Tarea 11: Mínimos Cuadrados e Integración
Métodos Numéricos

Rafael Alejandro García Ramírez

6 de noviembre de 2023

1. Introducción

La siguiente práctica se divide en mínimos cuadrados e integración: para mínimos cuadrados se presentan funciones que utilizan kernels para construir una curva de forma que para un conjunto de datos $x^{(k)}, y^{(k)}$ tenemos una función aproximadora tal que

$$y = f(x) = \sum_{i=1}^{m} w_i \phi_i(x)$$

Tal que para una matriz $\phi_j == [\phi_j(x^{(1)}, \phi_j(x^{(2)}, \ldots, \phi_j(x^{(p)}]^T$ tenemos para Λ una matriz diagonal de λ_r para $1 \leq r \leq m$ la siguiente relación

$$(\Phi^T \Phi + \Lambda) w = \Phi^T y$$

De donde buscamos obtener w que serán nuestros coeficientes (pesos) para cada función ϕ_j. Esto nos permite construir curvas más completas que el típico mínimos cuadrados pues realmente el kernel puede ser, en principio, cualquier función.

La integración de funciones por otra parte nos permite determinar el valor de integrales definidas difíciles de calcular analíticamente, estos en lo general utilizan técnicas para derivadas numéricas y se construye una suma sobre todo un intervalo a integrar. Los diferentes métodos en esencia consisten en modificar el como dividir este intervalo para alcanzar una mayor precisión, estos se pueden dividir en aquellos con intervalo igualmente espaciado (como Simpson y Newton-Cotes) o en aquellos con intervalo no igualmente espaciados (como lo es el método de la Cuadratura Gaussiana).

A continuación veremos la implementación de estos dos temas, utilizando Newton-Cotes, Cuadratura Gaussiana y la interpolación mediante mínimos cuadrados con kernels polinómicos, trigonométricos y esponenciales.

2. Pseudocódigos

2.1. Newton-Cotes Abierto

Algorithm 1 Método de Newton-Cotes Abierto

Require: $func$, a, b, n
Ensure: Resultado de la integral

function NewtonCotesAbierto($func, a, b, n$)
  if $n < 0$ **o** $n > 3$ **then**
    Imprimir(Error: n debe ser un entero entre 0 y 3)
    return 0.0
  end if
  $h \leftarrow (b - a)/(n + 2.0)$
  $resultado \leftarrow 0.0$
  if $n = 0$ **then**
    $resultado \leftarrow 2 \cdot h \cdot func(a + h)$
  else if $n = 1$ **then**
    $resultado \leftarrow (3.0 \cdot h/2.0) \cdot (func(a + h) + func(a + 2.0 \cdot h))$
  else if $n = 2$ **then**
    $resultado \leftarrow (4.0 \cdot h/3.0) \cdot (2.0 \cdot func(a + h) - func(a + 2.0 \cdot h) + 2.0 \cdot func(a + 3.0 \cdot h))$
  else if $n = 3$ **then**
    $resultado \leftarrow (5.0 \cdot h/24.0) \cdot (11.0 \cdot func(a + h) + func(a + 2.0 \cdot h) + func(a + 3.0 \cdot h) + 11.0 \cdot func(a + 4.0 \cdot h))$
  end if
  return $resultado$
end function

2.2. Newton-Cotes Cerrado

Algorithm 2 Método de Newton-Cotes Cerrado

Require: $func$, a, b, n
Ensure: Resultado de la integral
function NewtonCotesCerrado($func, a, b, n$)
  if $n < 1$ **o** $n > 4$ **then**
    Imprimir(Error: n debe ser un entero entre 1 y 4)
    return 0.0
  end if
  $h \leftarrow (b - a)/n$
  $resultado \leftarrow 0.0$
  if $n = 1$ **then**
    $resultado \leftarrow 0.5 \cdot h \cdot (func(a) + func(b))$
  else if $n = 2$ **then**
    $resultado \leftarrow (h/3.0) \cdot (func(a) + 4.0 \cdot func(a + h) + func(b))$
  else if $n = 3$ **then**
    $resultado \leftarrow (3.0 \cdot h/8.0) \cdot (func(a) + 3.0 \cdot func(a + h) + 3.0 \cdot func(a + 2.0 \cdot h) + func(b))$
  else if $n = 4$ **then**
    $resultado \leftarrow (2.0 \cdot h/45.0) \cdot (7.0 \cdot func(a) + 32.0 \cdot func(a + h) + 12.0 \cdot func(a + 2.0 \cdot h) + 32.0 \cdot func(a + 3.0 \cdot h) + 7.0 \cdot func(b))$
  end if
  return $resultado$
end function

