = a;
> for i = 1:nx1
> u(i,j) = uh(k);
> k = k + 1;
> ue(i,j) = feval(solexacta,x,y);
> x = x + dx;
> end
> y = y + dy;
> end
> x = [a:dx:b];
> y = [c:dy:d];
> if nargin == 4
> if nargin == 8
> warning('La solución exacta no se conoce');
> error = [ ];
> else
> error = max(max(abs(u-ue)))/max(max(abs(ue)));
> end
> end
> return

```

2. SEMINARIOS: EJERCICIOS RESUELTOS

Observaciones: Toda línea que comience por % indicará un comentario y lo que siga al símbolo prompt del sistema, >, es una línea de comandos.

1. Sea dada la función

$$f(x) = x^3 + 4x^2 - 10,$$

- a) Aplicar el algoritmo de bisección para calcular todas las soluciones de la ecuación $f(x) = 0$, trabajando con una tolerancia de 10^{-3} y un número máximo de 100 iteraciones.
- b) Aplicar el método de Newton para resolver la misma ecuación con los mismos valores de tolerancia y número máximo de iteraciones.

Solución. Escribimos en un script el siguiente texto:

```
% Apartado a)

> fecu=@(x) x.^ 3+4.*x.^ 2-10;
> a=0;b=2;maxitera=100;errorper=1e-3;

% Comprobamos que hemos elegido bien los extremos del intervalo:

> fecu(a); fecu(b)

% y vemos que toman signos distintos, -10 y 14.

% Hacemos la llamada al código que resuelve el problema por bisección,
% metbiseccion.m:

> [sol,itera]=metbiseccion(fecu,a,b,errorper,maxitera);

% Apartado b). A continuación resolvemos el problema por el método de
Newton-Raphson:

% sólo queda por definir:

> dfecu=@(x) 3.*x.^ 2+8.*x; x0=1;
```

```
% y llamamos al código metnewton1ecu.m
> [sol2,itera2]=metnewton1ecu(fecu,dfecu,x0,errorper,maxitera)

%Fin del script.
```

Las soluciones obtenidas son: sol=1.3643, itera=10, con el método de bisección y sol2=1.3652, itera2=4, con el método de Newton.

2. Sea dada la función

$$f(x) = e^x - 3x^2.$$

- a) Considerando el esquema numérico asociado al método de Newton como un esquema de punto fijo, definir la función $g(x)$ asociada al método y utilizar el algoritmo de punto fijo para calcular las soluciones de la ecuación $f(x) = 0$, trabajando con una tolerancia de 10^{-6} y un número máximo de 1000 iteraciones.

Solución. Escribimos en un script el siguiente texto:

```
> f=@(x) exp(x)-3.*x.^ 2;

% Comprobamos que en el intervalo [0,1] hay una solución
> a=0;b=1; f(a),f(b)

% f(a)=1, f(b)=-0.2817
% En efecto, la función toma signos distintos en 0 y en 1: 1 y -0.28172, resp.

%Definimos g(x)=x-f/f'

> g=@(x) x-(exp(x)-3.*x.^ 2).*(exp(x)-6.*x).^ (-1);

> x0=0.5;errorper=1e-6;maxitera=1000;

% Hacemos la llamada al código metpuntofijo.m

> [x,itera]=metpuntofijo(g,x0,errorper,maxitera)
```

La solución obtenida es: x=0.91001 e itera=5.

3. Resolver el sistema de ecuaciones no lineales:

$$\begin{cases} \ln(x^2 + y^2) - \sin(xy) - (\ln(2) + \ln(\pi)) = 0 \\ e^{x-y} + \cos(xy) = 0. \end{cases}$$

utilizando el método de Newton y trabajando con una tolerancia de 10^{-6} . Utilizar como semilla $(x_0, y_0) = (2, 2)$.

Solución.

Primero definimos los scripts con las funciones fecusistema.m y jacobiana.m:

```
> function F=fecusistema(x,y)

> F(1,1)=log(x.^ 2+y.^ 2)-sin(x.*y)-(log(2)+log(pi));
> F(2,1)=exp(x-y)+cos(x.*y);
> end

> function J=jacobiana(x,y)
> J(1,1)=2.*x.*(x.^ 2+y.^ 2).^ (-1)-y.*cos(x.*y);
> J(1,2)=2.*y.*(x.^ 2+y.^ 2).^ (-1)-x.*cos(x.*y);
> J(2,1)=exp(x-y)-y.*sin(x.*y);
> J(2,2)=-exp(x-y)-x.*sin(x.*y);
> end
```

A continuación, llamamos al código metnewtonsistemas.m, para ellos escribimos un nuevo script con los comandos:

