MÉTODOS DE COMPUTACIÓN ESTADÍSTICA

Chapter 1

Introduction to MATLAB

1.38. The classic quadratic formula says that the two roots of the quadratic equa−

tion

$ax2 + bx + c = 0$

are

$x1; x2 = ¡b \S pb2 ¡ 4ac$

$2a$

Use this formula in Matlab to compute both roots for

$a = 1; b = ¡100000000; c = 1$

Compare your computed results with

roots([a b c])

What happens if you try to compute the roots by hand or with a hand

calculator?

You should ¯nd that the classic formula is good for computing one root, but

not the other. So use it to compute one root accurately and then use the fact

that

$x1x2 = c$

a

to compute the other.

X1=(−10^−8+sqrt(10^16−4))/2

X1 =

5.0000e+007

X2=(−10^−8−sqrt(10^16−4))/2

X2 =

−50000000

>> help roots

roots([1 10^8 1])

ans =

1.0e+008 *

−1.0000

−0.0000

Ejercicio sin número sobre error absoluto y error relativo:

Calcular el error absoluto y relativo producido al calcular n factorial con la formula de Stirling n=1,..,17.

f=factorial(10)

f =

3628800

>> fs=sqrt(2*pi*10)*exp(−10)*(10^10)

fs =

3.598695618741036e+006

>> er=abs(f−fs)/f

er =

0.00829596044394

ea=abs(f−fs)

ea =

3.010438125896407e+004

f=factorial(20)

f =

2.432902008176640e+018

>> fs=sqrt(2*pi*20)*exp(−20)*(20^20)

```
fs =

2.422786846761134e+018

>> ea=abs(f−fs)

ea =

1.011516141550643e+016

>> er=abs(f−fs)/f

er =

0.00415765262288
```

Ejercicio sin numero:

```
% error relativo y absoluto

%en la formula de Stirling

%imprime una tabla mostrando

%el error producido por la

%aproximacion de Stirling

%para n!

close all;clc

disp('')

disp(' Aproximation absolute relative')

disp(' n n! Stirling error error')

disp('−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

e=exp(1);

nfact=1;

for n=1:17

%f=factorial(n);

f=n*f;

fs=sqrt(2*pi*n)*((n/e)^n);
```

abserror=abs(f−fs);

relerror=abserror/f;

d=sprintf(' %2.0f %16.0f %18.2f %16.2f %5.2e',...

n,f,fs,abserror,relerror);

IMPORTANTE:%18.2 es 18 digitos con 2decimales,por lo tanto quedan 16enteros;la f sigifica el formato que le corresponda y la e significa en formato decimal;como d=sprintf(' %2.0f %16.0f %18.2f %16.2f %5.2e',...

n,f,fs,abserror,relerror) esta entre comillas eso significa que es un texto

disp(d)

end

Lo compruebo o ejecuto:

Aproximation absolute relative

n n! Stirling error error

_____

1 1 0.92 0.08 7.79e−002

2 2 1.92 0.08 4.05e−002

3 6 5.84 0.16 2.73e−002

4 24 23.51 0.49 2.06e−002

5 120 118.02 1.98 1.65e−002

6 720 710.08 9.92 1.38e−002

7 5040 4980.40 59.60 1.18e−002

8 40320 39902.40 417.60 1.04e−002

9 362880 359536.87 3343.13 9.21e−003

10 3628800 3598695.62 30104.38 8.30e−003

11 39916800 39615625.05 301174.95 7.55e−003

12 479001600 475687486.47 3314113.53 6.92e−003

13 6227020800 6187239475.19 39781324.81 6.39e−003

14 87178291200 86661001740.60 517289459.40 5.93e−003

15 1307674368000 1300430722199.46 7243645800.54 5.54e−003

16 20922789888000 20814114415223.09 108675472776.91 5.19e−003

17 355687428096000 353948328666099.44 1739099429900.56 4.89e003

>>

14−marzo−06

MATLAB respuestas a la pagina 49 de unidades didácticas 01_intro.pdf

```
%este programa calcula el
%seno de x usando el
%desarrollo en serie de taylor
%en torno al cero
%sin(x) = x − x^3/3! + x^5/5! − ...
%el criterio de parada del
%bucle "while" consiste en
%que el siguiente termino
%de la serie sea tan pequeño
%que no modifique el resultado
%obtenido hasta el momento
function s = powersin(x)
s = 0;
t = x,
n = 1;
while s + t ~= s;
s = s+t;
t = − x.^2/((n+1)*(n+2)).*t *para hallar ~ es: alt+126
n = n+2;
end
```

- *control+c desbloquea las operaciones*

lo abrimos y lo usamos:

>> powersin(pi)

t =

3.1416

ans =

2.4791e−016

>> x = 11*(pi)/2;

>> e=abs(powersin(x)−1)

t =

17.2788

e =

2.0000

>> sin(11*pi/2)

ans =

−1

>> sin(21*pi/2)

ans =

1

>> e = abs(powersin(x)+1)

t =

17.2788

e =

2.1287e−010

**donde e es igual al erros, abs igual al valor absoluto.*

>> x = 21*(pi)/2;

```
>> e = abs(powersin(x)+1)

t =

32.9867

e =

1.9999

>> e = abs(powersin(x)−1)

t =

32.9867

e =

1.3324e−004

>> x = 31*(pi)/2;

>> e = abs(powersin(x)−1)

t =

48.6947

e =

5.8230e+003
```

- modificamos el programa inicial:

```
function [s,k] = powersin(x)

s = 0;

t = x,

n = 1;

k = 0;

while s + t ~= s;

k = k +1;

s = s+t;

t = − x.^2/((n+1)*(n+2)).*t;
```

```
n = n+2;

end
```

Operamos:

```
>> [s,k]= powersin(x)

t =

48.6947

s =

−5.8220e+003

k =

78

>> x = (pi)/2;

>> [s,k]= powersin(x)

t =

1.5708

s =

1.0000

k =

11
```

*cambiamos de nuevo el programa :

```
function [s,k,tmax] = powersin(x)

s = 0;

t = x,

n = 1;

k = 0;

tmax=x;

while s + t ~= s;
```

```
k = k +1;

s = s+t;

t = − x.^2/((n+1)*(n+2)).*t;

if t > tmax

tmax = t;

end

n = n+2;

end
```

operamos:

```
>> x = (pi)/2;

>> [s,k,tmax]= powersin(x)

t =

1.5708

s =

1.0000

k =

11

tmax =

1.5708

>>

>> x = 11*(pi)/2;

>> [s,k,tmax]= powersin(x)

t =

17.2788

s =

−1.0000
```

k =

37

tmax =

3.0665e+006

Modificando de nuevo:

```
function [s,k,tmax,t] = powersin(x)

s = 0;

t = x,

n = 1;

k = 0;

tmax=x;

while s + t ~= s;

k = k +1;

s = s+t;

t = − x.^2/((n+1)*(n+2)).*t;

if t > tmax

tmax = t;

end

n = n+2;

end
```

Operamos

```
>> [s,k,tmax,t]= powersin(x)
```

t =

17.2788

s =

−1.0000

k =

37

tmax =

3.0665e+006

t =

−2.6232e−017

>>

28−marzo−06

Tenemos el programa lutx:

```
function [L,U,p] = lutx(A)
%LUTX Triangular factorization, textbook version
% [L,U,p] = lutx(A) produces a unit lower triangular matrix L,
% an upper triangular matrix U, and a permutation vector p,
% so that L*U = A(p,:)
[n,n] = size(A);
p = (1:n)';
for k = 1:n−1
% Find index of largest element below diagonal in k−th column
[r,m] = max(abs(A(k:n,k)));
m = m+k−1;
% Skip elimination if column is zero
if (A(m,k) ~= 0)
% Swap pivot row
if (m ~= k)
A([k m],:) = A([m k],:);
p([k m]) = p([m k]);
```

```
end
% Compute multipliers
i = k+1:n;
A(i,k) = A(i,k)/A(k,k);
% Update the remainder of the matrix
j = k+1:n;
A(i,j) = A(i,j) − A(i,k)*A(k,j);
end
end
```

y aplicamos en matlab:

```
>>A= [0 1;1 0]
A =
0 1
1 0
>> [L,U,p]= lutx(A)
```

Y:

```
function [L,U,p,sig] = lutx(A)
%LUTX Triangular factorization, textbook version
% [L,U,p] = lutx(A) produces a unit lower triangular matrix L,
% an upper triangular matrix U, and a permutation vector p,
% so that L*U = A(p,:)
[n,n] = size(A);
p = (1:n)';
sig=1;
for k = 1:n−1
% Find index of largest element below diagonal in k−th column
```

```
[r,m] = max(abs(A(k:n,k)));
m = m+k−1;
% Skip elimination if column is zero
if (A(m,k) ~= 0)
% Swap pivot row
if (m ~= k)
A([k m],:) = A([m k],:);
p([k m]) = p([m k]);
sig = −1*sig;
end
% Compute multipliers
i = k+1:n;
A(i,k) = A(i,k)/A(k,k);
% Update the remainder of the matrix
j = k+1:n;
A(i,j) = A(i,j) − A(i,k)*A(k,j);
end
end
% Separate result
L = tril(A,−1) + eye(n,n);
U = triu(A);
```

Aplicamos::

```
>> A
A =
0 1
1 0
```

```
>> [L,U,p] = lutx(A)

L =

1 0

0 1

U =

1 0

0 1

p =

2

1

>> [L,U,p,sig] = lusigno(A)

> [L,U,p,sig] = lusigno(A)

L =

1 0

0 1

U =

1 0

0 1

p =

2

1

sig =

−1

>> A = [1 2 3;2 1 4;−1 2 1]

A =

1 2 3
```

2 1 4

−1 2 1

>> [L,U,p,sig] = lusigno(A)

L =

1.0000 0 0

−0.5000 1.0000 0

0.5000 0.6000 1.0000

U =

2.0000 1.0000 4.0000

0 2.5000 3.0000

0 0 −0.8000

p =

2

3

1

sig =

1

>>

Programa:

```
%Ejercicio 2.7 del libro Moler
%este programa calcula el determinante
%de la matriz A utilizando la factorización
%LU como de A ,PA=LU
%det(A) = +− U(1,1)*U(2,2)*...*U(n,n)
%donde U(i,i) son los elementos de la diagonal
%de la matriz U, y el signo +− es
```

%+ si el numero de intercambios de filas

%que hemos realizado es par y menos en caso

%contrario. El signo se almacena en la

%variable sig

function d = mideterminante(A)

[L,U,p,sig] = lusigno(A)

Aplico:

>> A

A =

1 2 3

2 1 4

−1 2 1

>> A = [0 1;1 0]

A =

0 1

1 0

>> diag(A)

ans =

0

0

>> X

>> X=diag(A)

X =

0

0

>> B=diag(X)

B =

0 0

0 0

>> X= 1:5

X =

1 2 3 4 5

B =

1 0 0 0 0

0 2 0 0 0

0 0 3 0 0

0 0 0 4 0

0 0 0 0 5

>> mideterminante(A)

ans =

−1

> B=eye(50)

B =

Columns 1 through 12

1 0 0 0 0 0 0 0 0 0 0 0

0 1 0 0 0 0 0 0 0 0 0 0

0 0 1 0 0 0 0 0 0 0 0 0

0 0 0 1 0 0 0 0 0 0 0 0

0 0 0 0 1 0 0 0 0 0 0 0

0 0 0 0 0 1 0 0 0 0 0 0

0 0 0 0 0 0 1 0 0 0 0 0

0 0 0 0 0 0 0 1 0 0 0 0

0 0 0 0 0 0 0 0 1 0 0 0

0 0 0 0 0 0 0 0 0 1 0 0

0 0 0 0 0 0 0 0 0 0 1 0

0 0 0 0 0 0 0 0 0 0 0 1

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

Columns 13 through 24

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

1 0 0 0 0 0 0 0 0 0 0 0

0 1 0 0 0 0 0 0 0 0 0 0

0 0 1 0 0 0 0 0 0 0 0 0

0 0 0 1 0 0 0 0 0 0 0 0

0 0 0 0 1 0 0 0 0 0 0 0

0 0 0 0 0 1 0 0 0 0 0 0

0 0 0 0 0 0 1 0 0 0 0 0

0 0 0 0 0 0 0 1 0 0 0 0

0 0 0 0 0 0 0 0 1 0 0 0

0 0 0 0 0 0 0 0 0 1 0 0

0 0 0 0 0 0 0 0 0 0 1 0

0 0 0 0 0 0 0 0 0 0 0 1

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

Columns 25 through 36

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

1 0 0 0 0 0 0 0 0 0 0 0

0 1 0 0 0 0 0 0 0 0 0 0

0 0 1 0 0 0 0 0 0 0 0 0

0 0 0 1 0 0 0 0 0 0 0 0

0 0 0 0 1 0 0 0 0 0 0 0

0 0 0 0 0 1 0 0 0 0 0 0

0 0 0 0 0 0 1 0 0 0 0 0

0 0 0 0 0 0 0 1 0 0 0 0

0 0 0 0 0 0 0 0 1 0 0 0

0 0 0 0 0 0 0 0 0 1 0 0

0 0 0 0 0 0 0 0 0 0 1 0

0 0 0 0 0 0 0 0 0 0 0 1

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

Columns 37 through 48

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

```
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0
```

0 1 0 0 0 0 0 0 0 0 0 0

0 0 1 0 0 0 0 0 0 0 0 0

0 0 0 1 0 0 0 0 0 0 0 0

0 0 0 0 1 0 0 0 0 0 0 0

0 0 0 0 0 1 0 0 0 0 0 0

0 0 0 0 0 0 1 0 0 0 0 0

0 0 0 0 0 0 0 1 0 0 0 0

0 0 0 0 0 0 0 0 1 0 0 0

0 0 0 0 0 0 0 0 0 1 0 0

0 0 0 0 0 0 0 0 0 0 1 0

0 0 0 0 0 0 0 0 0 0 0 1

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

Columns 49 through 50

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

0 0

1 0

0 1

Pasamos al ejemplo 2.11 del libro de moler capitulo 002 pagina 37.

2.7 lutx, bslashtx, lugui

We have three functions implementing the algorithms discussed in this chapter.

The ¯rst function, lutx, is a readable version of the built−in Matlab function lu.

