Fundamentos Matemáticos y Físicos para Informática Gráfica

Memoria de Laboratorio del Módulo 2: Matemáticas

Félix de las Pozas Álvarez - felix.delaspozas@urjc.es
Gael Rial Costas - gael.rial.costas@urjc.es
Antonio Arian Silaghi - aa.silaghi.2018@alumnos.urjc.es

Máster Universitario en Informática Gráfica, Juegos y Realidad Virtual 2022/2023

11 de diciembre de 2022

1. Hoja 1: Integración numérica

Para el apartado de integración numérica se han implementado los métodos del rectángulo (por la derecha, izquierda y punto medio), del trapecio y de Simpson.

```
function res = rectanguloA(f,a,b) % Rectángulo izquierda.
    res = f(a)*(b-a);
end

function res = rectanguloB(f,a,b) % Rectángulo derecha.
    res = f(b)*(b-a);
end

function res = rectanguloAB(f,a,b) % Rectángulo punto medio.
    res = f((a+b)/2)*(b-a);
end

function res = trapecio(f,a,b) % Fórmula del Trapecio
    res = ((b-a)/2)*(f(a)+f(b));
end

function res = simpson(f,a,b) % Fórmula de Simpson 1/3
    res = (b-a)*(f(a)+4*f((a+b)/2)+f(b))*1/6;
end
```

Listado 1: Fórmulas de integración.

Adicionalmente se ha implementado una función de Matlab que recibe como parámetros la función a integrar, una referencia al método y una lista de puntos en el eje X que servirán de intervalos.

```
% Para aplicar diversos métodos en el apartado B
function res = aplicarMetodo(func, met, X)
    res = 0;
    for i = 1:length(X)-1
        res = res + met(func, X(i), X(i+1));
    end
end
```

Listado 2: Función que aplica el método de integración en los intervalos.

1.1. Apartado A

Aplicamos los métodos al cálculo de $\int_0^1 e^x \, dx$ y los comparamos con el valor exacto computado con:

Metodo	abs(exacto-resultado)
Rectángulo izquierda	$8{,}59 \times 10^{-04}$
Rectángulo derecha	$8{,}59 \times 10^{-04}$
Rectángulo punto medio	$7{,}16 \times 10^{-08}$
Trapecio	$1{,}4319 \times 10^{-07}$
Simpson	$1{,}9984 \times 10^{-15}$

Tabla 1: Error absoluto cometido para el apartado A.

```
fA=@(x) exp(x);
exactoA = integral(fA,0,1); % valor exacto e^x en [0,1]
```

Podemos apreciar que el incremento en el número de intervalos aproxima el resultado a la solución exacta usando el mismo método y que, comparando la solución de cada método con el mismo número de intervalos (1000 en nuestro caso) con el valor exacto, vemos que el error cometido es mucho menor con la fórmula de Simpson que aproxima cuadráticamente.

1.2. Apartado B

Aplicamos los métodos al cálculo de $\int_0^{2\pi} cos(x^2 - 1)\, dx$ y los comparamos con el valor exacto y la aproximación computados con:

```
fB=@(x) cos((x.^2) - 1);
X = 0:pi/1000:2*pi;
Y = arrayfun(fB,X);
aproxiB = trapz(X,Y);
exactoB = integral(fB, 0, 2*pi);
```

Usando esta vez 2000 intervalos obtenemos unas conclusiones similares al apartado A.

2. Hoja 2: Resolución numérica de sistemas lineales

Para la resolución de los sistemas lineales dados para los apartados 1 y 2 implementamos los métodos de Jacobi y de Gauss-Seidel como funciones. Ambas reciben como parámetros la matriz A con los coeficientes y la matriz columna B con los valores resultado del sistema de ecuaciones Ax = B, una solución inicial **"sol"** para la iteración y el valor de tolerancia. Y ambas

Metodo	abs(exacto-resultado)
Rectángulo izquierda	$2{,}7603 \times 10^{-04}$
Rectángulo derecha	$2{,}615 \times 10^{-04}$
Rectángulo punto medio	$3{,}632 \times 10^{-06}$
Trapecio	$7{,}2638 \times 10^{-06}$
Simpson	$4{,}5356 \times 10^{-11}$

Tabla 2: Error absoluto cometido para el apartado B.