```
> vectorx0=[2;2]; errorper=1e-6;maxitera=100;
> [X,itera]=metnewtonsistema(@fecusistema,...
> @jacobiana,vectorx0,errorper,maxitera)
```

La solución obtenida es: $X=(x,y)=(1.77245,1.77245)$, itera=6.

4. Se considera el problema de valor inicial:

$$\begin{cases} y' = y - \sin(t) + \cos(t), \\ y(0) = 1. \end{cases}$$

En el intervalo temporal $[0, 2]$, utilizar los esquemas numéricos de Euler explícito, Euler implícito y Crank-Nicolson para obtener una solución aproximada, tomando un paso de discretización $h = 0,1$. Sabiendo que la solución exacta es $y(t) = e^t + \sin(t)$, dibujar las soluciones obtenidas

con los tres métodos y la solución exacta en un mismo plot. Comparar los resultados obtenidos. Escribir los valores de las distintas soluciones en $t = 0,5$ y $t = 1,5$.

Solución. Abrimos un nuevo script para escribir el siguiente código y luego ejecutarlo:

```
> f=@(t,y) y-sin(t)+cos(t);

> intiempo=[0 2]; valorini=1;npasos=20;

% Hacemos una llamada a los códigos correspondientes:
> [solt,solye]=eulerexpicito(f,intiempo,valorini,npasos);

> [solt,solyi]=eulerimplicito(f,intiempo,valorini,npasos);

> [solt,solyc]=cranknicolson(f,intiempo,valorini,npasos);

% Observa que llamamos de forma distinta a los resultados para y.
> solexac=@(t) exp(t)+sin(t); figure;

% Dibujamos las soluciones junto con los valores de la solución exacta
para comparar
> plot(solt,solye,'ro',solt,solyi,'b+',...

> solt,solyc,'g*',solt,solexac(solt),'k^')

> t1=min(find(solt>=0.5));

% también t1=find(solt==0.5);

> t2=min(find(solt>=1.5));

% también t2=find(solt==1.5);

> solye(t1),solyi(t1),solyc(t1), solye(t2),solyi(t2),solyc(t2)
```

Las soluciones son: para $t=0,5$, con EE 2.096, con EI 2.1646 y con CN 2.1283. Para $t=1,5$, con EE 5.2516, con EI 5.7585 y con CN, 5.4825.

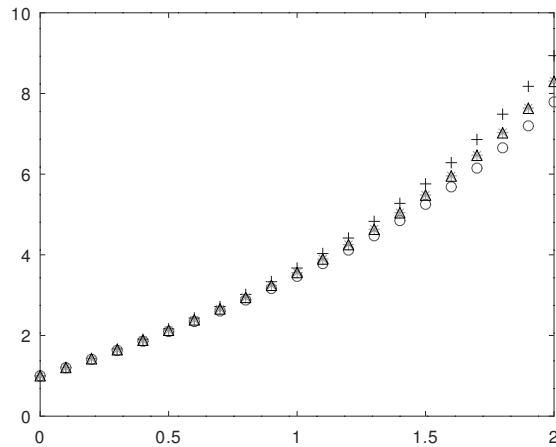


Figura 1: Resultados obtenidos con los 3 métodos junto con la exacta (en círculos rojos EE, en + azules EI, en * verdes CN, y la exacta en triángulos negros).

5. Se considera el problema de valor inicial:

$$\begin{cases} y' = -2ty^2, \\ y(0) = 1. \end{cases}$$

Aplicar los métodos tipo Runge-Kutta dados por los algoritmos de Heun (heun.m) y Simpson (rungekuttao3.m), para resolver el PVI anterior en el intervalo temporal $[0,1]$, tomando como paso de discretización $h = 0,01$. Escribir los valores obtenidos con los dos métodos para los tiempos $t = 0,1$, $t = 0,5$ y $t = 1$.

Solución. Abrimos un script con el siguiente código :

```
> f=@(t,y) -2.*t.*y.^ 2; intiempo=[0 1];

> npasos=100;valorini=1;

> [solt,soly1]=heun(f,intiempo,valorini,npasos);

> [solt,soly2]=rungekuttao3(f,intiempo,valorini,npasos);

% Dibujamos los resultados obtenidos con los dos métodos:
> plot(solt,soly1,'ro',solt,soly2,'b*')