There is one outer for loop on k that counts the elimination steps. The inner loops

on i and j are implemented with vector and matrix operations, so that the overall

function is reasonably e±cient.

```
function [L,U,p] = lutx(A)
%LU Triangular factorization
% [L,U,p] = lutx(A) produces a unit lower triangular
% matrix L, an upper triangular matrix U, and a
% permutation vector p, so that L*U = A(p,:).
[n,n] = size(A);
p = (1:n)'
```

```
for k = 1:n−1
% Find largest element below diagonal in k−th column
[r,m] = max(abs(A(k:n,k)));
m = m+k−1;
% Skip elimination if column is zero
if (A(m,k) ~= 0)
% Swap pivot row
if (m ~= k)
A([k m],:) = A([m k],:);
p([k m]) = p([m k]);
end
% Compute multipliers
i = k+1:n;
A(i,k) = A(i,k)/A(k,k);
% Update the remainder of the matrix
j = k+1:n;
A(i,j) = A(i,j) − A(i,k)*A(k,j);
end
end
% Separate result
L = tril(A,−1) + eye(n,n);
U = triu(A);
```

Y la versión simplificada de este operador la llama: Bslashtx(A,b) [backslash]:

```
function x = bslashtx(A,b)
% BSLASHTX Solve linear system (backslash)
% x = bslashtx(A,b) solves A*x = b
```

```
[n,n] = size(A);
if isequal(triu(A,1),zeros(n,n))
% Lower triangular
x = forward(A,b);
return
elseif isequal(tril(A,−1),zeros(n,n))
% Upper triangular
x = backsubs(A,b);
return
elseif isequal(A,A')
[R,fail] = chol(A);
if ~fail
% Positive definite
y = forward(R',b);
x = backsubs(R,y);
return
end
end
```

Realizamos otro programa para la inversa (ejercicio 2.11 de moler)

```
%Ejercicio 2.11 del libro de Moler
%este programa nos permite calcular
%la matriz inversa de una matriz
%usando la factorización LU y
%las versiones sencillas de los
%programas LU y \ que nos ofrece
%el libro de Moler
```

```
function X = miinversa(A)
[n,n] = size (A);
E = eye(n);
for i=1:n
%x(:,i)=bslashtx(A,E(:,i));
X(:,i)= A\E(:,i);
end
```

aplicamos en matlab:

```
A =
0 1
1 0
>> miinversa(A)
ans =
0 1
1 0
>> B=rand(50);
>> X=miinversa(B);
>> norm(B*X − eye(50))
ans =
2.3934e−014
>> X2=inv(B);
>>
>> norm(B*X2 − eye(50))
ans =
1.2760e−013
>> cond(B)
```

ans =

517.6219

4−abril−06

CAPITULO 3 PDF

A =

1 1 1

1 2 4

1 3 9

>> B= [2;1;3]

B =

2

1

3

>> P = A\B

P =

6.0000

−5.5000

1.5000

>> X= [1;2;3]

X =

1

2

3

>> Y = [2; 1; 3]

Y =

2

1

3

>>plot(X,Y,'r*')

Y obtenemos un gráfico:

>> plot(X,Y,'r*',u,v)

u=0:0.01:4;

>> v= 6 −5.5*u+315*u.^2;

(es el polinomio obtenido antes evaluado en los puntos dados)

>> plot(X,Y,'r*',u,v)

Y obtenemos:

*Tenemos los siguientes programas:

```
function v = polyinterp(x,y,u)
%POLYINTERP Polynomial interpolation.
% v = POLYINTERP(x,y,u) computes v(j) = P(u(j)) where P is the
% polynomial of degree d = length(x)−1 with P(x(i)) = y(i).
% Use Lagrangian representation.
% Evaluate at all elements of u simultaneously.
n = length(x);
v = zeros(size(u));
for k = 1:n
w = ones(size(u));
for j = [1:k−1 k+1:n]
w = (u−x(j))./(x(k)−x(j)).*w;
end
v = v + w*y(k);
end
```

número de puntos

Y:

```
function v = piecelin(x,y,u)
%PIECELIN Piecewise linear interpolation.
% v = piecelin(x,y,u) finds the piecewise linear L(x)
% with L(x(j)) = y(j) and returns v(k) = L(u(k)).
% First divided difference
delta = diff(y)./diff(x);
% Find subinterval indices k so that x(k) <= u < x(k+1)
n = length(x);
k = ones(size(u));
for j = 2:n−1
k(x(j) <= u) = j;
end
% Evaluate interpolant
s = u − x(k);
v = y(k) + s.*delta(k);
>> X= 0:10
X =
0 1 2 3 4 5 6 7 8 9 10
>> Y=rand(1,11);
>> u= −0.1:0.01:10.1;
>> Y = 10*Y;
>> v= polyinterp(X,Y,u);
>> plot(X,Y,'r*',u,v)
```

Y obtenemos:

>> x=[1 4 5 4 −1 2 0]

x =

1 4 5 4 −1 2 0

>> diff(x)

ans =

3 1 −1 −5 3 −2

>>

>> x

x =

0 1 2 3 4 5 6 7 8

>> y = 10*rand(1,9);

>> y

y =

Columns 1 through 7

4.3866 4.9831 2.1396 6.4349 3.2004 9.6010 7.2663

Columns 8 through 9

4.1195 7.4457

>> u = −1:0.01:9;

>> v = piecelin(x,y,u);

>> plot(x,y,'r*',u,v)

06/04/06

Del capitulo 3pdf la segunda parte.

$$P(x) = \frac{3hs^2 - 2s^3}{h^3} y_{k+1} + \frac{h^3 - 3hs^2 + 2s^3}{h^3} y_k + \frac{s^2(s-h)}{h^2} d_{k+1} + \frac{s(s-h)^2}{h^2} d_k$$

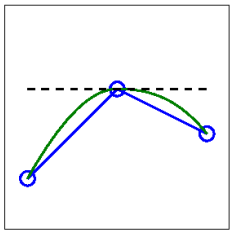
pchip

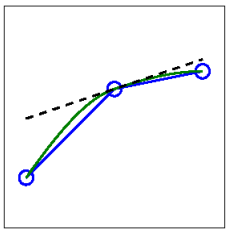
If δ_k and δ_{k-1} have opposite signs

$$d_k = 0$$

else

$$\frac{1}{d_k} = \frac{1}{2}\left(\frac{1}{\delta_{k-1}} + \frac{1}{\delta_k}\right)$$





Local power form

$$P(x) = y_k + sd_k + s^2c_k + s^3b_k$$

$$\begin{aligned} c_k &= \frac{3\delta_k - 2d_k - d_{k+1}}{h} \\ b_k &= \frac{d_k - 2\delta_k + d_{k+1}}{h^2} \end{aligned}$$

- Este es un buen método (el pichip) para interpolar, pero los hay mejores(el splines)

*Los ptos di son knots

K=1 d0+4d1+d3 = 3 +3 eliminada

K=2 d1+4d2+d4=3 +3

K=3 d2+4d3+d5=3 +3

K=4 d1+4d4+d6=3 +3

K=5 d4+4d5+ d7=3 +3 eliminada

.−Si esto no funciona usamos spline, de la cual una versión simplificada es la splinetx.m , que es:

```
function v = splinetx(x,y,u)

%SPLINETX Textbook spline function.

% v = splinetx(x,y,u) finds the piecewise cubic interpolatory

% spline S(x), with S(x(j)) = y(j), and returns v(k) = S(u(k)).

%

% See SPLINE, PCHIPTX.

% First derivatives

h = diff(x);

delta = diff(y)./h;

d = splineslopes(h,delta);

% Piecewise polynomial coefficients

n = length(x);

c = (3*delta − 2*d(1:n−1) − d(2:n))./h;

b = (d(1:n−1) − 2*delta + d(2:n))./h.^2;

% Find subinterval indices k so that x(k) <= u < x(k+1)

k = ones(size(u));

for j = 2:n−1

k(x(j) <= u) = j;

end

% Evaluate spline

s = u − x(k);

v = y(k) + s.*(d(k) + s.*(c(k) + s.*b(k)));
```

Ensaya las primeras derivadas

* Implementa alguno de los métodos para hallar las di. (not−a kan ; loked)

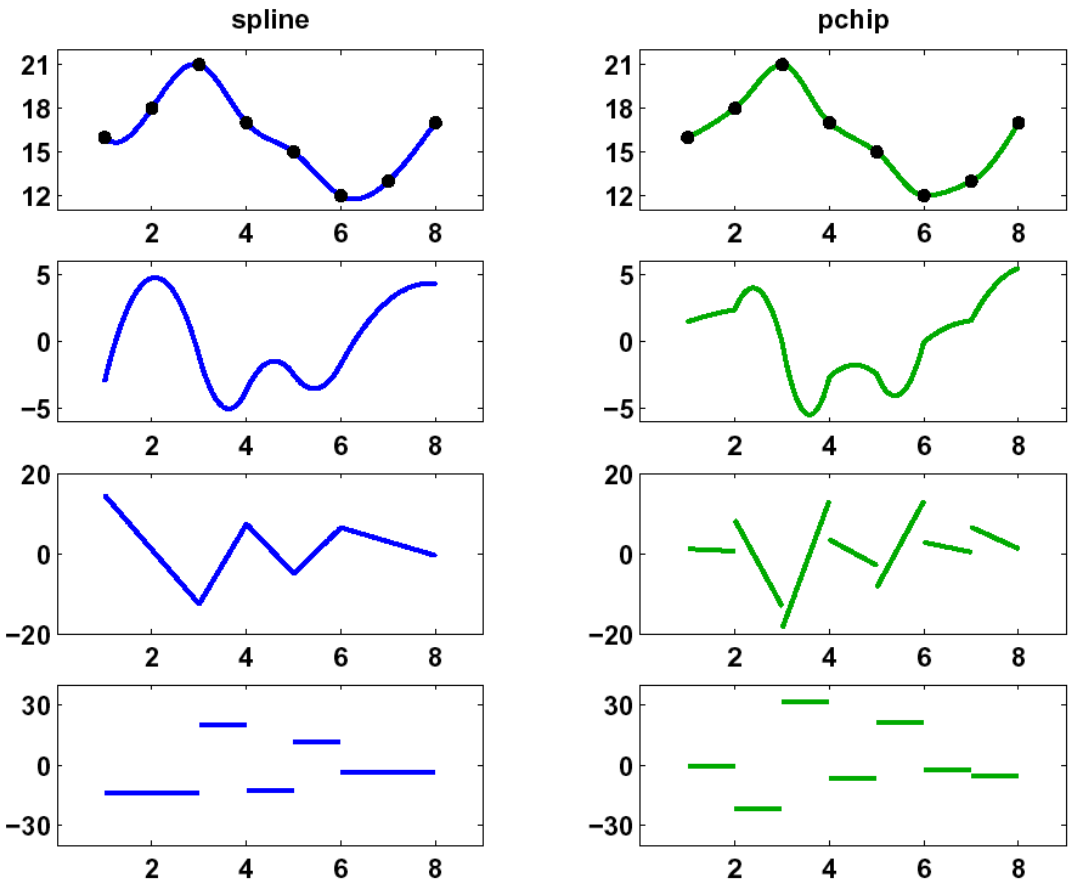
De momento esto lo olvidamos

P(x) = s, es el output.

*para ordenar los puntos de manera creciente, en matlab usamos el comando sort.

Por ejemplo si tenemos un vector X usamos un comando Y=sort(x), nos aparece un vector ordenado de menor a mayor.

miramos del capitulo 3:



Donde gráficamente reconocemos si una función es o no derivable por las propiedades estudiadas en cálculo.

Seguimos usando el programa splinetx.m y aplicamos en matlab.

*el punto antes de una operación significa, componente a componente.

También usamos el programa splineslopes:

function d = splineslopes(h,delta)

% SPLINESLOPES Slopes for cubic spline interpolation.

% splineslopes(h,delta) computes d(k) = S'(x(k)).

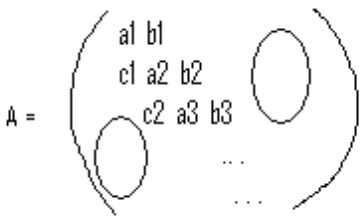
% Uses not−a−knot end conditions.

% Diagonals of tridiagonal system

```
n = length(h)+1;

a = zeros(size(h)); b = a; c = a; r = a;

a(1:n−2) = h(2:n−1);

a(n−1) = h(n−2)+h(n−1);

b(1) = h(2);

b(2:n−1) = 2*(h(2:n−1)+h(1:n−2));

b(n) = h(n−2);

c(1) = h(1)+h(2);

c(2:n−1) = h(1:n−2);

% Right−hand side

r(1) = ((h(1)+2*c(1))*h(2)*delta(1)+h(1)^2*delta(2))/c(1);

r(2:n−1) = 3*(h(2:n−1).*delta(1:n−2)+h(1:n−2).*delta(2:n−1));

r(n) = (h(n−1)^2*delta(n−2)+(2*a(n−1)+h(n−1))*h(n−2)*delta(n−1))/a(n−1);

% Solve tridiagonal linear system

d = tridisolve(a,b,c,r);
```

Usa una matriz tri−diagonal (con tres diagonales) y se soluciona gracias a tridisolve

Tridiagonal:



Trisolve(a,b,c,r)

Ax = r

x = A \ r

las condiciones descritas en nuestro ejemplo not−a−knot

$$A = \begin{pmatrix} h_2 & h_2+h_1 & & & \\ h_2 & 2(h_2+h_1) & h_1 & & \\ & \cdots & \cdots & \cdots & \\ & & h_{n-1} & 2(h_{n-1}+h_{n-2}) & h_{n-2} \\ & & & h_{n-1}+h_{n-2} & h_{n-2} \end{pmatrix}$$

Para la primera ecuación: h2d1+h2+a1d2 = = (hn−1+hn−2)dn−1+hn−2*dn=

Del capitulo 3 del libro de Mooler, página 22 :

3.11. Modify splinetx and pchiptx so that, if called with two output arguments,

they produce both the value of the interpolant and its ¯rst derivative. That

is,

[v,vprime] = pchiptx(x,y,u)

and

[v,vprime] = splinetx(x,y,u)

compute *P*(*u*) and *P0*(*u*).

[v,v1,v2,v3] = splintex4(x,y,n)

Creamos un nuevo programa a partir del splintex:

%Este programa resuelve el problema 3.11

% del libro de mooler(modificado)

function [v,v1,v2,v3] = splinetx3(x,y,u)

%SPLINETX Textbook spline function.

% v = splinetx(x,y,u) finds the piecewise cubic interpolatory

% spline S(x), with S(x(j)) = y(j), and returns v(k) = S(u(k)).

% and the first, second and third derivatives

%v1 , v2, v3.

% See SPLINE, PCHIPTX.

% First derivatives

h = diff(x);

delta = diff(y)./h;

```
d = splineslopes(h,delta);

% Piecewise polynomial coefficients

n = length(x);

c = (3*delta − 2*d(1:n−1) − d(2:n))./h;

b = (d(1:n−1) − 2*delta + d(2:n))./h.^2;

% Find subinterval indices k so that x(k) <= u < x(k+1)

k = ones(size(u));

for j = 2:n−1

k(x(j) <= u) = j;

end

% Evaluate spline and its three first derivatives.

s = u − x(k);

v = y(k) + s.*(d(k) + s.*(c(k) + s.*b(k)));

v1= d(k) +s.*(2*c(k)) + 3*s.*(b(k));

v2= 2+c(k)+6*s.*b(k);

v3= 6*b(k);

%−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
```

En command window:

```
>> x = [1 1.5 3 4.5 7 8.1 9.2 10]

x =

Columns 1 through 7

1.0000 1.5000 3.0000 4.5000 7.0000 8.1000 9.2000

Column 8

10.0000

>> y = [2 −3 4 6 2.3 4 −1 3 6 8]

y =
```

Columns 1 through 7

2.0000 −3.0000 4.0000 6.0000 2.3000 4.0000 −1.0000

Columns 8 through 10

3.0000 6.0000 8.0000

>> format short

>> x

x =

Columns 1 through 7

1.0000 1.5000 3.0000 4.5000 7.0000 8.1000 9.2000

Column 8

10.0000

>> u = 0.5:0.01:10.5;

>> [v,v1,v2,v3]=splinetx3(x,y,u)

>> [v,v1,v2,v3]=splinetx3(x,y,u);

>> plot(x,y,'r*',u,v2,'g',u,v3,'c')

>> plot(x,y,'r*',u,v,'r*',u,v1,'b',u,v2,'g',u,v3,'c')

Teniamos que tener el programa triidsolve:

function x = tridisolve(a,b,c,d)

% TRIDISOLVE Solve tridiagonal system of equations.

% x = TRIDISOLVE(a,b,c,d) solves the system of linear equations

% b(1)*x(1) + c(1)*x(2) = d(1),

% a(j−1)*x(j−1) + b(j)*x(j) + c(j)*x(j+1) = d(j), j = 2:n−1,

% a(n−1)*x(n−1) + b(n)*x(n) = d(n).

%

% The algorithm does not use pivoting, so the results might

% be inaccurate if abs(b) is much smaller than abs(a)+abs(c).

```
% More robust, but slower, alternatives with pivoting are:
% x = T\d where T = diag(a,−1) + diag(b,0) + diag(c,1)
% x = S\d where S = spdiags([[a; 0] b [0; c]],[−1 0 1],n,n)
x = d;
n = length(x);
for j = 1:n−1
mu = a(j)/b(j);
b(j+1) = b(j+1) − mu*c(j);
x(j+1) = x(j+1) − mu*x(j);
end
x(n) = x(n)/b(n);
for j = n−1:−1:1
x(j) = (x(j)−c(j)*x(j+1))/b(j);
end
```

cambiamos le valor de y:

```
>> y = [0 2 4 −1 −3 0 4 8 9 9];
>> u=0.9:0.01:10.1;
>> [v,v1,v2,v3] = splinetx3(x,y,u);
>> plot(x,y,'r*',u,v,'r*',u,v1,'b',u,v2,'g',u,v3,'c')
```

Mirar problem3.11

Mirar problem 3.9

18−ABRIL−06

<<Chapter 4>>

Teorema de los ceros de Bolzano.