devuelven un vector solución y el número de iteraciones efecturadas, que nos servirá para medir la eficiencia del método.

```
function [res,count] = Jacobi(A, B, sol, tolerancia)
    next=zeros(1,length(sol));
    for i = 1:length(A)
        suma = 0;
        for j = 1:length(sol)
            if(i ~= j)
                suma = suma + A(i,j)*sol(j);
            end
        end
        next(i) = (B(i)-suma)/A(i,i);
    end
    count = 1;
    other = 0;
    for i = 1:length(sol)
        if(abs(sol(i)-next(i)) > tolerancia)
            [next, other] = Jacobi(A,B, next,tolerancia);
            break
        end
    end
    count = count + other;
    res = next;
end
```

Listado 3: Método de Jacobi.

```
function [res, count] = GaussSeidel(A, B, sol, tolerancia)
    next = sol;
    for i = 1:length(A)
        suma = 0;
        for j = 1:length(next)
            if(i ~= j)
                if(i > j)
                    suma = suma + A(i,j)*next(j);
                else
```

```
                suma = suma + A(i,j)*sol(j);
            end
        end
    end
    next(i) = (B(i)-suma)/A(i,i);
end
count = 1;
other = 0;
for i = 1:length(next)
    if(abs(sol(i)-next(i)) > tolerancia)
        [next, other] = GaussSeidel(A,B,next, tolerancia);
        break
    end
end
count = count + other;
res = next;
end
```

Listado 4: Método de Gauss-Seidel.

2.1. Apartado 1

Para el sistema de ecuaciones dado (que no incluiremos aquí por brevedad) la solución obtenida con ambos métodos es

$$x_1 = 3, x_2 = -2{,}5, x_3 = 7$$

pero con Gauss-Seidel obtenemos la solución en 3 iteraciones mientras que con el método de Jacobi usamos 4 con la misma toleracia de 0.01.

2.2. Apartado 2

Para el segundo sistema de cuatro ecuaciones con cuatro incógnitas podemos observar que Gauss-Seidel se aproxima a la solución de manera mucho más rápida y obtiene mejores resultados al reutilizar los valores de las variables ya computadas durante la iteración.

La solución dada en el enunciado es:

$$x_1 = 2{,}7273, x_2 = 0{,}4040, x_3 = 0{,}6364, x_4 = 0{,}1919$$

es la usada para computar el error.

Metodo	Tolerancia	Iteraciones (k)	max(Error^k)
Jacobi	0.1	5	> 100
Jacobi	0.01	12	17.46
Jacobi	0.001	21	1.19
Jacobi	0.0001	29	0.1155
Gauss-Seidel	0.1	4	31.9828
Gauss-Seidel	0.01	8	3.0241
Gauss-Seidel	0.001	12	0.3491
Gauss-Seidel	0.0001	16	0.0335

Tabla 3: Interaciones y error según la tolerancia para Jacobi y Gauss-Seidel.

3. Hoja 3: Raíces numéricas de funciones no lineales

A continuación se va a explicar la resolución de la práctica 4. Para esta práctica se han programado en Matlab dos métodos para encontrar las raíces numéricas de funciones no lineales. El método de bisección y el método de Newton-Raphson.

3.1. Método de bisección

Partiendo de una función $f(x)$ que este definida y sea continua en el intervalo $[x_1, x_2]$ tal que el signo de $f(x_1)$ es distinto al de $f(x_2)$ ó $f(x_1)f(x_2) < 0$. Según el teorema de Bolzano-Weierstrass, dentro del intervalo (x_1, x_2) tiene que existir al menos un valor de x tal que $f(x) = 0$.

El método de bisección es un método iterativo en el que en cada iteración se realizan los siguientes pasos:

1. Se calcula el punto medio m del intervalo como $m = (x_1 + x_2)/2$.
2. Si $f(m) = 0$ entonces m es una raíz y se termina el algoritmo. Si no, se calcula el signo de $f(m)$
3. En el intervalo actual $[x_1, x_2]$ se reemplaza o x_1 o x_2 con m tal que se siga cumpliendo que $f(x_1)f(x_2) < 0$ para el nuevo intervalo.
4. Se continua hasta el número de iteraciones determinado o hasta que el error sea menor que la cota predefinida.