% Calculamos los valores numéricos en los tiempos pedidos:
```

```

> t1=min(find(solt>=0.1));
> t2= min(find(solt>=0.5));
> t3=min(find(solt>=1));
> soly1(t1),soly1(t2),soly1(t3),soly2(t1),soly2(t2),soly2(t3)

```

Los resultados obtenidos con Heun son: en $t=0.1$, 0.9901, en $t=0.5$, 0.8000, en $t=1$, 0.5174. Los resultados obtenidos con Simpson son: en $t=0.1$, 0.9901, en $t=0.5$, 0.8006, en $t=1$, 0.5014.

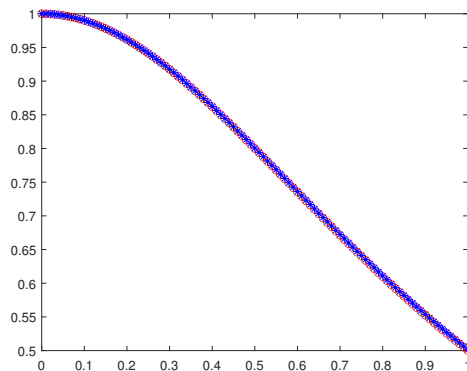


Figura 2: Resultados obtenidos con el método de Heun en rojo y con Simpson en azul.

6. Sea dado el Problema de Transporte Difusivo y Convectivo definido por el PVC:

$$\begin{cases} -u'' - 4u' = -16x^3 + 34x - 1, & \forall x \in (0, 2), \\ u(0) = 4, & u(2) = 2, \end{cases}$$

cuya solución analítica es:

$$sol(x) = x^4 - x^3 - 3,5x^2 + 2x + 4,$$

siendo u la concentración de un contaminante.

- Aplicar el algoritmo `bvdirichlet.m` para calcular la solución en el intervalo $[0, 2]$ con paso de discretización $h = 0,125$.
- Dibujar la solución analítica junto con la solución numérica.
- Determinar los valores máximos y mínimos de contaminación así como su localización utilizando la solución numérica y la analítica.

Solución Creamos un script con el siguiente código para luego ejecutarlo:

```

> a=0; b=2; D=1; V=-4; Q=0;

> f=@(x) -16.*x.^ 3+34.*x-1; ua=4; ub=2; NumeroNodos=17;

> [xh,uh]=bvpsdirichlet(a,b,NumeroNodos,D,V,Q,f,ua,ub);

> solexac=xh.^ 4-xh.^ 3-3.5.*xh.^ 2+2.*xh+4;

> figure; plot(xh,uh,'r',xh,solexac,'g')

```

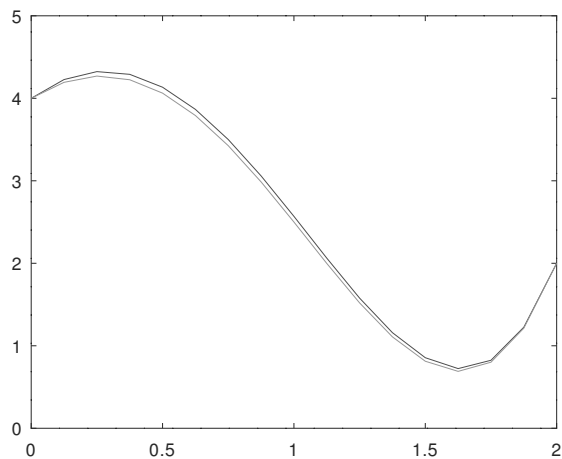


Figura 3: Resultados obtenidos con el método numérico en rojo, y la solución exacta en verde.

```

> nmax=max(uh),nmin=min(uh)

> emin=min(solexac),emax=max(solexac)

Los resultados son: nmax=4.3235, nmin=0.7233, emin=0.6897, emax=4.2695.
% Dibujamos el error cometido:
> figure; err=uh-solexac; plot(xh,err)