*en matlab el poner ó es lo mismo que poner =.

En los apuntes:

Given a,b,f(a),f(b),e

while |b−a|>e|b|

x=a+b/2

if f(x) = 0 then

end

if f(x)f(a)<0 then

b=x

else

a=x

end

end

Hacemos un programa para hala ceros con el método de la bisección:

```
%este programa calcula los ceros
%de una función f(x) por el
%método de la bisección
%Inputs a,b,e
%a,b extremos del intervalo
%e error relativo admisible
% la función f(x) se escribe
%en un fichero auxiliar de nombre
%fun.m
function c = bisec(a,b,e)
fa = feval(`fun',a);
fb = feval(`fun',b);
while abs(b−a) > e*abs(b)
```

```
c = (a+b)/2;

fc = feval(`fun',c);

if fc == 0

break

end

if fc*fa < 0

b = c;

else

a = c;

end

end
```

y otro programa:

```
% este fichero sirve para almacenar la función

%f(x)

function y = fun(x)

y = sin(x.^2)−x.^3;
```

En matlab:

```
>> c = feval('fun',0)

c =

0

>> feval('fun',3)

ans =

−26.5879

>>>> sin(3^2)−3^3

ans =

−26.5879
```

```
>>>> feval('fun',8)

ans =

−511.0800

>> feval('fun',0.1)

ans =

0.0090

>> feval('fun',0)

ans =

0

>> c= bisec(0.1,3,10^−6)

c =

0.8960

>> format long

>> c= bisec(0.1,3,10^−6)

c =

0.89599368572235
```

Modificamos el primer programa:

```
%este programa calcula los ceros
%de una función f(x) por el
%método de la bisección
%Inputs a,b,e
%a,b extremos del intervalo
%e error relativo admisible
% la función f(x) se escribe
%en un fichero auxiliar de nombre
%fun.m
```

```
function [c,k] = bisec(a,b,e)

k=0;

fa = feval('fun',a);

fb = feval('fun',b);

while abs(b−a) > e*abs(b)

k = k+1;

c = (a+b)/2;

fc = feval('fun',c);

if fc == 0

break

end

if fc*fa < 0

b = c;

else

a = c;

end

end
```

modificamos también el segundo:

```
% este fichero sirve para almacenar la función

%f(x)

function y = fun(x)

y = x.^3 − 2.000001*x.^2+1.000002*x − 0.000001;

>> feval('fun',−10)

ans =

−1.210000121000000e+003

>> feval('fun',10)
```

```
ans =

8.099999190000000e+002

>> c = bisec(−10,10,10^−6)

c =

1.000000224848918e−006

>> u = −2:0.01:2;

>> v = feval('fun',u);

>> plot(u,v)

>> z= 0*u;

>> plot(u,v,u,z,'k')

>> c = bisec(−1,0.5,10^−12)

c =

1.000000000000350e−006

>> c = bisec(−1,0.5,10^−13)

c =

9.999999999999430e−007

>> c = bisec(−1,0.5,10^−14)

c =

9.999999999999989e−007

>> c = bisec(−1,0.5,10^−15)

c =

9.999999999999995e−007

>> c = bisec(−1,0.5,10^−16)

c =

1.000000000000000e−006

>> u = 0.5:0.001:1.5;
```

```
>> z= 0*u;

>> v = feval('fun',u);

>> plot(u,v,u,z,'k')

>>

>> feval('fun',1)

ans =

−8.226659269441623e−017

>>
```

Otro método es: EL MÉTODO DE NEWTON

Creamos el programa newton:

```
%Este programa obtiene los

%ceros de la función f(x)

% por el método de Newton

%Inputs: x0 (punto de partida),e (error relativo)

%Necesitamos dos ficheros auxiliares:

% − fun.m (donde guardamos la función f(x))

% − derfun.m (donde almacenamos la derivada

%f'(x) de la función f(x))

function [c,k] = newton(a,e)

k=0;

b=a;

fa=feval('fun',a);

dfa=feval('derfun',a);

c=a−fa/dfa;

while abs(c−b) > e*abs(b)

a=c;
```

b=a;

k = k+1;

fa = feval ('fun',a);

dfa=feval('derfun',a);

c= a−fa/dfa;

end

creamos el programa derfun:

%este fichero contiene la derivada de la función

%f(x) que esta guardada en el fichero fun.m

function dy = derfun(x)

%la otra función que usabamos era: dy = 2*cos (x.^2).*x−3*x.^2;

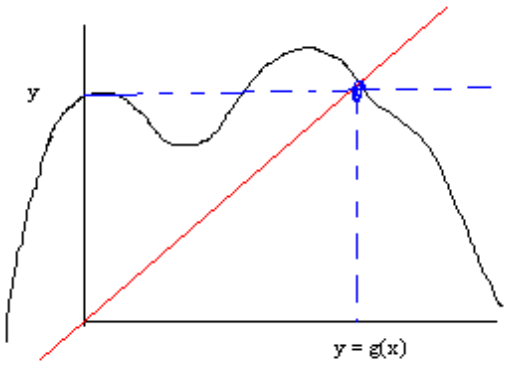
dy = 3*x^2−2*2.000001*x+1.000002;

20−abril−06

+página 10 del capítulo 5

Fix point method

(puntos fijos)



g(xo) = x1
g(x1)=x2
...
...
g(xn)= xn+1

xn--------->X*
n-->infi

lim xn = x
g(X*)= lim g(xn) = lim xn+1
= X*

(el lim tiende a infinito)

0 = f(x) =
g(x) - x

El algoritmo de punto fijo dice:

Dados x y e mientras que |g(x) −x | > |x|:

Given x

While |g(x) −x | > |x|

X g(x)

End

***** Creamos un programa llamado fijo.m: *******

%este programa calcula los puntos

%fijos de la función g(x)

% g(x*) = x*

% la función g(x) esta almacenada en el fichero fun.m

%inputs: dato inicio x0, error relativo (epsilon)

% outputs: punto fijo x*, numero de iteraciones k

function [x,k] = fijo(x0,e)

k = 0;

x= x0;

gx = feval('fun',x);

% en feval('fun',x); evalua g(x)

while abs(gx − x) > e*abs(x)

x = gx;

gx=feval('fun',x);

k = k+1;

end

y el programa:

function y = fun(x)

y = cos(x);

y ejecuto en matlab:

>> [x,k]=fijo (0,10^−6)

x =

0.7391

k =

35

Y con el programa derfun Nuevo:

%este fichero contiene la derivada de la función

%f(x) que esta guardada en el fichero fun.m

function dy = derfun(x)

%la otra función que usabamos era: dy = 2*cos (x.^2).*x−3*x.^2;

%dy = 3*x^2−2*2.000001*x+1.000002;

dy = −sin(x)−1;

en matlab:

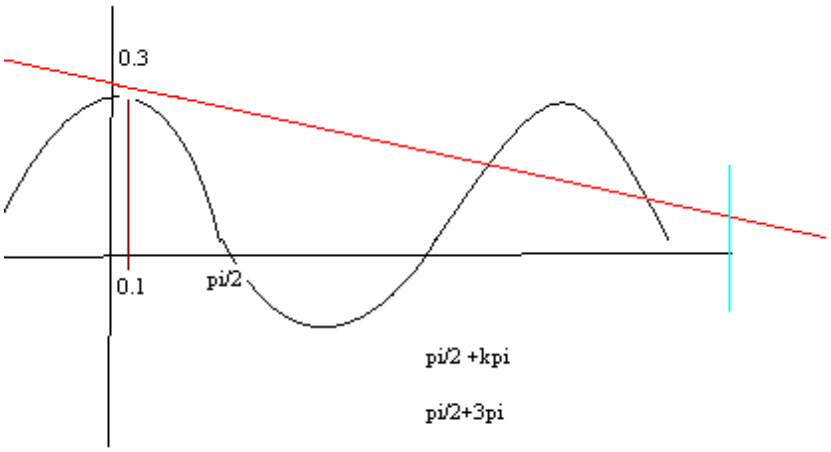
>> [x,k] = newton(0.1,10^−6)

x =

0.7391

k =

4



26−abril−06

+apuntes libro moler pagina zero página 20 en adelante

Con los programas:

derfun:

```
function dy = dfun(x)

% y = cos(x);

dy = 3*x^2−2 −5;
```

y:

fun:

```
function y = fun(x)

%y = cos(x)−x;

y = x.^3−2.*x−5;
```

En matlab:

```
>> u = 0:0.01:3;

>> v = fun(u);

>> z=0*u;

>> plot(u,v,u,z,'k');

>>

>> [c,k] = bisec(0,3,eps)

c =

2.0946

k =

53

>> [c,k] = newton(0,eps)

c =

2.0946

k =

19
```

```
>>
```

Donde bisec:

```
%Este programa calcula los ceros de una funcion f(x)
%por el metodo de la biseccion
%e error relativo admisible
%la funcion f(x) se escribe en un fichero auxiliar
%con el nombre fun.m
function [x,p] = bisec(a,b,e)
fa = feval('fun',a); %feval evalua la funcion en el punto dado
fb = feval('fun',b);
p = 0;
while abs(b−a) > e*abs(b)
p = p+1;
x = (a+b)/2;
fx = feval('fun',x);
if fx == 0
break
end
if fx*fa < 0
b = x;
else a = x;
end
end
```

Donde newton:

```
%Este programa obtiene los ceros de la funcion f(x)
%por el metodo de Newton
```

```
%x0 = punto de partida, e = error relativo
%necesitamos dos ficheros auxiliares
%fun.m(donde guardamos la funcion f(x)
%derfun.m(donde almacenamos la derivada
%f'(x) de la funcion f(x)
function [c,k] = newton(x,e)
k = 0;
b = x;
fx = feval('fun',x);
dfx = feval('derfun',x);
c = x − fx/dfx;
while abs(c−b) > e*abs(b);
x = c;
b = x;
k = k+1;
fx = feval('fun',x);
dfx = feval('derfun',x);
c = x−fx/dfx;
end
```

la nueva function derfun:

```
function dy = derfun(x)
% y = cos(x);
dy = 1./(1+x.^2);
>> [c,k] = newton(0,eps)
c =
1.7321
```

```
k =

6

>>
```

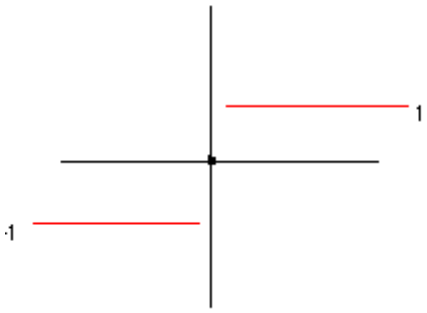
Ahora la funcion fun:

```
function y = fun(x)

y = sign(x−2).*sqrt(abs(x−2));

>> v = fun(u);

>> u = 0:0.01:4;

>> v = fun(u);

>> z=0*u;

>> plot(u,v,u,z,'k');
```

Ahora derfun:

```
function dy = derfun(x)

% y = cos(x);

%dy = 1./(1+x.^2);

dy = 1./(2*sqrt(abs(x−2)));

>> [c,k] = bisec(0,4,eps)

c =

2

k =

1

>>

>> v = fun(u);

>> u = 0:0.01:4;

>> v = fun(u);

>> z=0*u;
```

```
>> plot(u,v,u,z,'k');

>> [c,k] = bisec(0,4,eps)

c =

2

k =

1

[c,k] = newton (0,eps)
```



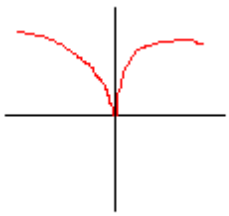
f(x) =
sign(x-2)raiz(abs(x-2))

f'(x)=
si x>a-> f(x) = raiz(x-a)
f '(x)= 1/(2*raiz(x-a))
si x <a f(x) = -raiz(a-x)
f ' (x)= -1/(2*raiz(z-x))
= 1/(2*raiz(a-x))

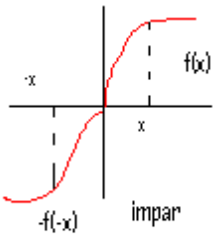
f ' (x) = 1/(2*raiz(abs(x-a))) x distinto de a

f(x) = feval ('fun',x)
f ' (x)

f(x) = f(-x) par
f(x) = -f(-x) impar



par



impar

```
>> u = −20:0.01:20;

>> v = besselj(0,u);

>> z=0*u;

>> plot(u,v,u,z,'k');

Warning: Imaginary parts of complex X and/or Y arguments ignored.
```

```
>>

>> u = 0:0.01:20;

>> v = besselj(0,u);

>> z=0*u;

>> plot(u,v,u,z,'k');

>> u = 0:0.01:40;

>> v = besselj(0,u);

>> z=0*u;

>> plot(u,v,u,z,'k');

>> u = −69:0.01:69;

>> v = besselj(0,u);

>> z=0*u;

>> plot(u,v,u,z,'k');
```

Cambio fun:

```
function y = fun(x)

%y = cos(x)−x;

%y = x.^3−2.*x−5;

%y = atan(x)−pi/3;

%y = sign(x−2).*sqrt(abs(x−2));

y = besselj(0,x);
```

y creo el problema 4 el programa:

```
% este pequeño programa (script)

%resuelve el problema 4.10

% del libro de mooler, buscamos

%los 10 primeros ceros de la función

% de Bessel de primera especie
```

```
%de orden 0, J_0(x),
%Para ello usamos el metodo de la bisección
%aplicado a los intervalos [1,4],
% [4,7],...,[29,32]
%Los intervalos los hemos obtenido
%por inspección representando graficamente
%la función.
x = [1 4 7 10 13 17 20 24 27 29 32];
c = zeros(10,1);
for n = 1:10
c(n) = bisec(x(n),x(n+1),eps)
end
*que no lleva function porque no tiene inputs
>> problem4_10
c =
2.4048
0
0
0
0
0
0
0
0
0
0
c =
```

2.4048

5.5201

0

0

0

0

0

0

0

0

c =

2.4048

5.5201

8.6537

0

0

0

0

0

0

0

0

c =

2.4048

5.5201

8.6537

11.7915

0

0

0

0

0

0

c =

2.4048

5.5201

8.6537

11.7915

14.9309

0

0

0

0

0

c =

2.4048

5.5201

8.6537

11.7915

14.9309

18.0711

0

0

0

0

c =

2.4048

5.5201

8.6537

11.7915

14.9309

18.0711

21.2116

0

0

0

c =

2.4048

5.5201

8.6537

11.7915

14.9309

18.0711

21.2116

24.3525

0

0

c =

2.4048

5.5201

8.6537

11.7915

14.9309

18.0711

21.2116

24.3525

27.4935

0

c =

2.4048

5.5201

8.6537

11.7915

14.9309

18.0711

21.2116

24.3525

27.4935

30.6346

>> besselj(0,c(1))

ans =

0

>>

09/05/06

Capitulo 5 Last squares (minimos cuadrados)

Recordamos:

Reales(n) Complejos(n)

(,) < , >

Linealidad:

< x, y+z > = <x,y> + <x,z>

<x,By> = B <x,y>

<x,x> mayor o igual que cero, <x,y> + <x,z>

<x,y> = 1/ <x,y>

(x,y) = xT * y ; <x,y> = media(xT) * y

[e1,e2,,en] ortonormal

<ei,ej> = dij = 1 si i = j; 0 si i distinto de j

En matlab:

>> Y = [−1,3,0];

>> X=[1,2,3];

>> X*Y'

ans =

5

>> X=[i,−1,0]

X =

0 + 1.0000i −1.0000

X =

0 + 1.0000i −1.0000 0

>> Y = [−2; sqrt(2) + i; 1]

Y =

−2.0000

1.4142 + 1.0000i

1.0000

y =

−2.0000

1.4142 + 1.0000i

1.0000

>> Y'*Y

ans =

8

>> X= [i;−1;0]

X =

0 + 1.0000i

−1.0000

0

>> X'*X

ans =

2

>> X'*Y

ans =

−1.4142 + 1.0000i

>> norm(X)

ans =

1.4142 (es raís de2)

>> norm(Y)

ans =

- (es raíz de8)

*tenemos un vector fila: (v1,.,vn) no ortonormal

aplicamos Grand Smith (q1,,qn) ortonormal

<qi,qj> = dij

Peo no nos sirve Grand Smith, asi que usamos otros métodos. En matlab tenemos la factorización de QR.