2.3. Mínimos cuadrados interpolación polinómica

Algorithm 3 Interpolación de Cuadrados Mínimos con Polinomios

Require: $N, x, y, w, regulador$
Ensure: Vector de pesos w calculado
function InterpolacionCuadradosPolinomio($N, x, y, w, regulador$)
  $PI \leftarrow 3.14159265358979323846$
  Crear matriz phi de tamaño $N \times N$
  for $i \leftarrow 0$ **hasta** $N - 1$ **do**
    for $j \leftarrow 0$ **hasta** $N - 1$ **do**
      $phi[i][j] \leftarrow$ potencia$(x[i], j)$
    end for
  end for
  Crear matriz $phiT$ de tamaño $N \times N$
  Calcular la matriz transpuesta $phiT$ de phi
  Crear matriz $phiTphi$ de tamaño $N \times N$
  Multiplicar matrices $phiT$ y phi y almacenar el resultado en $phiTphi$
  for $i \leftarrow 0$ **hasta** $N - 1$ **do**
    $phiTphi[i][i] \leftarrow phiTphi[i][i] + regulador$
  end for
  Crear vector $phiTy$ de tamaño N
  Multiplicar matriz $phiT$ y vector y y almacenar el resultado en $phiTy$
  Crear vector d de tamaño N
  Multiplicar matriz $phiT$ y vector y y almacenar el resultado en d
  Crear matriz L de tamaño $N \times N$
  Crear matriz U de tamaño $N \times N$
  Realizar descomposición de Doolittle de la matriz $phiTphi$ en las matrices L y U
  Crear vector $dummy$ de tamaño N
  Resolver el sistema triangular inferior $L \cdot dummy = d$
  Resolver el sistema triangular superior $U \cdot w = dummy$
  Liberar memoria
end function

2.4. Mínimos cuadrados interpolación trigonométrica

Algorithm 4 Interpolación de Cuadrados Mínimos con Cosenos

Require: $N, x, y, w, regulador$
Ensure: Vector de pesos w calculado

function InterpolacionCuadradosCoseno($N, x, y, w, regulador$)
    $PI \leftarrow 3.14159265358979323846$
    Crear matriz phi de tamaño $N \times N$
    for $i \leftarrow 0$ **hasta** $N - 1$ **do**
        for $j \leftarrow 0$ **hasta** $N - 1$ **do**
            $phi[i][j] \leftarrow \cos(j \cdot x[i] \cdot \frac{PI}{6.0})$
        end for
    end for
    Crear matriz $phiT$ de tamaño $N \times N$
    Calcular la matriz transpuesta $phiT$ de phi
    Crear matriz $phiTphi$ de tamaño $N \times N$
    Multiplicar matrices $phiT$ y phi y almacenar el resultado en $phiTphi$
    for $i \leftarrow 0$ **hasta** $N - 1$ **do**
        $phiTphi[i][i] \leftarrow phiTphi[i][i] + regulador$
    end for
    Crear vector $phiTy$ de tamaño N
    Multiplicar matriz $phiT$ y vector y y almacenar el resultado en $phiTy$
    Crear vector d de tamaño N
    Multiplicar matriz $phiT$ y vector y y almacenar el resultado en d
    Crear matriz L de tamaño $N \times N$
    Crear matriz U de tamaño $N \times N$
    Realizar descomposición de Doolittle de la matriz $phiTphi$ en las matrices L y U
    Crear vector $dummy$ de tamaño N
    Resolver el sistema triangular inferior $L \cdot dummy = d$
    Resolver el sistema triangular superior $U \cdot w = dummy$
    Liberar memoria
end function