% Vemos dónde se alcanzan el valor máximo y el mínimo
> xm=find(uh>=nmax), xmin=find(uh<=nmin), xh(xm),xh(xmin)

```

Los valores obtenidos (componentes) son: $x_m=3$ y $x_{min}=14$. Por tanto, el valor máximo se alcanza en $x=0.25$, y el valor mínimo en $x=1.6250$.

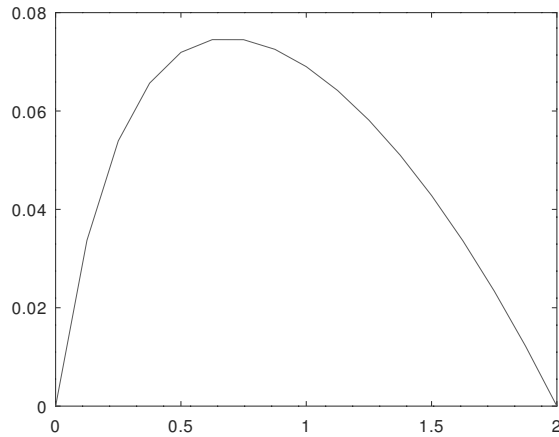


Figura 4: Error cometido.

7. Sea dado el PVIC:

$$\begin{cases} \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + f(x, t), \\ u(x, t) = cc(x, t), \quad x = 0, \quad x = 1, \\ u(x, 0) = u_0(x), \quad x \in (0, 1) \end{cases}$$

donde

$$\begin{aligned} f(x, t) &= 2t - x^2 + 10xt, \\ cc(x, t) &= x^2(x - 1), \\ u_0(x) &= 0. \end{aligned}$$

Aplicar el algoritmo definido en eucalor.m para calcular la solución en el intervalo temporal $[0,1]$, con paso de discretización espacial, $\Delta x = 0,05$ y de discretización temporal, $\Delta t = 0,02$. Dibujar la solución obtenida para $\theta = 0$ (Euler Explícito). ¿Puedes justificar la gráfica obtenida? Determinar el paso de discretización temporal máximo que asegura estabilidad y calcular la solución para ese paso temporal. ¿Cuánto vale la solución en el punto $x = 0.85$?

Solución. Abrimos un nuevo script con el siguiente código:

```
> intespacio=[0 1]; int tiempo=[0 1];
> pasosespacio=20,pasost tiempo=100;
> theta=0; c=1; u0=@(x) 0.*x;
```

```

> cc=@(t,x) (x.^2).*(x-1);

> f=@(t,x) 2.*t-x.^2+10.*x.*t;

> [x,u]=ecucalor(c,intespacio,intiempo,pasosespacio,...

> pasostiempo,theta,u0,cc,f);

> figure; plot(x,u)

% Se puede observar que el plot no aparece puesto que hay valores
% que son del orden de 1e+11
> x1=min(find(x>=0.85)); u(x1)

% Se tiene que x1=18 y u(18)=3.6504e+11
% Esto se debe a la inestabilidad asociada al método explícito
% Para que sea estable, debemos reducir el paso temporal
> dx=0.05; dt=(0.05)^2; pasost=1/(0.05)^2 ;

> pasosespacio=20; pasosespacio=800;
> [x,u1]=ecucalor(c,intespacio,intiempo,pasosespacio,pasostiempo,...

> theta,u0,cc,f);

> figure;