Es decir: tenemos = |v1|, |vn| qr Q 0 |q1|,|qn|

En matlab:

>> A= rand(4)

A =

0.9501 0.8913 0.8214 0.9218

0.2311 0.7621 0.4447 0.7382

0.6068 0.4565 0.6154 0.1763

0.4860 0.0185 0.7919 0.4057

>> rank(A)

ans =

4

>>> v1 = A(:,1)

v1 =

0.9501

0.2311

0.6068

0.4860

>> v2 = A(:,2)

v2 =

0.8913

0.7621

0.4565

0.0185

```
>> v3 = A(:,3)

v3 =

0.8214

0.4447

0.6154

0.7919

>> v4 = A(:,4)

v4 =

0.9218

0.7382

0.1763

0.4057

>> v2'*v3

ans =

1.3666

>> v1'*v4

ans =

1.3506

>> [Q,R]= qr(A)

Q =

−0.7606 −0.1354 0.4523 0.4457

−0.1850 −0.8151 −0.5489 −0.0062

−0.4858 0.0754 0.0617 −0.8686

−0.3890 0.5582 −0.7002 0.2163

R =

−1.2492 −1.0478 −1.3141 −1.0811
```

0 −0.6971 0.0148 −0.4867

0 0 −0.3892 −0.2615

0 0 0 0.3409

q3 =

0.4523

−0.5489

0.0617

−0.7002

>> q4=Q(:,4)

q4 =

0.4457

−0.0062

−0.8686

0.2163

>> q1 =

−0.7606

−0.1850

−0.4858

−0.3890

>> q2=Q(:,2)

q2 =

−0.1354

−0.8151

0.0754

0.5582

>> q2'*q3

ans =

5.5511e−017

>> q1'*q4

ans =

1.3878e−017

>> q1'*q1

ans =

1.0000

>> q4'*q4

ans =

1.0000

*perpendicular X e Y : X_|_ Y

En los apuntes:

Classical Gram−Schmidt algorithm:

for j = 1:n

vj = aj

for k = 1:j−1

rjk = aj*qk

vj= vj − rjk qk

end

rjj = ||vj||

qj = vj/rjj

end

Exercise: Study the numerical stability of the

classical GS algorithm.

Exercise: Modify the classical GS algorithm

to correct its numerical unstability.

Gram−Schmidt

orthogonallization method:

Initial basis *B = {ai }*

ui ð

vi

ri

, ri

2

= vi ,vi

vi ð *ai* ð *ai ,uk uk*

*k*ð1

*i*ð1

ð

Classical Gram−Schmidt algorithm:

for j = 1:n

vj = aj

for k = 1:j−1

rjk = aj*qk

vj= vj − rjk qk

end

rjj = ||vj||

qj = vj/rjj

end

Exercise: Study the numerical stability of the

classical GS algorithm.

Exercise: Modify the classical GS algorithm

to correct its numerical unstability.

QR factorization:

A = QR

Q = [u1 u2 un] orthogonal matrix

R = [Rij] upper triangular matrix

Tengo los siguientes programas:

Demo_ortog:

```
% This script explores the lost of orthogonality

% due to the intrinsic numerical unstability

% of the classical and

% modified Gram−Schmidt algorithms.

clc

disp(' Losing orthogonality on Gram−Schmidt algorithms ')

disp(' delta Error on QR Error on CGS Error on MGS ')

disp('−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−')

for delta = logspace(−4,−16,7)

A = [1+delta 2;1 2];

[Q1,R1]= qr(A);

[Q2,R2]= cgs(A);

[Q3,R3]= mgs(A);

error1 = norm(Q1'*Q1 − eye(2));

error2 = norm(Q2'*Q2 − eye(2));

error3 = norm(Q3'*Q3 − eye(2));

disp(sprintf(' %5.0e %20.15f %20.15f %20.15f ', delta,error1,error2,error3))

end
```

cgs:

```
% Gram−Schmidt orthogonalization (unstable)
function [Q,R] = cgs(A)
n = length(A);
% starting variables
R = zeros(n,n);
Q = zeros(n,n);
V = zeros(n,n);
% Normalizing the first column vector A
R(1,1)=norm(A(:,1));
Q(:,1)= A(:,1)/R(1,1);
% Classical Gram−Schmidt
for j = 2:n
V(:,j) = A(:,j);
for i = 1:j−1
R(i,j) = Q(:,i)'*A(:,j);
V(:,j)= V(:,j)−R(i,j)*Q(:,i);
end
R(j,j)=norm(V(:,j));
Q(:,j)=V(:,j)/R(j,j);
end
Q;
R;
```

Mgs:

```
% Modified Gram−Schmidt algorithm (stable)
function [Q,R] = mgs(A)
```

```
[m,n] = size(A);

% Initializating variables

R = zeros(m,n);

S = zeros(m,m);

Q = zeros(m,m);

V = [A floor(10*rand(m,m−n))];

for i = 1:n

R(i,i)=norm(V(:,i));

Q(:,i)=V(:,i)/R(i,i);

for j = i+1:n

R(i,j) = Q(:,i)'*V(:,j);

V(:,j)= V(:,j)−R(i,j)*Q(:,i);

end

end

for i = n+1:m

S(i,i)=norm(V(:,i));

Q(:,i)=V(:,i)/S(i,i);

for j = i+1:n

S(i,j) = Q(:,i)'*V(:,j);

V(:,j)= V(:,j)−S(i,j)*Q(:,i);

end

end

Q;

R;
```

En matlab:

Demo_ortog

Losing orthogonality on Gram−Schmidt algorithms

delta Error on QR Error on CGS Error on MGS

−−

1e−004 0.000000000000000 0.000000000002109 0.000000000002109

1e−006 0.000000000000001 0.000000000399139 0.000000000399139

1e−008 0.000000000000000 0.000000028306691 0.000000028306691

1e−010 0.000000000000000 0.000003330622496 0.000003330622496

1e−012 0.000000000000000 0.000444049689291 0.000444049689291

1e−014 0.000000000000000 0.032949132854297 0.032949132854297

1e−016 0.000000000000000 1.000000000000000 1.000000000000000

```
>> A= [1 + 10^−14 2;1 2]

A =

1.0000 2.0000

1.0000 2.0000

>> format long

>> A= [1 + 10^−14 2;1 2]

A =

1.00000000000001 2.00000000000000

1.00000000000000 2.00000000000000

>> [Q,R] = cgs(A)

Q =

0.70710678118655 −0.68342428808113

0.70710678118654 0.73002139863212

R =

1.41421356237310 2.82842712474619

0 0.00000000000001
```

```
>> q1=Q(:,1)

q1 =

0.70710678118655

0.70710678118654

>> q2=Q(:,2)

q2 =

−0.68342428808113

0.73002139863212

>> q1'*q1

ans =

1.00000000000000

>> q2'*q2

ans =

1.00000000000000

>> q1'*q2

ans =

0.03294913285430

>> plot(q1(1),q1(2),'r*',q2(1),q2(2),'b*')

>> plot(q1(1),q1(2),'r*',q2(1),q2(2),'b*',0,0,'k*')
```

*nota : hemos añadido un punto en el (0,0)

Con el programa demo_cgs_mgs.m:

```
% Numerical comparison:

% Classical Gram−Schmidt and

% modified Gram−Schmidt

% First we generate a random matrix A

% with eigenvalues spaced by a factor 2
```

```
% between 2^−1 and 2^−80

[U,X]= qr(randn(80));

S = diag(2.^(−1:−1:−80));

A = U*S*U';

E = eig(A);

% We use now classical Gram−Schmid cgs(A)

% and modified Gram−Schmidt mgs(A) to compute Q and R

[Q1,R1]=cgs(A);

[Q2,R2]=mgs(A);

% We represent the diagonal R elements

% in logarithmic scale for both matrices R1 and R2.

x = 1:80;

y1 = log2(diag(R1));

y2 = log2(diag(R2));

y6 = log2(−sort(−abs(E)));

y3 = −23*ones(1,80);

y4 = −55*ones(1,80);

y5 = −x;

plot(x,y1,'bo',x,y2,'rx',x,y6,'g*',x,y3,':',x,y4,':',x,y5,'−');

xlabel('R(i,i) elements in logarithmic scale');
```

en matlab:

```
>> demo_cgs_mgs

>> 2^−55

ans =

2.775557561562891e−017
```

11−05−06

AX = B

A = Q * R = triángular superior

Q columnas (ó filas) son una base ortonormal, se llama ortogonal.

Q' * Q = Q * Q' = I

Q' Q RX = Q' B ; como Q' Q = I RX = Q' B

5. Singular value decomposition

(Página 7 del capitulo 5)

A = Q R = U S V'

Donde S es diagonal, y U y V' ortgonales

|s1 0 |

S = | s2 |

| 0 s3 |

Real Ortgonal

Compleja Unitaria

Ambas tmatrices tienen las columnas en bases ortonormales.

En marlab sería: >> [u,s,v] = svd(A) s1>>s2>>>>sn

Hacemos en matlab:

>> help svd

SVD Singular value decomposition.

[U,S,V] = SVD(X) produces a diagonal matrix S, of the same

dimension as X and with nonnegative diagonal elements in

decreasing order, and unitary matrices U and V so that

X = U*S*V'.

S = SVD(X) returns a vector containing the singular values.

[U,S,V] = SVD(X,0) produces the "economy size"

decomposition. If X is m−by−n with m > n, then only the

first n columns of U are computed and S is n−by−n.

For m <= n, SVD(X,0) is equivalent to SVD(X).

[U,S,V] = SVD(X,'econ') also produces the "economy size"

decomposition. If X is m−by−n with m >= n, then it is

equivalent to SVD(X,0). For m < n, only the first m columns

of V are computed and S is m−by−m.

See also svds, gsvd.

Overloaded functions or methods (ones with the same name in other directories)

help sym/svd.m

Reference page in Help browser doc svd

>> A = rand(3)

A =

0.9501 0.4860 0.4565

0.2311 0.8913 0.0185

0.6068 0.7621 0.8214

>> [u,s,v]= svd(A)

u =

−0.6068 0.4443 −0.6591

−0.4007 −0.8871 −0.2290

−0.6865 0.1251 0.7163

s =

1.8109 0 0

0 0.6319 0

0 0 0.3748

v =

−0.5995 0.4637 −0.6523

−0.6489 −0.7587 0.0572

−0.4684 0.4576 0.7558

>> >> u*u'

ans =

1.0000 0 −0.0000

0 1.0000 0.0000

−0.0000 0.0000 1.0000

>> A − u*s*v'

ans =

1.0e−015 *

0.1110 −0.3886 0.1110

−0.3608 0 −0.3192

−0.3331 −0.6661 −0.1110

>> norm(A − u*s*v')

ans =

8.2930e−016

>> A = [1 0 0; 0 0 1; 1 1 1]

A =

1 0 0

0 0 1

1 1 1

>> spy(A)

>> A = [1 1 1 1; 1 1 1 1; 1 1 0 0; 1 1 1 1;1 1 0 0;1 1 1 1 ;1 1 1 1]

A =

1 1 1 1

1 1 1 1

1 1 0 0

1 1 1 1

1 1 0 0

1 1 1 1

1 1 1 1

>> spy(A)

En editor:

function L = letterL

L = [1 1 0 0 0 0;

1 1 0 0 0 0;

1 1 0 0 0 0;

1 1 0 0 0 0;

1 1 0 0 0 0;

1 1 1 1 1 1;

1 1 1 1 1 1];

function O = letterO

O = [1 1 1 1 1 1;

1 1 1 1 1 1;

1 1 0 0 1 1;

1 1 0 0 1 1;

1 1 0 0 1 1;

1 1 0 0 1 1;

1 1 1 1 1 1;

1 1 1 1 1 1];

function E = letterE

```
E = [1 1 1 1 1 1;

1 1 1 1 1 1;

1 1 1 0 0 0;

1 1 1 1 1 1;

1 1 1 1 1 1;

1 1 1 0 0 0;

1 1 1 1 1 1 ;

1 1 1 1 1 1];

function H = letterH

H = [1 1 0 0 1 1;

1 1 0 0 1 1;

1 1 0 0 1 1;

1 1 1 1 1 1;

1 1 1 1 1 1;

1 1 0 0 1 1;

1 1 0 0 1 1;

1 1 0 0 1 1];
```

Hacemos el programa:

```
%este script nos escribe la palabra "HELLO"

% por una matriz formada con 1 y 0

function h = hello

h = zeros(12,42);

h(3:10,3:8)= h(3:10,3:8) + letterH;

h(3:10,11:16)= h(3:10,11:16) + letterE;

h(3:10,19:24)= h(3:10,19:24) + letterL;

h(3:10,27:32)= h(3:10,27:32) + letterL;
```

h(3:10,35:40)= h(3:10,35:40) + letterO;

>> spy(hello)

>> [U,S,V]= svd(hello);

>> diag(S)

ans =

12.0087

3.7201

2.1501

1.8247

0.0000

0.0000

0.0000

0.0000

0

0

0

0

>> pcolor(hello)

U1 =

0 0 0 0

0 0 0 0

−0.3598 −0.0869 0.1134 0.5917

−0.3598 −0.0869 0.1134 0.5917

−0.2923 −0.2906 −0.5635 −0.1124

−0.3483 −0.3132 0.4093 −0.3362

−0.3483 −0.3132 0.4093 −0.3362

−0.2923 −0.2906 −0.5635 −0.1124

−0.4047 0.5567 −0.0461 −0.1555

−0.4047 0.5567 −0.0461 −0.1555

0 0 0 0

0 0 0 0

>> V1 =V(:,1:4);

>> S1= S(1:4,1:4);

>> S1

S1 =

12.0087 0 0 0

0 3.7201 0 0

0 0 2.1501 0

0 0 0 1.8247

>> hello = U1*S1*V1';

>> pcolor(hello)

>> U2 = U(:,1:2);

>> V2 =V(:,1:2);

>> S2= S(1:2,1:2);

>> hello3 = U2*S2*V2'

16/05/06

(Tema 6)

CHAPTER 6. Quadratures

Tenemos el programa PesosNC:

% Este fichero almacena los

% pesos de las reglas de

% cuadratura de Newton−Cotes

```
% con m puntos
%
% m es un entero que satisface 2 <= m <= 11
%
% p es un vector columna que contiene
% los pesos para la regla de cuadratura
% de Newton−Cotes con m puntos.
%
function p = pesosNC(m);
if m==2
p=[1 1]'/2;
elseif m==3
p=[1 4 1]'/6;
elseif m==4
p=[1 3 3 1]'/8;
elseif m==5
p=[7 32 12 32 7]'/90;
elseif m==6
p=[19 75 50 50 75 19]'/288;
elseif m==7
p=[41 216 27 272 27 216 41]'/840;
elseif m==8
p=[751 3577 1323 2989 2989 1323 3577 751]'/17280;
elseif m==9
p=[989 5888 −928 10496 −4540 10496 −928 5888 989]'/28350;
elseif m==10
```

p=[2857 15741 1080 19344 5778 5778 19344 1080 15741 2857]'/89600;

else

p=[16067 106300 −48525 272400 −260550 427368 −260550 272400 −48525 106300 16067]'/598752;

end;

regla se simpson

en matlab:

>> pesosnc(3)

ans =

0.1667

0.6667

0.1667

>> pesosnc(5)

ans =

0.0778

0.3556

0.1333

0.3556

0.0778

Con el programa simpson:

% This function computes the

% Simpson quadrature rule for

% computing the integral of f(x) on

% the interval [a,b]

function I = simpson(fname,a,b)

fa = feval(fname,a);

fm = feval(fname, (a+b)/2);

```
fb = feval(fname, b);

I = ( fa + 4*fm + fb)*(b−a)/6;
```