Usando superíndices para representar la iteración, el máximo error que

se puede cometer dentro de un intervalo $[x_1^0, x_2^0]$ es $e^1 = x_2^1 - x_1^1$ ó $e^1 = (x_2^0 - x_2^0)/2$. Después de la n iteración, el error se puede calcular como

$$e^n = (x_2^0 - x_1^0)/2^n$$

El número de iteraciones n para que el error sea inferior a un valor ϵ viene dado por

$$n >= \ln((x_2^0 - x_1^0)/\epsilon)/\ln(2)$$

```
function res = Bisection(f, interval, error)
  iter = 0;
  totalIters = log((interval(2)-interval(1))/error)/log(2);

  while(iter < totalIters)
    xm = abs((interval(2)-interval(1))/2.0) + interval(1);

    if(f(xm) == 0.000)
      res = xm;
      break;
    elseif(sign(f(xm)) == sign(f(interval(2))))
      interval(2) = xm;
    elseif(sign(f(xm)) == sign(f(interval(1))))
      interval(1) = xm;
    end
    res = xm;
    iter = iter + 1;
  end
end
```

Listado 5: Algorítmo de bisección.

3.2. Método de Newton-Raphson

Dada una función de valores reales que sea continua y diferenciable, el método se basa en coger un punto inicial $(x_0^0, f(x_0^0))$ y seguir la derivada de $f(x)$ en dicho punto en cada iteración de tal manera que el resultado de la iteración número n viene dado por la ecuación.

$$x_0^n = x_0^{n-1} - \frac{f(x_0^{n-1})}{f'(x_0^{n-1})}$$

Este método puede no funcionar si alrededor del punto x o de la raíz existen máximos, mínimos o puntos de inflexión ($f'(x) = 0$).

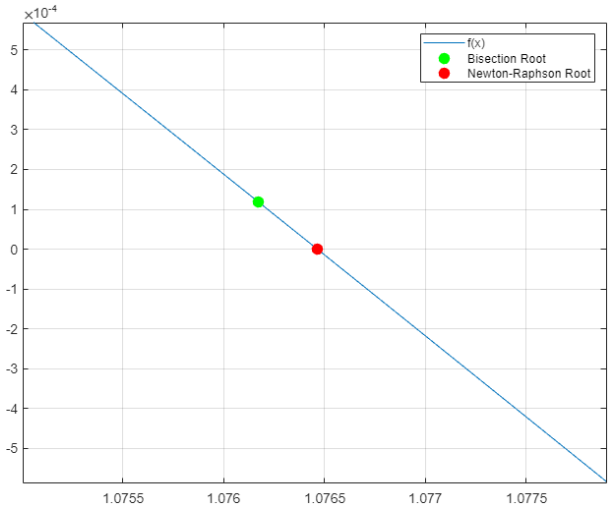
El error estimado ϵ en una iteración n se ha calculado como $\epsilon_n = |x_n - x_{n-1}|$.

```
function res = NewtonRaphson(f, df, startX, error)
  newX = startX;
  keepGoing = true;
  while(keepGoing)
    newX = startX - (f(startX) / df(startX));
    if ( abs(newX - startX) < error)
      keepGoing = false;
    end
    startX = newX;
  end
  res = newX;
end
```