2.5. Mínimos cuadrados interpolación exponencial

Algorithm 5 Interpolación de Cuadrados Mínimos con Exponenciales

Require: $N, x, y, w, regulador$
Ensure: Vector de pesos w calculado

function InterpolacionCuadradosExponencial($N, x, y, w, regulador$)
  Crear matriz phi de tamaño $N \times N$
  for $i \leftarrow 0$ **hasta** $N - 1$ **do**
    for $j \leftarrow 0$ **hasta** $N - 1$ **do**
      $r \leftarrow (x[i] - x[j]) \cdot (x[i] - x[j])$
      $phi[i][j] \leftarrow \exp(-r)$
    end for
  end for
  Crear matriz $phiT$ de tamaño $N \times N$
  Calcular la matriz transpuesta $phiT$ de phi
  Crear matriz $phiTphi$ de tamaño $N \times N$
  Multiplicar matrices $phiT$ y phi y almacenar el resultado en $phiTphi$
  for $i \leftarrow 0$ **hasta** $N - 1$ **do**
    $phiTphi[i][i] \leftarrow phiTphi[i][i] + regulador$
  end for
  Crear vector $phiTy$ de tamaño N
  Multiplicar matriz $phiT$ y vector y y almacenar el resultado en $phiTy$
  Crear vector d de tamaño N
  Multiplicar matriz $phiT$ y vector y y almacenar el resultado en d
  Crear matriz L de tamaño $N \times N$
  Crear matriz U de tamaño $N \times N$
  Realizar descomposición de Doolittle de la matriz $phiTphi$ en las matrices L y U
  Crear vector $dummy$ de tamaño N
  Resolver el sistema triangular inferior $L \cdot dummy = d$
  Resolver el sistema triangular superior $U \cdot w = dummy$
  Liberar memoria
end function

3. Resultados

A continuación presentamos la resolución de los problemas presentados en la tarea:

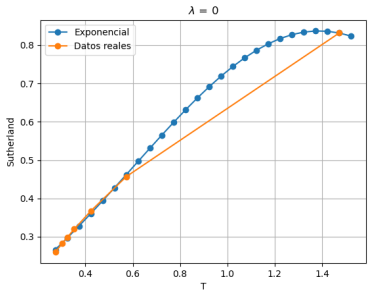
1. a. Los resultados de las evaluaciones son las siguientes

```
Resultados de la evaluacion de la funcion de Sutherland:
T[0] = 0.273000, viscosidad = 0.259708
T[1] = 0.303000, viscosidad = 0.283062
T[2] = 0.323000, viscosidad = 0.298058
T[3] = 0.353000, viscosidad = 0.319769
T[4] = 0.423000, viscosidad = 0.367209
T[5] = 0.573000, viscosidad = 0.456974
T[6] = 1.473000, viscosidad = 0.831914
```

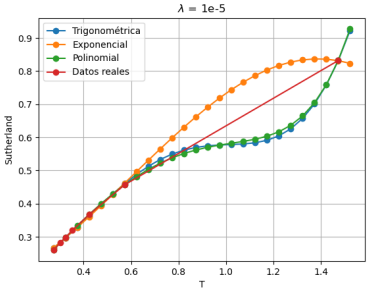
b. Los datos al evaluar en las funciones generadas para cada λ y sus respectivos errores absolutos es

```
Valor a evaluar T = 1.200000
Valor real de la viscosidad = 0.736428

Lambda = 0.000000
viscosidad polinomica = 0.801678
El error absoluto es 0.065249721417
viscosidad coseno = 3.246349
El error absoluto es 2.509920838789
viscosidad exponencial ẹxp(-x^2) = 0.779528
El error absoluto es 0.043099127443


Lambda = 0.006738
viscosidad polinomica = 0.909667
El error absoluto es 0.173238113329
viscosidad coseno = 0.839817
El error absoluto es 0.103388655138
viscosidad exponencial ẹxp(-x^2) = 0.804164
El error absoluto es 0.067735534521


Lambda = 0.000912
viscosidad polinomica = 0.813598
El error absoluto es 0.077169807073
viscosidad coseno = 0.836126
El error absoluto es 0.099697905061
viscosidad exponencial ẹxp(-x^2) = 0.811369
El error absoluto es 0.074940142974
```