Y el programa fname:

```
%

function y = fname(x)

y = 2 − 3*x + 4*x.^2;
```

En matlab:

```
>> I = simpson('fname',1,4)

I =

67.5000
```

Cambiamos el programa:

```
%

function y = fname(x)

%y = 2 − 3*x + 4*x.^2;

y = 1 − x.^4;

>> I = simpson('fname',0,2)

I =

−4.6667

>> 22/5

ans =

4.4000
```

Otro programa: simpsoncom:

```
% This function computes

% the Simpson composite quadrature rule

% with n subintervals

function I = simpsoncom(fname,a,b,n);
```

```
h = (b−a)/n;

x = [a:h:b];

f = feval(fname, x);

I = 0;

for k = 1:n

I = I + simpson('fname',x(k),x(k+1));

End
```

Y cuadraturaNC:

```
% Este fichero calcula la cuadratura

% de Newton−Cotes de la funcion fname

% con m puntos y extremos del intervalo a y b.

% a,b numeros reales

% m entero entre 2 <= m <=11

% El computo devuelve el numero I

%

function I = cuadraturaNC(fname,a,b,m)

p = pesosNC(m);

%x = linspace(a,b,m)';

h = (b−a)/(m−1);

x = (a:h:b)';

f = feval(fname,x);

I = (b−a)*(p'*f);

>> I = cuadraturaNC('fname',0,2,3)

Warning: Function call pesosNC invokes inexact match E:\computacion estadistica\16−05−06\pesosnc.m.

> In cuadraturaNC at 9

I =
```

−4.6667

>> I = cuadraturaNC('fname',0,2,4)

I =

−4.5185

>> I = cuadraturaNC('fname',0,2,5)

I =

−4.40000000000000

>> 22/5

ans =

4.40000000000000

Cambiamos fname:

%

function y = fname(x)

%y = 2 − 3*x + 4*x.^2;

%y = 1 − x.^4;

y = sin(x);

>> I = cuadraturaNC('fname',0,pi,3)

I =

2.09439510239320

>> I = cuadraturaNC('fname',0,pi,5)

I =

1.99857073182384

>> I = cuadraturaNC('fname',0,pi,11)

I =

2.00000000114677

>> I = simpsoncom('fname',0,pi,2)

I =

2.00455975498442

>> I = simpsoncom('fname',0,pi,10)

I =

2.00000678444180

>> I = simpsoncom('fname',0,pi,100)

I =

2.00000000067647

>> I = simpsoncom('fname',0,pi,1000)

I =

2.00000000000007

>> I = simpsoncom('fname',0,pi,10000)

I =

2

*nota, para borrar la pantalla: clear clc

Tenemos cuadraturacom;

% Este fichero calculo la cuadratura

% de Newton−Cotes compuesta con m nodos

% y n subintervalos

% a,b numeros reales, a < b

% m numero de puntos, 2 <= m <=11

% n numero de subintervalos

%

% El programa calcula

% I La aproximacion de Newton−Cotes compuesta

% de la integral de f(x) entre a y b.

```
% La regla es aplicada a cada uno de los
% n subintervalos iguales de [a,b].
%
function I = cuadcompNC(fname,a,b,m,n)
Delta = (b−a)/n;
h = Delta/(m−1);
x = a+h*(0:(n*(m−1)))';
p = pesosNC(m);
%x = linspace(a,b,n*(m−1)+1)';
f = feval(fname,x);
I = 0;
first = 1;
last = m;
for i=1:n
%A−adimos al producto interno el valor en
% el subintervalo i−esimo.
I = I + p'*f(first: last);
first = last;
last = last+m−1;
end
I = Delta*I;
```

Cambio fname:

```
%
function y = fname(x)
%y = 2 − 3*x + 4*x.^2;
%y = 1 − x.^4;
```

```
%y = sin(x);

y = sin(x)./x;
```

con el programa cuadraturacomNC:

```
% Este fichero calcula la cuadratura
% de Newton−Cotes de la funcion fname
% con m puntos y extremos del intervalo a y b.
% a,b numeros reales
% m entero entre 2 <= m <=11
% El computo devuelve el numero I
%
function I = cuadraturaNC(fname,a,b,m)
p = pesosNC(m);
%x = linspace(a,b,m)';
h = (b−a)/(m−1);
x = (a:h:b)';
f = feval(fname,x);
I = (b−a)*(p'*f);
```

18/05/06

Tema 6 del book moler

Ejercicio 5.9

5.9. *Statistical Reference Datasets.* NIST, the National Institute of Standards and Technology, is the branch of the U.S. Department of Commerce responsible for setting national and international standards. NIST maintains Statistical Reference Datasets, StRD, for use in testing and certifying statistical soft− ware. The home page on the Web is [3]. Data sets for linear least squares are under \Linear Regression." This exercise involves two of the NIST reference

data sets.

² Norris. Linear polynomial for calibration of ozone monitors.

² Pontius. Quadratic polynomial for calibration of load cells.

For each of these data sets, follow the Web links labeled

² Data ¯le (ASCII Format)

² Certi¯ed Values

² Graphics

Download each ASCII ¯le. Extract the observations. Compute the polyno−

mial coe±cients. Compare the coe±cients with the certi¯ed values. Make

plots similar to the NIST plots of both the ¯t and the residuals.

Copiamos de: http://www.itl.nist.gov/div898/strd/lls/data/LINKS/DATA/Norris.dat

NIST/ITL StRD

Dataset Name: Norris (Norris.dat)

File Format: ASCII

Certified Values (lines 31 to 46)

Data (lines 61 to 96)

Procedure: Linear Least Squares Regression

Reference: Norris, J., NIST.

Calibration of Ozone Monitors.

Data: 1 Response Variable (y)

1 Predictor Variable (x)

36 Observations

Lower Level of Difficulty

Observed Data

Model: Linear Class

2 Parameters (B0,B1)

y = B0 + B1*x + e

Certified Regression Statistics

Standard Deviation

Parameter Estimate of Estimate

B0 −0.262323073774029 0.232818234301152

B1 1.00211681802045 0.429796848199937E−03

Residual

Standard Deviation 0.884796396144373

R−Squared 0.999993745883712

Certified Analysis of Variance Table

Source of Degrees of Sums of Mean

Variation Freedom Squares Squares F Statistic

Regression 1 4255954.13232369 4255954.13232369 5436385.54079785

Residual 34 26.6173985294224 0.782864662630069

Data: y x

0.1 0.2

338.8 337.4

118.1 118.2

888.0 884.6

9.2 10.1

228.1 226.5

668.5 666.3

998.5 996.3

449.1 448.6

778.9 777.0

559.2 558.2

0.3 0.4

0.1 0.6

778.1 775.5

668.8 666.9

339.3 338.0

448.9 447.5

10.8 11.6

557.7 556.0

228.3 228.1

998.0 995.8

888.8 887.6

119.6 120.2

0.3 0.3

0.6 0.3

557.6 556.8

339.3 339.1

888.0 887.2

998.5 999.0

778.9 779.0

10.2 11.1

117.6 118.3

228.9 229.2

668.4 669.1

449.2 448.9

0.2 0.5

*Para importer datos: File Import Data seleccionas los datos que quieres importar

aceptas a todo y finalizas luego en matlab pones el nombre del archivo y se abre en el programa.

En matlab:

>> y = norris2(:,1)

y =

0.1000

338.8000

118.1000

888.0000

9.2000

228.1000

668.5000

998.5000

449.1000

778.9000

559.2000

0.3000

0.1000

778.1000

668.8000

339.3000

448.9000

10.8000

557.7000

228.3000

998.0000

888.8000

119.6000

0.3000

0.6000

557.6000

339.3000

888.0000

998.5000

778.9000

10.2000

117.6000

228.9000

668.4000

449.2000

0.2000

>> x = norris2(:,2)

x =

0.2000

337.4000

118.2000

884.6000

10.1000

226.5000

666.3000

996.3000

448.6000

777.0000

558.2000

0.4000

0.6000

775.5000

666.9000

338.0000

447.5000

11.6000

556.0000

228.1000

995.8000

887.6000

120.2000

0.3000

0.3000

556.8000

339.1000

887.2000

999.0000

779.0000

11.1000

118.3000

229.2000

669.1000

448.9000

0.5000

```
>> n = length(x)

n =

36

>> A = ones(36,2);

>> A(:,2) = x;

>> A

A =

1.0000 0.2000

1.0000 337.4000

1.0000 118.2000

1.0000 884.6000

1.0000 10.1000

1.0000 226.5000

1.0000 666.3000

1.0000 996.3000

1.0000 448.6000

1.0000 777.0000

1.0000 558.2000

1.0000 0.4000

1.0000 0.6000

1.0000 775.5000

1.0000 666.9000

1.0000 338.0000

1.0000 447.5000

1.0000 11.6000

1.0000 556.0000
```

1.0000 228.1000

1.0000 995.8000

1.0000 887.6000

1.0000 120.2000

1.0000 0.3000

1.0000 0.3000

1.0000 556.8000

1.0000 339.1000

1.0000 887.2000

1.0000 999.0000

1.0000 779.0000

1.0000 11.1000

1.0000 118.3000

1.0000 229.2000

1.0000 669.1000

1.0000 448.9000

1.0000 0.5000

>> rank(A)

ans =

2

>> rank([A y])

ans =

3

>> beta = A\y

beta =

−0.2623

1.0021

>> format long

>> beta = A\y

beta =

−0.26232307377407

1.00211681802045

Ahora pegamos de la misma página :

Pointius:

NIST/ITL StRD

Dataset Name: Pontius

File Format: ASCII

Certified Values (lines 31 to 47)

Data (lines 61 to 100)

Procedure: Linear Least Squares Regression

Reference: Pontius, P., NIST.

Load Cell Calibration.

Data: 1 Response Variable (y)

1 Predictor Variable (x)

40 Observations

Lower Level of Difficulty

Observed Data

Model: Quadratic Class

3 Parameters (B0,B1,B2)

y = B0 + B1*x + B2*(x**2)

Certified Regression Statistics

Standard Deviation

Parameter Estimate of Estimate

B0 0.673565789473684E−03 0.107938612033077E−03

B1 0.732059160401003E−06 0.157817399981659E−09

B2 −0.316081871345029E−14 0.486652849992036E−16

Residual

Standard Deviation 0.205177424076185E−03

R−Squared 0.999999900178537

Certified Analysis of Variance Table

Source of Degrees of Sums of Mean

Variation Freedom Squares Squares F Statistic

Regression 2 15.6040343244198 7.80201716220991 185330865.995752

Residual 37 0.155761768796992E−05 0.420977753505385E−07

Data: y x

.11019 150000

.21956 300000

.32949 450000

.43899 600000

.54803 750000

.65694 900000

.76562 1050000

.87487 1200000

.98292 1350000

1.09146 1500000

1.20001 1650000

1.30822 1800000

1.41599 1950000

1.52399 2100000

1.63194 2250000

1.73947 2400000

1.84646 2550000

1.95392 2700000

2.06128 2850000

2.16844 3000000

.11052 150000

.22018 300000

.32939 450000

.43886 600000

.54798 750000

.65739 900000

.76596 1050000

.87474 1200000

.98300 1350000

1.09150 1500000

1.20004 1650000

1.30818 1800000

1.41613 1950000

1.52408 2100000

1.63159 2250000

1.73965 2400000

1.84696 2550000

1.95445 2700000

2.06177 2850000

2.16829 3000000

Y en matlab se importan igual los datos y (solo los numeros le llamo pointius2):

pointius2 =

1.0e+006 *

0.00000011019000 0.15000000000000

0.00000021956000 0.30000000000000

0.00000032949000 0.45000000000000

0.00000043899000 0.60000000000000

0.00000054803000 0.75000000000000

0.00000065694000 0.90000000000000

0.00000076562000 1.05000000000000

0.00000087487000 1.20000000000000

0.00000098292000 1.35000000000000

0.00000109146000 1.50000000000000

0.00000120001000 1.65000000000000

0.00000130822000 1.80000000000000

0.00000141599000 1.95000000000000

0.00000152399000 2.10000000000000

0.00000163194000 2.25000000000000

0.00000173947000 2.40000000000000

0.00000184646000 2.55000000000000

0.00000195392000 2.70000000000000

0.00000206128000 2.85000000000000

0.00000216844000 3.00000000000000

0.00000011052000 0.15000000000000

0.00000022018000 0.30000000000000

0.00000032939000 0.45000000000000

0.00000043886000 0.60000000000000

0.00000054798000 0.75000000000000

0.00000065739000 0.90000000000000

0.00000076596000 1.05000000000000

0.00000087474000 1.20000000000000

0.00000098300000 1.35000000000000

0.00000109150000 1.50000000000000

0.00000120004000 1.65000000000000

0.00000130818000 1.80000000000000

0.00000141613000 1.95000000000000

0.00000152408000 2.10000000000000

0.00000163159000 2.25000000000000

0.00000173965000 2.40000000000000

0.00000184696000 2.55000000000000

0.00000195445000 2.70000000000000

0.00000206177000 2.85000000000000

0.00000216829000 3.00000000000000

- Nota : si en el programa no tenemos la opción de import data , importaríamkos los datos de la siguiente manera: Pointius = load(`pointius.txt');

>> y = pointius2(:,1);

>> x = pointius2(:,2);

>> n = lenght(x)

>> n = length(x)

n =

40

>> A = ones(40,3);

```
>> A(:,2) = x;

>> A(:,3) = x.^2;

A =

1.0e+012 *

0.00000000000100 0.00000015000000 0.02250000000000

0.00000000000100 0.00000030000000 0.09000000000000

0.00000000000100 0.00000045000000 0.20250000000000

0.00000000000100 0.00000060000000 0.36000000000000

0.00000000000100 0.00000075000000 0.56250000000000

0.00000000000100 0.00000090000000 0.81000000000000

0.00000000000100 0.00000105000000 1.10250000000000

0.00000000000100 0.00000120000000 1.44000000000000

0.00000000000100 0.00000135000000 1.82250000000000

0.00000000000100 0.00000150000000 2.25000000000000

0.00000000000100 0.00000165000000 2.72250000000000

0.00000000000100 0.00000180000000 3.24000000000000

0.00000000000100 0.00000195000000 3.80250000000000

0.00000000000100 0.00000210000000 4.41000000000000

0.00000000000100 0.00000225000000 5.06250000000000

0.00000000000100 0.00000240000000 5.76000000000000

0.00000000000100 0.00000255000000 6.50250000000000

0.00000000000100 0.00000270000000 7.29000000000000

0.00000000000100 0.00000285000000 8.12250000000000

0.00000000000100 0.00000300000000 9.00000000000000

0.00000000000100 0.00000015000000 0.02250000000000

0.00000000000100 0.00000030000000 0.09000000000000
```

0.00000000000100 0.00000045000000 0.20250000000000

0.00000000000100 0.00000060000000 0.36000000000000

0.00000000000100 0.00000075000000 0.56250000000000

0.00000000000100 0.00000090000000 0.81000000000000

0.00000000000100 0.00000105000000 1.10250000000000

0.00000000000100 0.00000120000000 1.44000000000000

0.00000000000100 0.00000135000000 1.82250000000000

0.00000000000100 0.00000150000000 2.25000000000000

0.00000000000100 0.00000165000000 2.72250000000000

0.00000000000100 0.00000180000000 3.24000000000000

0.00000000000100 0.00000195000000 3.80250000000000

0.00000000000100 0.00000210000000 4.41000000000000

0.00000000000100 0.00000225000000 5.06250000000000

0.00000000000100 0.00000240000000 5.76000000000000

0.00000000000100 0.00000255000000 6.50250000000000

0.00000000000100 0.00000270000000 7.29000000000000

0.00000000000100 0.00000285000000 8.12250000000000

0.00000000000100 0.00000300000000 9.00000000000000

>> rank(A)

ans =

3

>> rank([A y])

ans =

3

>> beta = A\y

beta =

1.0e−003 *

0.67356578947336

0.00073205916040

−0.00000000000316

1) Ecuaciones normales AX = B ; A' A X = A' B

A' A X = M; MX = C

2) A = Q R

RX = Q' R X = R(Q*__); donde (Q*___) = D

3) A = U S V'

X = VS + U' B

>> M = A'*A

M =

1.0e+026 *

0.00000000000000 0.00000000000000 0.00000000000129

0.00000000000000 0.00000000000129 0.00000297675000

0.00000000000129 0.00000297675000 7.31699325000000

>> C = A' * y

C =

1.0e+014 *

0.00000000000046

0.00000093646979

2.15689932675000

>> beta1= M\C

Warning: Matrix is close to singular or badly scaled.