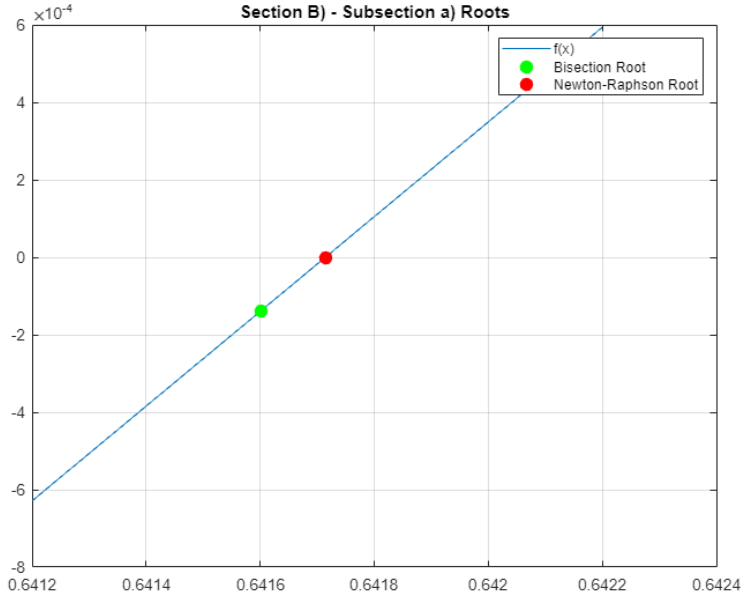
Listado 6: Algorítmo de Newton-Raphson

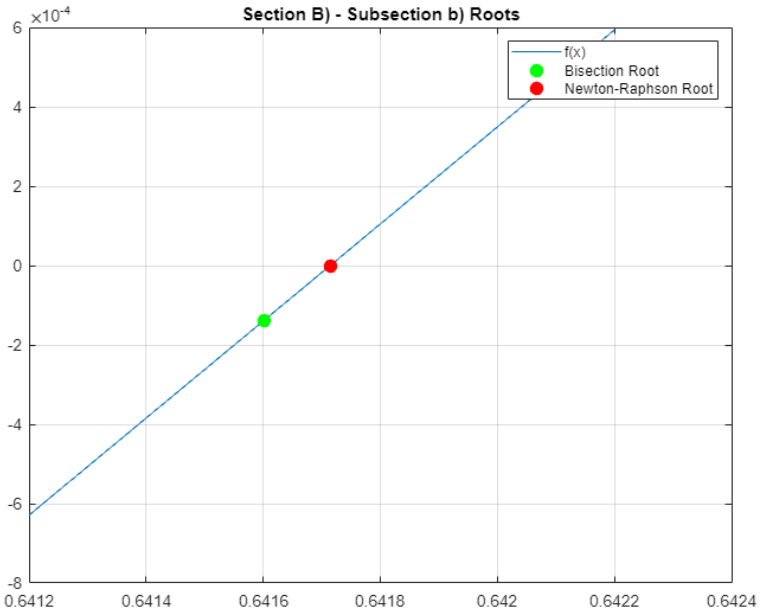
3.3. Resultados

3.3.1. Ejercicio A

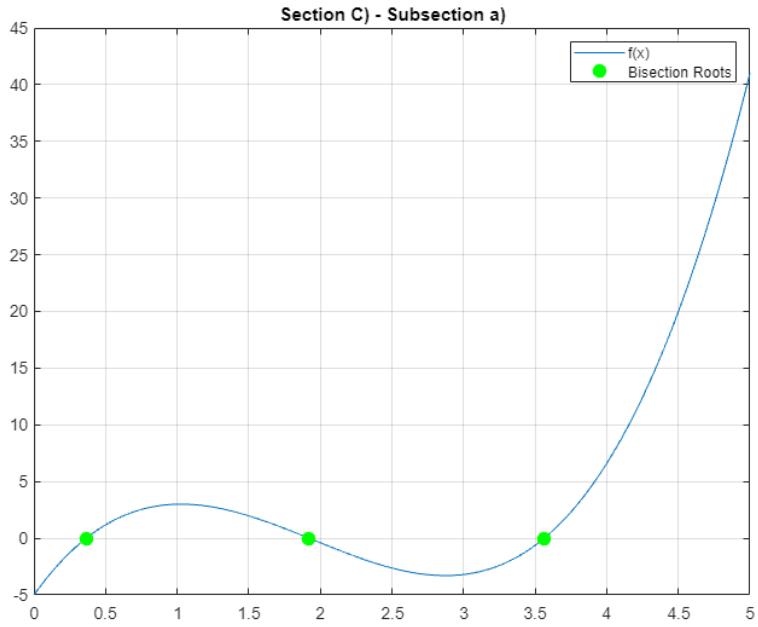