> plot(x,u1), x1=min(find(x>=0.85)), u1(x1)

```

El resultado obtenido es $u_1(18)=0.4496$.

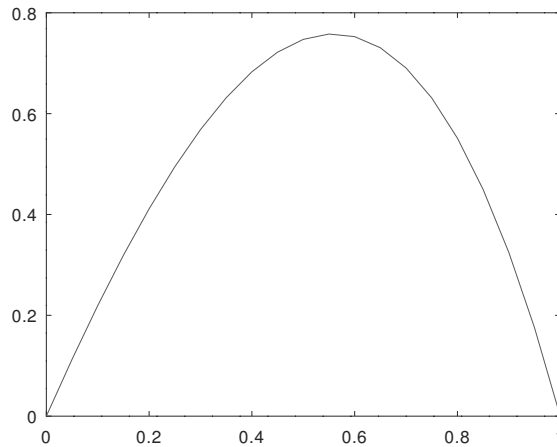


Figura 5: Resultados obtenidos cumpliendo el criterio de estabilidad.

3. EJERCICIOS DE PRÁCTICAS

1. Sea dada la función

$$f(x) = e^x - 15 - \operatorname{arctg}(x).$$

- a) Aplicar el algoritmo de bisección para calcular la solución de la ecuación $f(x) = 0$, trabajando con una tolerancia de 10^{-3} y un número máximo de 100 iteraciones.
- b) Aplicar el método de Newton para resolver la misma ecuación con los mismos valores de tolerancia y número máximo de iteraciones.

2. Sea dada la función

$$f(x) = \sqrt{x}\sin(x) - x^3 + 2.$$

- a) Considerando el esquema numérico asociado al método de Newton como un esquema de punto fijo, definir la función $g(x)$ asociada al método y utilizar el algoritmo de punto fijo para calcular la solución de la ecuación $f(x) = 0$, trabajando con una tolerancia de 10^{-6} y un número máximo de 1000 iteraciones.
- b) Define un esquema de punto fijo diferente al considerado en el apartado anterior justificando la elección, para calcular la solución de la ecuación $f(x) = 0$ trabajando con una tolerancia de 10^{-6} y un número máximo de 1000 iteraciones.

3. Resolver el sistema de ecuaciones no lineales:

$$\begin{cases} -x - y^3 + x^2 - 2 = 0, \\ 1 - 2x + y - y^2 = 0. \end{cases}$$

utilizando el método de Newton y trabajando con una tolerancia de 10^{-6} . Utilizar como semilla $(x_0, y_0) = (-0,8, -0,4)$.

4. Se considera el problema de valor inicial:

$$\begin{cases} y' = -kt^\beta y, \\ y(0) = 1, \end{cases}$$

donde $k = 4$ y $\beta = 0,25$. Resolver el PVI en el intervalo $[0, 4]$, utilizando los métodos de Euler explícito y Euler implícito. Analizar la estabilidad de los dos métodos utilizando para ello distintos pasos de discretización, por ejemplo $h_1 = 0,5$, $h_2 = 0,4$, $h_3 = 0,2$ y $h_4 = 0,1$.

5. Se considera el problema de valor inicial:

$$\begin{cases} y' = \frac{2y}{t+1}, \\ y(0) = 1. \end{cases}$$

En el intervalo temporal $[0, 1]$, utilizar los esquemas numéricos de Euler explícito, Euler implícito y Crank-Nicholson para obtener el valor aproximado de la solución en $t = 1$, tomando un paso de discretización $h = 0,1$. Sabiendo que la solución exacta es $y(t) = (t + 1)^2$, dibujar las soluciones obtenidas con los tres métodos y la solución exacta en un mismo plot. Comparar los resultados obtenidos. Escribir los valores de las distintas soluciones en $t = 0,5$ y $t = 0,75$. Encontrar los intervalos temporales para los cuales las distintas soluciones numéricas alcanzan valores superiores a tres.

6. Se considera el problema de valor inicial:

$$\begin{cases} y' = -y(y - 1), \\ y(0) = 2. \end{cases}$$

Aplicar los métodos tipo Runge-Kutta dados por los algoritmos `heun.m` y `rungekuttao3.m`, para resolver el PVI anterior en el intervalo temporal $[0,1]$, tomando como paso de discretización $h = 0,01$. Escribir los valores obtenidos con los dos métodos para los tiempos $t = 0,1$, $t = 0,5$ y $t = 1$.

7. Sea dado el Problema de Transporte Difusivo y Convectivo definido por el PVC:

$$\begin{cases} -u'' + u' = -2e^{-x}, & \forall x \in (0, 1), \\ u(0) = 2, & u(1) = 2\cosh(1), \end{cases}$$

cuya solución analítica es:

$$sol(x) = e^x + e^{-x}$$

- a) Aplicar el algoritmo `bvmdirichlet.m` para calcular la solución en el intervalo $[0, 1]$ con paso de discretización $h = 0,01$.
- b) Dibujar la solución analítica junto con la solución numérica.
- c) Determinar los valores máximos y mínimos de u y su localización, para la solución numérica y la solución analítica. Calcular los errores cometidos.

8. Sea dado el PVIC:

$$\begin{cases} \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + f(x, t), \\ u(x, t) = cc(x, t), \quad x = 0, \quad x = 2, \\ u(x, 0) = u_0(x), \quad x \in (0, 2), \end{cases}$$

donde

$$f(x, t) = t \sin(x),$$

$$cc(x, t) = 0,$$

$$u_0(x) = x, \text{ para } 0 \leq x \leq 1 \quad \text{y} \quad u_0(x) = 2 - x, \text{ para } 1 < x \leq 2.$$

Aplicar el algoritmo de `ecucalor.m` para calcular la solución en el intervalo temporal $[0,1]$, con paso de discretización espacial $\Delta x = 0,05$ y discretización temporal, $\Delta t = 0,02$. Dibujar la solución obtenida para $\theta = 0,5$ en $t = 1$. ¿Cuánto vale la solución en el punto $x = 1$?