Results may be inaccurate. RCOND = 4.938239e−027.

beta1 =

```
1.0e−003 *

0.67356578947410

0.00073205916040

−0.00000000000316

>> [Q,R]=qr(A);

>> D = Q'*y;

>> beta2= R\D;

>> beta2

beta2 =

1.0e−003 *

0.67356578947465

0.00073205916040

−0.00000000000316

>> [U,S,V]= svd(A);

>> V

V =

0.00000000000018 0.00000129952456 0.99999999999916

0.00000040682694 0.99999999999907 −0.00000129952456

0.99999999999992 −0.00000040682694 0.00000000000035

>> S1 = pinv(S);

>> S1(1,2)

ans =

0

>> diag(S1)

ans =

0.00000000000004
```

0.00000035250209

0.52607450611596

>> beta3 = V*S1*U'*y;

>> beta3

beta3 =

1.0e−003 *

0.67356576935354

0.00073205916049

−0.00000000000316

>> norm(A*(beta2 − beta3))

ans =

2.128137835323598e−009

>> norm(A'*A*(beta2 − beta3))

ans =

5.735906807875822e+004

Cogemos el ejercicio 5.10 del moler (y es el que entregamos de práctica)

5.10. *Filip data set.* One of the Statistical Reference Datasets from the NIST is the \Filip" dataset. The data consists of several dozen observations of a variable *y* at di®erent values of *x*. The task is to model *y* by a polynomial of degree 10 in *x*.

This dataset is controversial. A search of the Web for \¯lip strd" will ¯nd several dozen postings, including the original page at NIST [3]. Some math− ematical and statistical packages are able to reproduce the polynomial coef− ¯cients that NIST has decreed to be the \certi¯ed values." Other packages give warning or error messages that the problem is too badly conditioned to solve. A few packages give di®erent coe±cients without warning. The Web

o®ers several opinions about whether or not this is a reasonable problem. Let's see what MATLAB does with it.

The data set is available from the NIST Web site. There is one line for each data point. The data is given with the ¯rst number on the line a value of *y*, and the second number the corresponding *x*. The *x*−values are not monotonically ordered, but it is not necessary to sort them. Let *n* be the number of data points and *p* = 11 the number of polynomial coe±cients.

(a) As your ¯rst experiment, load the data into MATLAB, plot it with '.' as the line type, and then invoke the Basic Fitting tool available under the Tools menu on the ¯gure window. Select the 10th degree polynomial ¯t. You will be warned that the polynomial is badly conditioned, but ignore that for now. How do the computed coe±cients compare with the certi¯ed values on the NIST Web page? How does the plotted ¯t compare with the graphic on the NIST Web page? The basic ¯tting tool also displays the norm of the residuals, *krk*. Compare this with the NIST quantity \Residual Standard Deviation," which is

$$krk\,pn ¡ p$$

(b) Examine this data set more carefully by using six di®erent methods to compute the polynomial ¯t. Explain all the warning messages you receive during these computations.

² Poly¯t: Use polyfit(x,y,10)

² Backslash: Use X*n*y where *X* is the *n*−by−*p* truncated Vandermonde ma− trix with elements

$$Xi;j = xp¡j$$

$$i\ ;\ i = 1; : : : ; n;\ j = 1; : : : ; p$$

² Pseudoinverse: Use pinv(X)*y

² Normal equations: Use inv(X'*X)*X'*y.

² Centering: Let ¹ = mean(*x*); ¾ = std(*x*); *t* = (*x* ¡ ¹)=¾.

Use polyfit(t,y,10).

² Certi¯ed coe±cients: Obtain the coe±cients from the NIST Web page.

(c) What are the norms of the residuals for the ¯ts computed by the six di®erent methods?

(d) Which one of the six methods gives a very poor ¯t? (Perhaps the packages that are criticized on theWeb for reporting bad results are using this method.)

(e) Plot the ¯ve good ¯ts. Use dots, '.', at the data values and curves obtained by evaluating the polynomials at a few hundred points over the range of the *x*'s. The plot should look like ¯gure 5.5. There are ¯ve di®erent plots, but only two visually distinct ones. Which methods produce which plots?

(f) Why do poly¯t and backslash give di®erent results?

"9 "8 "7 "6 "5 "4 "3

0.76

0.78

0.8

0.82

0.84

0.86

0.88

0.9

0.92

0.94

NIST Filip data set

Data

Rank 11

Rank 10

Figure 5.5. *NIST Filip standard reference data set*

TEMA 6 del mooler:

Ejercicio 6.4:

6.4. Use quadtx with various tolerances to compute pi by approximating

pi = integral desde −1 a 1 de : 2/(1+x^2) dx

solución : 2*arcotg entre −1 y 1 = 2 [pi/4 − (−pi/4)] = pi

How do the accuracy and the function evaluation count vary with tolerance?

Una manera de calcular pi es por cuadraturas, pero no es muy eficiente.

F(x) = 2/(1+x^2)

Fname.m

Inline

En matlab:

>> F = inline('2./(1+x^2)')

F =

Inline function:

F(x) = 2./(1+x^2)

>> G = inline('x.^y','x','y')

G =

Inline function:

G(x,y) = x.^y

Con el programa quadtx:

function [Q,fcount] = quadtx(F,a,b,tol,varargin)

```
%QUADTX Evaluate definite integral numerically.
% Q = QUADTX(F,A,B) approximates the integral of F(x) from A to B
% to within a tolerance of 1.e−6. F is a string defining a function
% of a single variable, an inline function, a function handle, or a
% symbolic expression involving a single variable.
%
% Q = QUADTX(F,A,B,tol) uses the given tolerance instead of 1.e−6.
%
% Arguments beyond the first four, Q = QUADTX(F,a,b,tol,p1,p2,...),
% are passed on to the integrand, F(x,p1,p2,..).
%
% [Q,fcount] = QUADTX(F,...) also counts the number of evaluations
% of F(x).
%
% See also QUAD, QUADL, DBLQUAD, QUADGUI.
% Make F callable by feval.
if ischar(F) & exist(F)~=2
F = inline(F);
elseif isa(F,'sym')
F = inline(char(F));
end
% Default tolerance
if nargin < 4 | isempty(tol)
tol = 1.e−6;
end
% Initialization
```

```
c = (a + b)/2;

fa = feval(F,a,varargin{:});

fc = feval(F,c,varargin{:});

fb = feval(F,b,varargin{:});

% Recursive call

[Q,k] = quadtxstep(F, a, b, tol, fa, fc, fb, varargin{:});

fcount = k + 3;

% −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

function [Q,fcount] = quadtxstep(F,a,b,tol,fa,fc,fb,varargin)

% Recursive subfunction used by quadtx.

h = b − a;

c = (a + b)/2;

fd = feval(F,(a+c)/2,varargin{:});

fe = feval(F,(c+b)/2,varargin{:});

Q1 = h/6 * (fa + 4*fc + fb);

Q2 = h/12 * (fa + 4*fd + 2*fc + 4*fe + fb);

if abs(Q2 − Q1) <= tol

Q = Q2 + (Q2 − Q1)/15;

fcount = 2;

else

[Qa,ka] = quadtxstep(F, a, c, tol, fa, fd, fc, varargin{:});

[Qb,kb] = quadtxstep(F, c, b, tol, fc, fe, fb, varargin{:});

Q = Qa + Qb;

fcount = ka + kb + 2;

end

>> I = quadtx(F,−1,1,10^−6)
```

I =

3.14159265370804

I2 = cuadcompNC(F,−1,1,3,10)

I2=

3.14159261393922

I2= cuadcmpNC(F,−1,1,8,10)

I2 =

3.14159265358986

I2= cuadcompNC(F,−1,1,8,100)

I2 =

3.14159265358979

Otro ejercicio es el 6.6 (página14 del capitulo 6 del mooler)

6.6. The error function, erf(x), is de¯ned by an integral,

erf(x) = 2/raiz(pi) * integral de 0 a x de e elevado a −x^2

Use quadtx to tabulate erf(x) for x = 0:1*; 0:2; : : : ;* 1:0*:* Compare the results

with the built−in Matlab function erf(x).

**Nota para el 6.8 . Tenemos una gamma(n+1) = n!*

30−05−06

Capitulo 8 (chapter 7)

*Metodo de Picard

Tenemos el archivo:

Harmonic:

function ydot = harmonic(t,y)

ydot = [y(2); −y(1)];

en matlab:

>> [t,y] = ode23('harmonic',[0 5],[3;0]);

```
>> [t,y] = ode23('harmonic',[0 5],[3;0])

t =

0

0.0000

0.0002

0.0008

0.0042

0.0208

0.1008

0.2361

0.4164

0.6367

0.8973

1.1880

1.4396

1.5862

1.7328

1.8913

2.0918

2.3317

2.6147

2.8857

3.0680

3.2062

3.3445

3.5157
```

3.7276

3.9793

4.2768

4.5361

4.6975

4.8589

5.0000

y =

3.0000 0

3.0000 −0.0001

3.0000 −0.0005

3.0000 −0.0025

3.0000 −0.0125

2.9993 −0.0625

2.9848 −0.3019

2.9167 −0.7019

2.7435 −1.2134

2.4118 −1.7834

1.8704 −2.3443

1.1194 −2.7813

0.3917 −2.9719

−0.0467 −2.9972

−0.4841 −2.9581

−0.9448 −2.8446

−1.4923 −2.5993

−2.0671 −2.1698

−2.5902 −1.5058

−2.8981 −0.7573

−2.9872 −0.2194

−2.9889 0.1945

−2.9335 0.6046

−2.7875 1.0955

−2.4945 1.6571

−2.0029 2.2258

−1.2619 2.7144

−0.5234 2.9467

−0.0431 2.9924

0.4383 2.9604

0.8503 2.8692

>> plot(t,y(:,1))

> [t,y] = ode23('harmonic',[0 50],[3;0]);

>> plot(t,y(:,1))

Tenemos el programa :

Harmonic_damped al que llamamos por comodidad : harmonicd :

function ydot = harmonic_damped(t,y)

alpha = .5;

ydot = [y(2); −y(1) − alpha *y(2)];

en matlab:

>> [t,y] = ode23('harmonicd',[0 50],[3;0]);

>> plot(t,y(:,1))

Cambiamos el alfa del programa:

function ydot = harmonic_damped(t,y)

```
alpha = .1;

ydot = [y(2); −y(1) − alpha *y(2)];

>> [t,y] = ode23('harmonicd',[0 50],[3;0]);

>> plot(t,y(:,1))
```

Cambiamos el programa:

```
function ydot = harmonic_damped(t,y)

alpha = .1; beta = 1.2;

ydot = [y(2); −y(1) − alpha *(y(2).^beta)];
```

en matlab:

```
>> [t,y] = ode23('harmonicd',[0 50],[3;0]);

>> plot(t,y(:,1))

Warning: Imaginary parts of complex X and/or Y arguments ignored.

>> plot(y(:,1),y(:,2))

Warning: Imaginary parts of complex X and/or Y arguments ignored.

>> [t,y] = ode23('harmonicd',[0 50],[7;2]);

>> [t,y] = ode23('harmonicd',[0 100],[7;2]);

>> plot(y(:,1),y(:,2))

Warning: Imaginary parts of complex X and/or Y arguments ignored.
```

Tenemos el programa:

Twobody:

```
function ydot = twobody(t,y)

r = sqrt(y(1)^2 + y(2)^2);

ydot = [y(3); y(4); −y(1)/r^3 ; −y(2)/r^3];

>> [t,y] = ode23('twobody',[0 3],[5;0;0;0.1]);

>> plot(y(:,1),y(:,2))

>> [t,y] = ode23('twobody',[0 10],[10;0;0;0.1]);
```

```
>> plot(y(:,1),y(:,2))
>> [t,y] = ode23('twobody',[0 10],[3;0;0;0.1]);
>> plot(y(:,1),y(:,2))
>>
>> [t,y] = ode23('twobody',[0 20],[3;0;0;0.1]);
>> plot(y(:,1),y(:,2))
>> [t,y] = ode45('twobody',[0 20],[3;0;0;0.1]);
>> plot(y(:,1),y(:,2))
>> [t,y] = ode45('twobody',[0 200],[3;0;0;0.1]);
>> plot(y(:,1),y(:,2))
```

1−junio−06

Ejercicio del libro moler (26 Chapter 5. Least Squares):

5.12. Planetary orbit [2]. The expression $z = ax2+bxy+cy2+dx+ey+f$ is known as a *quadratic form*. The set of points $(x; y)$ where $z = 0$ is a *conic section*. It can be an ellipse, a parabola, or a hyperbola, depending on the sign of the discriminant $b2 ¡ 4ac$. Circles and lines are special cases. The equation $z = 0$ can be normalized by dividing the quadratic form by any nonzero coe±cient. For example, if $f 6= 0$, we can divide all the other coe±cients by f and obtain a quadratic form with the constant term equal to one. You can use the Matlab meshgrid and contour functions to plot conic sections. Use meshgrid to create arrays X and Y. Evaluate the quadratic form to produce Z. Then use contour to plot the set of points where Z is zero.

```
[X,Y] = meshgrid(xmin:deltax:xmax,ymin:deltay:ymax);
Z = a*X.^2 + b*X.*Y + c*Y.^2 + d*X + e*Y + f;
contour(X,Y,Z,[0 0])
```

A planet follows an elliptical orbit. Here are ten observations of its position

in the (*x; y*) plane:

x = [1.02 .95 .87 .77 .67 .56 .44 .30 .16 .01]';

y = [0.39 .32 .27 .22 .18 .15 .13 .12 .13 .15]';

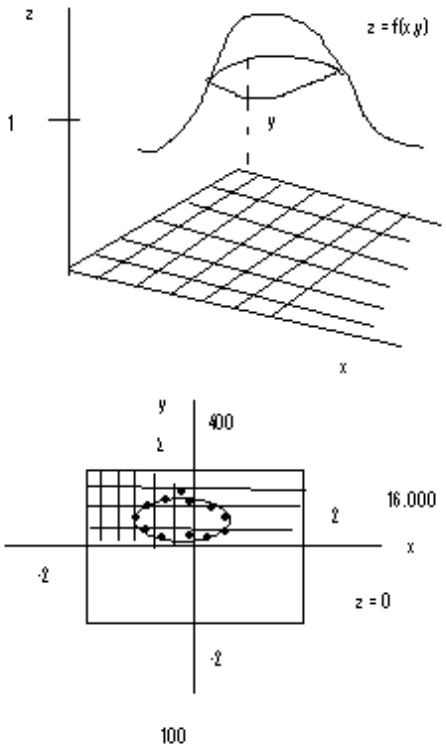
(a) Determine the coe±cients in the quadratic form that ¯ts this data in the least squares sense by setting one of the coe±cients equal to one and solving a 10−by−5 overdetermined system of linear equations for the other ¯ve coe±cients. Plot the orbit with *x* on the *x*−axis and *y* on the *y*−axis. Superimpose the ten data points on the plot.

(b) This least squares problem is nearly rank de¯cient. To see what e®ect this has on the solution, perturb the data slightly by adding to each coordinate of each data point a random number uniformly distributed in the interval [−.005,.005]. Compute the new coe±cients resulting from the perturbed data. Plot the new orbit on the same plot with the old orbit. Comment on your comparison of the sets of coe±cients and the orbits.