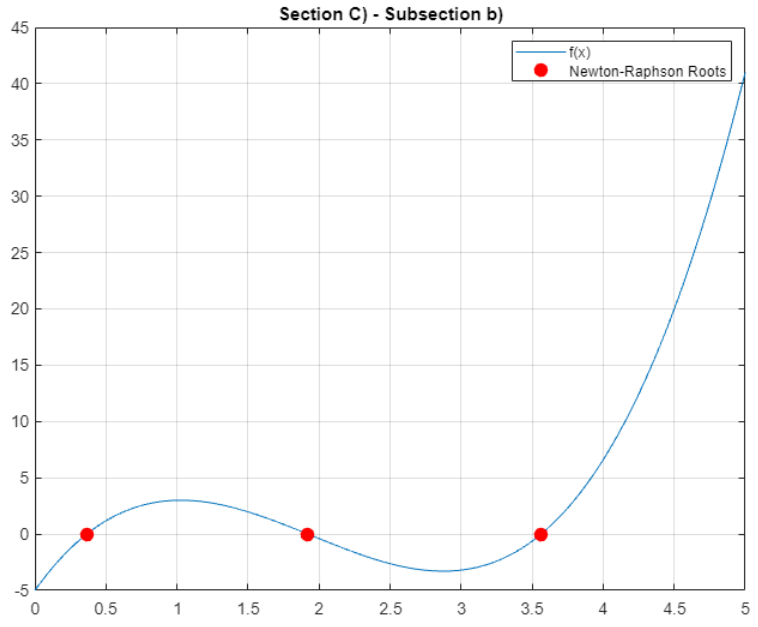
3.3.2. Ejercicio B



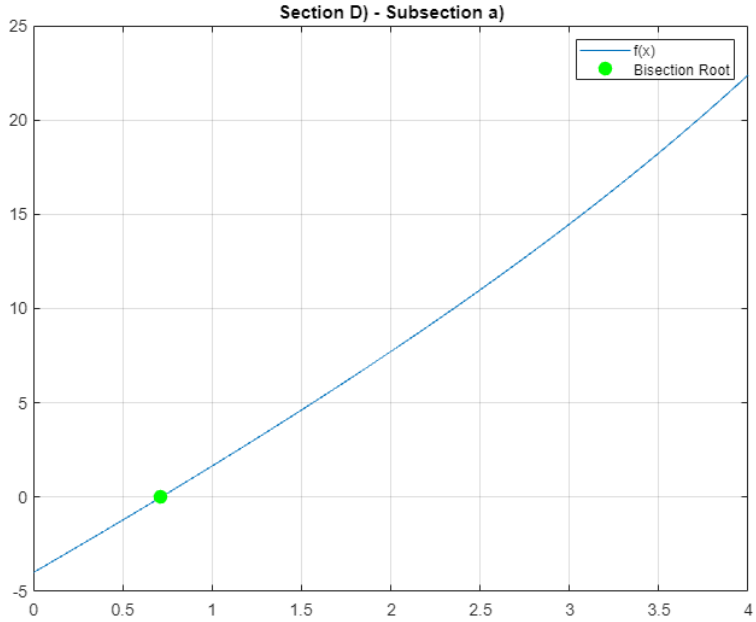


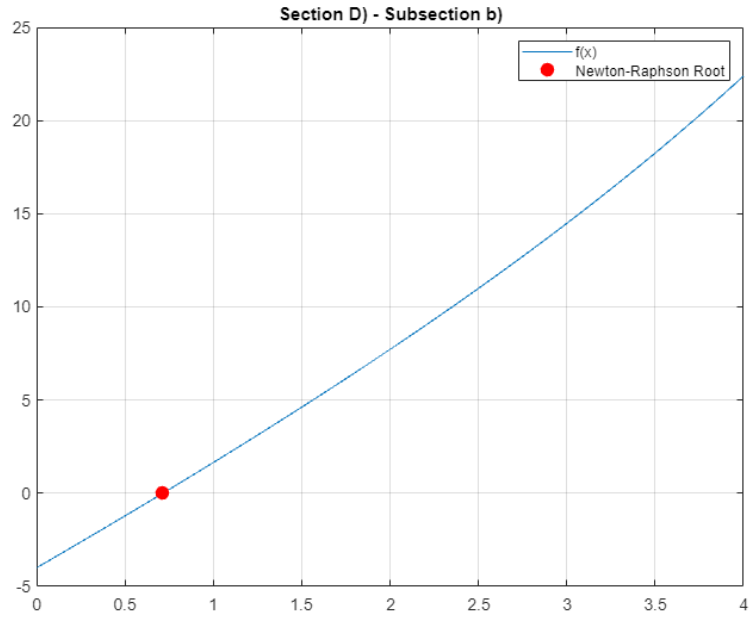
3.3.3. Ejercicio C





3.3.4. Ejercicio D





4. Métodos de 1 paso: Euler