En matlab:

Practica3

```
[x,y]=meshgrid(−2:0.01:2,−2:0.01:2);
a=1;b=1;c=1;e=1;f=−1;
>> z = a*x.^2+b*x.*y+c*y.^2+b*x+e*y+f;
>> contour(x,y,z,[0 0])
>> contour(x,y,z,5)
>> contour(x,y,z,10)
```



Para cada punto de x e y tenemos una ecuación diferente de:

= ax^2 + bxy +cy^2+dx+ey

C= (a b c d e)' ; A*C = (1 . 1)' = r

X`2= (x1^2 . X10^2)'

A=[X`2 XY Y^2 X Y]

C = A/r

En matlab:

>> x = [1.02 .95 .87 .77 .67 .56 .44 .30 .16 .01]';

>> y = [0.39 .32 .27 .22 .18 .15 .13 .12 .13 .15]';

>> x

x =

1.0200

0.9500

0.8700

0.7700

0.6700

0.5600

0.4400

0.3000

0.1600

0.0100

>> y

y =

0.3900

0.3200

0.2700

0.2200

0.1800

0.1500

0.1300

0.1200

0.1300

0.1500

>> A = [x.^2 x.*y y.^2 x y];

>> size(A)

ans =

10 5

>> r= ones(10,1)

r =

1

```
1

1

1

1

1

1

1

1

1

>> [X,Y]= meshgrid(−2:0.01:2,−2:0.01:2);

>> z = a*X.^2+b*X.*Y+c*Y.^2+b*X+e*Y+f;

>> A = [x.^2 x.*y y.^2 x y];

>> C=A\r;

>> a = C(1);b=C(2);c= C(3);d=C(4);e=C(5);

>> Z = a*X.^2+b*X.*Y+c*Y.^2+b*X+e*Y+f;

>> [X,Y]= meshgrid(−2:0.01:2,−2:0.01:2);

>> x = [1.02 .95 .87 .77 .67 .56 .44 .30 .16 .01]';

>> y = [0.39 .32 .27 .22 .18 .15 .13 .12 .13 .15]';

>> Z = a*X.^2+b*X.*Y+c*Y.^2+b*X+e*Y+f;

>> C=A\r;

>> a = C(1);b=C(2);c= C(3);d=C(4);e=C(5);

>> contour(X,Y,Z,[0 0])

>> plot(x,y,'r*')

>> x = x + 0.005*(rand(10,1)−0.5);

>> y = y + 0.005*(rand(10,1)−0.5);

>> A = [x.^2 x.*y y.^2 x y];
```

```
>> C = A\r;

>> Z = a*X.^2+b*X.*Y+c*Y.^2+b*X+e*Y+f;

>> a = C(1); b = C(2); c = C(3); d = C(4); e = C(5);

>> Z = a*X.^2+b*X.*Y+c*Y.^2+b*X+e*Y+f;

>> contour(X,Y,Z,[0 0])

>> hold on
```

PROGRAMAS USADOS:

BSLASHTX

```
function x = bslashtx(A,b)

% BSLASHTX Solve linear system (backslash)

% x = bslashtx(A,b) solves A*x = b

[n,n] = size(A);

if isequal(triu(A,1),zeros(n,n))

% Lower triangular

x = forward(A,b);

return

elseif isequal(tril(A,−1),zeros(n,n))

% Upper triangular

x = backsubs(A,b);

return

elseif isequal(A,A')

[R,fail] = chol(A);

if ~fail

% Positive definite

y = forward(R',b);

x = backsubs(R,y);
```

```
return

end

end

% Triangular factorization

[L,U,p] = lutx(A);

% Permutation and forward elimination

y = forward(L,b(p));

% Back substitution

x = backsubs(U,y);

% −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

function x = forward(L,x)

% FORWARD. Forward elimination.

% For lower triangular L, x = forward(L,b) solves L*x = b.

[n,n] = size(L);

x(1) = x(1)/L(1,1);

for k = 2:n

j = 1:k−1;

x(k) = (x(k) − L(k,j)*x(j))/L(k,k);

end

% −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

function x = backsubs(U,x)

% BACKSUBS. Back substitution.

% For upper triangular U, x = backsubs(U,b) solves U*x = b.

[n,n] = size(U);

x(n) = x(n)/U(n,n);

for k = n−1:−1:1
```

```
j = k+1:n;

x(k) = (x(k) − U(k,j)*x(j))/U(k,k);

end
```

CGS

```
% Gram−Schmidt orthogonalization (unstable)

function [Q,R] = cgs(A)

n = length(A);

% starting variables

R = zeros(n,n);

Q = zeros(n,n);

V = zeros(n,n);

% Normalizing the first column vector A

R(1,1)=norm(A(:,1));

Q(:,1)= A(:,1)/R(1,1);

% Classical Gram−Schmidt

for j = 2:n

V(:,j) = A(:,j);

for i = 1:j−1

R(i,j) = Q(:,i)'*A(:,j);

V(:,j)= V(:,j)−R(i,j)*Q(:,i);

end

R(j,j)=norm(V(:,j));

Q(:,j)=V(:,j)/R(j,j);

end

Q;

R;
```

CUADCOMNC

```
% Este fichero calcula la cuadratura
% de Newton−Cotes de la funcion fname
% con m puntos y extremos del intervalo a y b.
% a,b numeros reales
% m entero entre 2 <= m <=11
% El computo devuelve el numero I
%
function I = cuadraturaNC(fname,a,b,m)
p = pesosNC(m);
%x = linspace(a,b,m)';
h = (b−a)/(m−1);
x = (a:h:b)';
f = feval(fname,x);
I = (b−a)*(p'*f);
```

CUADRATURA

```
% Este fichero calcula la cuadratura
% de Newton−Cotes de la funcion fname
% con m puntos y extremos del intervalo a y b.
% a,b numeros reales
% m entero entre 2 <= m <=11
% El computo devuelve el numero I
%
function I = cuadraturaNC(fname,a,b,m)
p = pesosNC(m);
%x = linspace(a,b,m)';
```

```
h = (b−a)/(m−1);

x = (a:h:b)';

f = feval(fname,x);

I = (b−a)*(p'*f);
```

CUADRATURANC

```
% Este fichero calculo la cuadratura

% de Newton−Cotes compuesta con m nodos

% y n subintervalos

% a,b numeros reales, a < b

% m numero de puntos, 2 <= m <=11

% n numero de subintervalos

%

% El programa calcula

% I La aproximacion de Newton−Cotes compuesta

% de la integral de f(x) entre a y b.

% La regla es aplicada a cada uno de los

% n subintervalos iguales de [a,b].

%

function I = cuadcompNC(fname,a,b,m,n)

Delta = (b−a)/n;

h = Delta/(m−1);

x = a+h*(0:(n*(m−1)))';

p = pesosNC(m);

%x = linspace(a,b,n*(m−1)+1)';

f = feval(fname,x);

I = 0;
```

```
first = 1;

last = m;

for i=1:n

%A−adimos al producto interno el valor en

% el subintervalo i−esimo.

I = I + p'*f(first: last);

first = last;

last = last+m−1;

end

I = Delta*I;
```

DEMO_CGS_MGS

```
% Numerical comparison:

% Classical Gram−Schmidt and

% modified Gram−Schmidt

% First we generate a random matrix A

% with eigenvalues spaced by a factor 2

% between 2^−1 and 2^−80

[U,X]= qr(randn(80));

S = diag(2.^(−1:−1:−80));

A = U*S*U';

E = eig(A);

% We use now classical Gram−Schmid cgs(A)

% and modified Gram−Schmidt mgs(A) to compute Q and R

[Q1,R1]=cgs(A);

[Q2,R2]=mgs(A);

% We represent the diagonal R elements
```

```
% in logarithmic scale for both matrices R1 and R2.

x = 1:80;

y1 = log2(diag(R1));

y2 = log2(diag(R2));

y6 = log2(−sort(−abs(E)));

y3 = −23*ones(1,80);

y4 = −55*ones(1,80);

y5 = −x;

plot(x,y1,'bo',x,y2,'rx',x,y6,'g*',x,y3,':',x,y4,':',x,y5,'−');

xlabel('R(i,i) elements in logarithmic scale');
```

DEMO_INTER

```
%

% This script shows the graphic of cos(4x)

% by interpolating with a polynomial of degree

% 12 found by least squares.

close all

t = (linspace(0,5,50))';

A = vander(t);

A = A(:,38:50);

b = (cos(4*t));

u = 0:0.01:5;

v = cos(4*u);

p = A\b;

pvals = horner(p,t);

plot(t,b,'or',t,pvals,'b',u,v,'g')
```

DEMO_ORTOG

```
% This script explores the lost of orthogonality
% due to the intrinsic numerical unstability
% of the classical and
% modified Gram−Schmidt algorithms.
clc
disp(' Losing orthogonality on Gram−Schmidt algorithms ')
disp(' delta Error on QR Error on CGS Error on MGS ')
disp('−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−')
for delta = logspace(−4,−16,7)
A = [1+delta 2;1 2];
[Q1,R1]= qr(A);
[Q2,R2]= cgs(A);
[Q3,R3]= mgs(A);
error1 = norm(Q1'*Q1 − eye(2));
error2 = norm(Q2'*Q2 − eye(2));
error3 = norm(Q3'*Q3 − eye(2));
disp(sprintf(' %5.0e %20.15f %20.15f %20.15f ', delta,error1,error2,error3))
end
```

EULER

```
function y = euler(f,t0,y0,h,tf)
t = t0;
y = y0;
while t <= tf
y = y + h*feval(f,t,y);
t = t + h;
end
```

FZEROTX

```
function b = fzerotx(F,ab,varargin)
%FZEROTX Textbook version of FZERO.
% x = fzerotx(F,[a,b]) tries to find a zero of F(x) between a and b.
% F(a) and F(b) must have opposite signs. fzerotx returns one
% end point of a small subinterval of [a,b] where F changes sign.
% Arguments beyond the first two, fzerotx(F,[a,b],p1,p2,...),
% are passed on, F(x,p1,p2,..).
%
% Example:
% fzerotx('sin(x)',[1,4])
% Make F callable by feval.
if ischar(F) & exist(F)~=2
F = inline(F);
elseif isa(F,'sym')
F = inline(char(F));
end
% Initialize.
a = ab(1);
b = ab(2);
fa = feval(F,a,varargin{:});
fb = feval(F,b,varargin{:});
if sign(fa) == sign(fb)
error('Function must change sign on the interval')
end
c = a;
```

```
fc = fa;

d = b − c;

e = d;

% Main loop, exit from middle of the loop

while fb ~= 0

% The three current points, a, b, and c, satisfy:

% f(x) changes sign between a and b.

% abs(f(b)) <= abs(f(a)).

% c = previous b, so c might = a.

% The next point is chosen from

% Bisection point, (a+b)/2.

% Secant point determined by b and c.

% Inverse quadratic interpolation point determined

% by a, b, and c if they are distinct.

if sign(fa) == sign(fb)

a = c; fa = fc;

d = b − c; e = d;

end

if abs(fa) < abs(fb)

c = b; b = a; a = c;

fc = fb; fb = fa; fa = fc;

end

% Convergence test and possible exit

m = 0.5*(a − b);

tol = 2.0*eps*max(abs(b),1.0);

if (abs(m) <= tol) | (fb == 0.0)
```

```
break

end

% Choose bisection or interpolation

if (abs(e) < tol) | (abs(fc) <= abs(fb))

% Bisection

d = m;

e = m;

else

% Interpolation

s = fb/fc;

if (a == c)

% Linear interpolation (secant)

p = 2.0*m*s;

q = 1.0 − s;

else

% Inverse quadratic interpolation

q = fc/fa;

r = fb/fa;

p = s*(2.0*m*q*(q − r) − (b − c)*(r − 1.0));

q = (q − 1.0)*(r − 1.0)*(s − 1.0);

end;

if p > 0, q = −q; else p = −p; end;

% Is interpolated point acceptable

if (2.0*p < 3.0*m*q − abs(tol*q)) & (p < abs(0.5*e*q))

e = d;

d = p/q;
```

```
else

d = m;

e = m;

end;

end

% Next point

c = b;

fc = fb;

if abs(d) > tol

b = b + d;

else

b = b − sign(b−a)*tol;

end

fb = feval(F,b,varargin{:});

end
```

HARMONIC

```
function ydot = harmonic(t,y)

ydot = [y(2); −y(1)];
```

HARMONIC DAM PED

```
function ydot = harmonic_damped(t,y)

alpha = .5;

ydot = [y(2); −y(1) − alpha *y(2)];
```

HORNER

```
% This function evaluates

% the polynomial of degree n−1 with coefficients a1,...,an

% at the point z by using Horner's algorithm
```

```
function p = horner(a,z)

n = length(a);

m = length(z);

p = a(1)*ones(m,1);

for k = 2:n

p = z.*p + a(k);

end
```

LUTX

```
function [L,U,p] = lutx(A)

%LUTX Triangular factorization, textbook version

% [L,U,p] = lutx(A) produces a unit lower triangular matrix L,

% an upper triangular matrix U, and a permutation vector p,

% so that L*U = A(p,:)

[n,n] = size(A);

p = (1:n)';

for k = 1:n−1

% Find index of largest element below diagonal in k−th column

[r,m] = max(abs(A(k:n,k)));

m = m+k−1;

% Skip elimination if column is zero

if (A(m,k) ~= 0)

% Swap pivot row

if (m ~= k)

A([k m],:) = A([m k],:);

p([k m]) = p([m k]);

end
```

```
% Compute multipliers
i = k+1:n;
A(i,k) = A(i,k)/A(k,k);
% Update the remainder of the matrix
j = k+1:n;
A(i,j) = A(i,j) − A(i,k)*A(k,j);
end
end
% Separate result
L = tril(A,−1) + eye(n,n);
U = triu(A);
```

MGS

```
% Modified Gram−Schmidt algorithm (stable)
function [Q,R] = mgs(A)
[m,n] = size(A);
% Initializating variables
R = zeros(m,n);
S = zeros(m,m);
Q = zeros(m,m);
V = [A floor(10*rand(m,m−n))];
for i = 1:n
R(i,i)=norm(V(:,i));
Q(:,i)=V(:,i)/R(i,i);
for j = i+1:n
R(i,j) = Q(:,i)'*V(:,j);
V(:,j)= V(:,j)−R(i,j)*Q(:,i);
```

```
end

end

for i = n+1:m

S(i,i)=norm(V(:,i));

Q(:,i)=V(:,i)/S(i,i);

for j = i+1:n

S(i,j) = Q(:,i)'*V(:,j);

V(:,j)= V(:,j)−S(i,j)*Q(:,i);

end

end

Q;

R;
```

ODE23TX

```
function [tout,yout] = ode23tx(F,tspan,y0,arg4,varargin)

%ODE23TX Solve non−stiff differential equations. Textbook version of ODE23.

%

% ODE23TX(F,TSPAN,Y0) with TSPAN = [T0 TFINAL] integrates the system

% of differential equations y' = f(t,y) from t = T0 to t = TFINAL. The

% initial condition is y(T0) = Y0. The input argument F is the

% name of an M−file, or an inline function, or simply a character string,

% defining f(t,y). This function must have two input arguments, t and y,

% and must return a column vector of the derivatives, y'.

%

% With two output arguments, [T,Y] = ODE23TX(...) returns a column

% vector T and an array Y where Y(:,k) is the solution at T(k).

%
```

```
% With no output arguments, ODE23TX plots the emerging solution.
%
% ODE23TX(F,TSPAN,Y0,RTOL) uses the relative error tolerance RTOL
% instead of the default 1.e−3.
%
% ODE23TX(F,TSPAN,Y0,OPTS) where OPTS = ODESET('reltol',RTOL, ...
% 'abstol',ATOL,'outputfcn',@PLOTFUN) uses relative error RTOL instead
% of 1.e−3, absolute error ATOL instead of 1.e−6, and calls PLOTFUN
% instead of ODEPLOT after each successful step.
%
% More than four input arguments, ODE23TX(F,TSPAN,Y0,RTOL,P1,P2,...),
% are passed on to F, F(T,Y,P1,P2,...).
%
% ODE23TX uses the Runge−Kutta (2,3) method of Bogacki and Shampine (BS23).
%
% Example
% tspan = [0 2*pi];
% y0 = [1 0]';
% F = '[0 1; −1 0]*y';
% ode23tx(F,tspan,y0);
%
% See also ODE23.
% Initialize variables.
rtol = 1.e−3;
atol = 1.e−6;
plotfun = @odeplot;
```

```
if nargin >= 4 & isnumeric(arg4)
rtol = arg4;
elseif nargin >= 4 & isstruct(arg4)
if ~isempty(arg4.RelTol), rtol = arg4.RelTol; end
if ~isempty(arg4.AbsTol), atol = arg4.AbsTol; end
if ~isempty(arg4.OutputFcn), plotfun = arg4.OutputFcn; end
end
t0 = tspan(1);
tfinal = tspan(2);
tdir = sign(tfinal − t0);
plotit = (nargout == 0);
threshold = atol / rtol;
hmax = abs(0.1*(tfinal−t0));
t = t0;
y = y0(:);
% Make F callable by feval.
if ischar(F) & exist(F)~=2
F = inline(F,'t','y');
elseif isa(F,'sym')
F = inline(char(F),'t','y');
end
% Initialize output.
if plotit
feval(plotfun,tspan,y,'init');
else
tout = t;
```

```
yout = y.';
end
% Compute initial step size.
s1 = feval(F, t, y, varargin{:});
r = norm(s1./max(abs(y),threshold),inf) + realmin;
h = tdir*0.8*rtol^(1/3)/r;
% The main loop.
while t ~= tfinal
hmin = 16*eps*abs(t);
if abs(h) > hmax, h = tdir*hmax; end
if abs(h) < hmin, h = tdir*hmin; end
% Stretch the step if t is close to tfinal.
if 1.1*abs(h) >= abs(tfinal − t)
h = tfinal − t;
end
% Attempt a step.
s2 = feval(F, t+h/2, y+h/2*s1, varargin{:});
s3 = feval(F, t+3*h/4, y+3*h/4*s2, varargin{:});
tnew = t + h;
ynew = y + h*(2*s1 + 3*s2 + 4*s3)/9;
s4 = feval(F, tnew, ynew, varargin{:});
% Estimate the error.
e = h*(−5*s1 + 6*s2 + 8*s3 − 9*s4)/72;
err = norm(e./max(max(abs(y),abs(ynew)),threshold),inf) + realmin;
% Accept the solution if the estimated error is less than the tolerance.
if err <= rtol
```

```
t = tnew;
y = ynew;
if plotit
if feval(plotfun,t,y,'');
break
end
else
tout(end+1,1) = t;
yout(end+1,:) = y.';
end
s1 = s4; % Reuse final function value to start new step.
end
% Compute a new step size.
h = h*min(5,0.8*(rtol/err)^(1/3));
% Exit early if step size is too small.
if abs(h) <= hmin
warning('Step size %e too small at t = %e.\n',h,t);
t = tfinal;
end
end
if plotit
feval(plotfun,[],[],'done');
end
```

ORBIT EULER

```
function orbit_euler(F,tspan,h,y0)
clc
```

```
%t0 = 0;

%h = 0.001;

%y0 = [1;0];

t0 = tspan(1);

tf = tspan(2);

t1 = t0 + h;

u = t0:h:tf;

v = u;

k = 1;

hs = h/10;

while t1 <= tf

y = euler(F,t0,y0,hs,t1);

u(k) = y(1);

v(k) = y(2);

plot(u(k),v(k),'b')

hold on

t0 = t1;

t1 = t0 + h;

y0 = y;

k = k+1;

end
```

PCHIPSLOPES

```
function d = pchipslopes(h,delta)

% PCHIPSLOPES Slopes for shape−preserving Hermite cubic

% interpolation. pchipslopes(h,delta) computes d(k) = P'(x(k)).

% Slopes at interior points
```

```
% delta = diff(y)./diff(x).
% d(k) = 0 if delta(k−1) and delta(k) have opposites signs
% or either is zero.
% d(k) = weighted harmonic mean of delta(k−1) and delta(k)
% if they have the same sign.
n = length(h)+1;
d = zeros(size(h));
k = find(sign(delta(1:n−2)).*sign(delta(2:n−1)) > 0) + 1;
w1 = 2*h(k)+h(k−1);
w2 = h(k)+2*h(k−1);
d(k) = (w1+w2)./(w1./delta(k−1) + w2./delta(k));
% Slopes at endpoints
d(1) = pchipendpoint(h(1),h(2),delta(1),delta(2));
d(n) = pchipendpoint(h(n−1),h(n−2),delta(n−1),delta(n−2));
% −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
```

PCHIPTX

```
function v = pchiptx(x,y,u)
%PCHIPTX Textbook piecewise cubic Hermite interpolation.
% v = pchiptx(x,y,u) finds the shape−preserving piecewise cubic
% interpolant P(x), with P(x(j)) = y(j), and returns v(k) = P(u(k)).
%
% See PCHIP, SPLINETX.
% First derivatives
h = diff(x);
delta = diff(y)./h;
d = pchipslopes(h,delta);
```

```
% Piecewise polynomial coefficients

n = length(x);

c = (3*delta − 2*d(1:n−1) − d(2:n))./h;

b = (d(1:n−1) − 2*delta + d(2:n))./h.^2;

% Find subinterval indices k so that x(k) <= u < x(k+1)

k = ones(size(u));

for j = 2:n−1

k(x(j) <= u) = j;

end

% Evaluate interpolant

s = u − x(k);

v = y(k) + s.*(d(k) + s.*(c(k) + s.*b(k)));

% −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

function d = pchipslopes(h,delta)

% PCHIPSLOPES Slopes for shape−preserving Hermite cubic

% interpolation. pchipslopes(h,delta) computes d(k) = P'(x(k)).

% Slopes at interior points

% delta = diff(y)./diff(x).

% d(k) = 0 if delta(k−1) and delta(k) have opposites signs

% or either is zero.

% d(k) = weighted harmonic mean of delta(k−1) and delta(k)

% if they have the same sign.

n = length(h)+1;

d = zeros(size(h));

k = find(sign(delta(1:n−2)).*sign(delta(2:n−1)) > 0) + 1;

w1 = 2*h(k)+h(k−1);
```

```
w2 = h(k)+2*h(k−1);

d(k) = (w1+w2)./(w1./delta(k−1) + w2./delta(k));

% Slopes at endpoints

d(1) = pchipendpoint(h(1),h(2),delta(1),delta(2));

d(n) = pchipendpoint(h(n−1),h(n−2),delta(n−1),delta(n−2));

% −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

function d = pchipendpoint(h1,h2,del1,del2)

% Noncentered, shape−preserving, three−point formula.

d = ((2*h1+h2)*del1 − h1*del2)/(h1+h2);

if sign(d) ~= sign(del1)

d = 0;

elseif (sign(del1) ~= sign(del2)) & (abs(d) > abs(3*del1))

d = 3*del1;

end
```

PESOSNC

```
% Este fichero almacena los

% pesos de las reglas de

% cuadratura de Newton−Cotes

% con m puntos

%

% m es un entero que satisface 2 <= m <= 11

%

% p es un vector columna que contiene

% los pesos para la regla de cuadratura

% de Newton−Cotes con m puntos.

%
```

```
function p = pesosNC(m);
if m==2
p=[1 1]'/2;
elseif m==3
p=[1 4 1]'/6;
elseif m==4
p=[1 3 3 1]'/8;
elseif m==5
p=[7 32 12 32 7]'/90;
elseif m==6
p=[19 75 50 50 75 19]'/288;
elseif m==7
p=[41 216 27 272 27 216 41]'/840;
elseif m==8
p=[751 3577 1323 2989 2989 1323 3577 751]'/17280;
elseif m==9
p=[989 5888 −928 10496 −4540 10496 −928 5888 989]'/28350;
elseif m==10
p=[2857 15741 1080 19344 5778 5778 19344 1080 15741 2857]'/89600;
else
p=[16067 106300 −48525 272400 −260550 427368 −260550 272400 −48525 106300 16067]'/598752;
end;
```

PIECELIN

```
function v = piecelin(x,y,u)
%PIECELIN Piecewise linear interpolation.
% v = piecelin(x,y,u) finds the piecewise linear L(x)
```

```
% with L(x(j)) = y(j) and returns v(k) = L(u(k)).

% First divided difference

delta = diff(y)./diff(x);

% Find subinterval indices k so that x(k) <= u < x(k+1)

n = length(x);

k = ones(size(u));

for j = 2:n−1

k(x(j) <= u) = j;

end

% Evaluate interpolant

s = u − x(k);

v = y(k) + s.*delta(k);
```

POLYINTERP

```
function v = polyinterp(x,y,u)

%POLYINTERP Polynomial interpolation.

% v = POLYINTERP(x,y,u) computes v(j) = P(u(j)) where P is the

% polynomial of degree d = length(x)−1 with P(x(i)) = y(i).

% Use Lagrangian representation.

% Evaluate at all elements of u simultaneously.

n = length(x);

v = zeros(size(u));

for k = 1:n

w = ones(size(u));

for j = [1:k−1 k+1:n]

w = (u−x(j))./(x(k)−x(j)).*w;

end
```

```
v = v + w*y(k);

end
```

QRSTEPS

```
function [A,b] = qrsteps(A,b)

%QRSTEPS Orthogonal−triangular decomposition.

% Demonstrates M−file version of built−in QR function.

% R = QRSTEPS(A) is the upper trapezoidal matrix R that

% results from the orthogonal transformation, R = Q'*A.

% With no output argument, QRSTEPS(A) shows the steps in

% the computation of R. Press <enter> key after each step.

% [R,bout] = QRSTEPS(A,b) also applies the transformation to b.

[m,n] = size(A);

if nargin < 2, b = zeros(m,0); end

% Householder reduction to triangular form.

if nargout == 0

clc

A

if ~isempty(b), b, end

pause

end

for k = 1:min(m−1,n)

% Introduce zeros below the diagonal in the k−th column.

% Use Householder transformation, I − rho*u*u'.

i = k:m;

u = A(i,k);

sigma = norm(u);
```

```
% Skip transformation if column is already zero.
if sigma ~= 0
if u(1) ~= 0, sigma = sign(u(1))*sigma; end
u(1) = u(1) + sigma;
rho = 1/(conj(sigma)*u(1));
% Update the k−th column.
A(i,k) = 0;
A(k,k) = −sigma;
% Apply the transformation to remaining columns of A.
j = k+1:n;
v = rho*(u'*A(i,j));
A(i,j) = A(i,j) − u*v;
% Apply the transformation to b.
if ~isempty(b)
tau = rho*(u'*b(i));
b(i) = b(i) − tau*u;
end
end
if nargout == 0
clc
A
if ~isempty(b), b, end
pause
end
end
```

QUADTX

```
function [Q,fcount] = quadtx(F,a,b,tol,varargin)
%QUADTX Evaluate definite integral numerically.
% Q = QUADTX(F,A,B) approximates the integral of F(x) from A to B
% to within a tolerance of 1.e−6. F is a string defining a function
% of a single variable, an inline function, a function handle, or a
% symbolic expression involving a single variable.
%
% Q = QUADTX(F,A,B,tol) uses the given tolerance instead of 1.e−6.
%
% Arguments beyond the first four, Q = QUADTX(F,a,b,tol,p1,p2,...),
% are passed on to the integrand, F(x,p1,p2,..).
%
% [Q,fcount] = QUADTX(F,...) also counts the number of evaluations
% of F(x).
%
% See also QUAD, QUADL, DBLQUAD, QUADGUI.
% Make F callable by feval.
if ischar(F) & exist(F)~=2
F = inline(F);
elseif isa(F,'sym')
F = inline(char(F));
end
% Default tolerance
if nargin < 4 | isempty(tol)
tol = 1.e−6;
end
```

```
% Initialization

c = (a + b)/2;

fa = feval(F,a,varargin{:});

fc = feval(F,c,varargin{:});

fb = feval(F,b,varargin{:});

% Recursive call

[Q,k] = quadtxstep(F, a, b, tol, fa, fc, fb, varargin{:});

fcount = k + 3;

% ––––––––––––––––––––––––––––––––––––––––––––––––––––––––

function [Q,fcount] = quadtxstep(F,a,b,tol,fa,fc,fb,varargin)

% Recursive subfunction used by quadtx.

h = b − a;

c = (a + b)/2;

fd = feval(F,(a+c)/2,varargin{:});

fe = feval(F,(c+b)/2,varargin{:});

Q1 = h/6 * (fa + 4*fc + fb);

Q2 = h/12 * (fa + 4*fd + 2*fc + 4*fe + fb);

if abs(Q2 − Q1) <= tol

Q = Q2 + (Q2 − Q1)/15;

fcount = 2;

else

[Qa,ka] = quadtxstep(F, a, c, tol, fa, fd, fc, varargin{:});

[Qb,kb] = quadtxstep(F, c, b, tol, fc, fe, fb, varargin{:});

Q = Qa + Qb;

fcount = ka + kb + 2;

end
```

RESTRICTED

```
function dydt = restricted(t,y)

mu = 1 / 82.45;

mustar = 1 − mu;

r13 = ((y(1) + mu)^2 + y(2)^2) ^ 1.5;

r23 = ((y(1) − mustar)^2 + y(2)^2) ^ 1.5;

dydt = [ y(3)

y(4)

(2*y(4) + y(1) − mustar*((y(1)+mu)/r13) − mu*((y(1)−mustar)/r23))

(−2*y(3) + y(2) − mustar*(y(2)/r13) − mu*(y(2)/r23)) ];
```

SIMPSON

```
% This function computes the

% Simpson quadrature rule for

% computing the integral of f(x) on

% the interval [a,b]

function I = simpson(fname,a,b)

fa = feval(fname,a);

fm = feval(fname, (a+b)/2);

fb = feval(fname, b);

I = ( fa + 4*fm + fb)*(b−a)/6;
```

SIMPSONCOM

```
% This function computes

% the Simpson composite quadrature rule

% with n subintervals

function I = simpsoncom(fname,a,b,n);

h = (b−a)/n;
```

```
x = [a:h:b];

f = feval(fname, x);

I = 0;

for k = 1:n

I = I + simpson('fname',x(k),x(k+1));

end
```

SPLINESLOPES

```
function d = splineslopes(h,delta)

% SPLINESLOPES Slopes for cubic spline interpolation.

% splineslopes(h,delta) computes d(k) = S'(x(k)).

% Uses not−a−knot end conditions.

% Diagonals of tridiagonal system

n = length(h)+1;

a = zeros(size(h)); b = a; c = a; r = a;

a(1:n−2) = h(2:n−1);

a(n−1) = h(n−2)+h(n−1);

b(1) = h(2);

b(2:n−1) = 2*(h(2:n−1)+h(1:n−2));

b(n) = h(n−2);

c(1) = h(1)+h(2);

c(2:n−1) = h(1:n−2);

% Right−hand side

r(1) = ((h(1)+2*c(1))*h(2)*delta(1)+h(1)^2*delta(2))/c(1);

r(2:n−1) = 3*(h(2:n−1).*delta(1:n−2)+h(1:n−2).*delta(2:n−1));

r(n) = (h(n−1)^2*delta(n−2)+(2*a(n−1)+h(n−1))*h(n−2)*delta(n−1))/a(n−1);

% Solve tridiagonal linear system
```

```
d = tridisolve(a,b,c,r);
```

SPLINTEX

```
function v = splinetx(x,y,u)

%SPLINETX Textbook spline function.

% v = splinetx(x,y,u) finds the piecewise cubic interpolatory

% spline S(x), with S(x(j)) = y(j), and returns v(k) = S(u(k)).

%

% See SPLINE, PCHIPTX.

% First derivatives

h = diff(x);

delta = diff(y)./h;

d = splineslopes(h,delta);

% Piecewise polynomial coefficients

n = length(x);

c = (3*delta − 2*d(1:n−1) − d(2:n))./h;

b = (d(1:n−1) − 2*delta + d(2:n))./h.^2;

% Find subinterval indices k so that x(k) <= u < x(k+1)

k = ones(size(u));

for j = 2:n−1

k(x(j) <= u) = j;

end

% Evaluate spline

s = u − x(k);

v = y(k) + s.*(d(k) + s.*(c(k) + s.*b(k)));

% −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
```

TRAPCOM

```
% This function computes

% the composite trapezoid quadrature rule with

% n subintervals

function I = trapezoid(fname,a,b,n);

h = (b−a)/n;

x = [a:h:b];

f = feval(fname, x);

I = h*(sum(f)−(f(1)+ f(length(f)))/2);
```

TWOBODY

```
function ydot = twobody(t,y)

r = sqrt(y(1)^2 + y(2)^2);

ydot = [y(3); y(4); −y(1)/r^3 ; −y(2)/r^3];
```