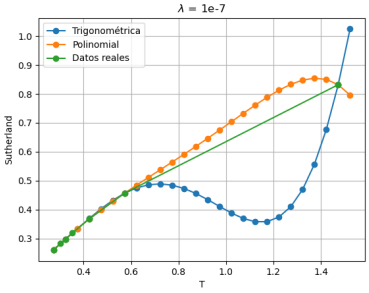
c. Los gráficos de las funciones son las siguientes



(a) $\lambda = 0$



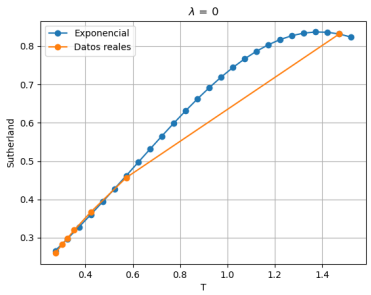
(b) $\lambda = 1e-5$



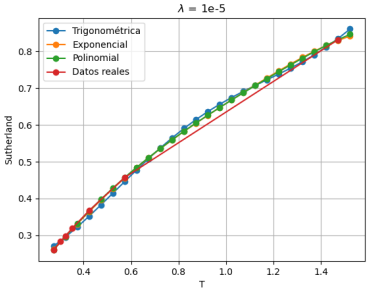
(c) $\lambda = 1e-7$

En el caso para $\lambda = 0$ las funciones polinómicas y trigonométricas se topan con una división entre cero al momento de realizar la resolución LU.

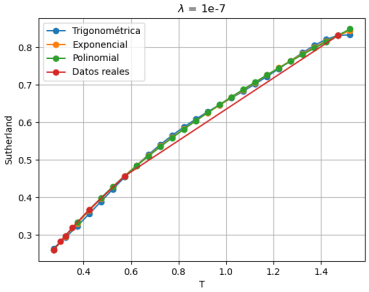
d. Repitiendo los cálculos con los nuevos puntos tenemos las siguientes gráficas (y con los mismos problemas)



(a) $\lambda = 0$



(b) $\lambda = 1e-5$



(c) $\lambda = 1e-7$

Podemos concluir que la exactitud de la interpolación varía bastante con respecto a la decisión de nuestra λ, este puede hacer que se ajuste bastante bien o al contrario que tome comportamientos diferentes.

2. Se creó una librería para realizar cada tipo de Newton-Cotes, a continuación presentamos el código que resuelve el problema

```
double newton_cotes_cerrado(double(*func)(double), double a, double b, int n){
    if (n < 1 || n > 4) {
       printf("\n---->Error: n debe ser un entero entre 1 y 5\n");
       return 0.0;
    }
    double h = (b-a)/n;
    double resultado;
    switch (n) {
        case 1:
            resultado = 0.5*h*(func(a) + func(b));
            break;
        case 2:
            resultado = (h/3.0)*(func(a) + 4.0*func(a + h) + func(b));
            break;
        case 3:
            resultado = (3.0*h/8.0)*(func(a) + 3.0*func(a + h) + 3.0*func(a + 2.0*h) + func(b));
            break;
        case 4:
            resultado =(2.0*h/45.0)*(7.0*func(a) + 32.0*func(a + h) + 12.0*func(a + 2.0*h) + 32.0*
    func(a + 3.0*h) + 7.0*func(b));
            break;
    }
    return resultado;
}

double newton_cotes_abierto(double(*func)(double), double a, double b, int n){
    if (n < 0 || n > 3) {
       printf("\n---->Error: n debe ser un entero entre 0 y 3\n");
       return 0.0;
    }
    double h = (b-a)/(n+2.0);
    double resultado;
    switch (n) {
    case 0:
        resultado = 2*h*func(a + h);
        break;
    case 1:
        resultado = (3.0*h/2.0)*(func(a + h) + func(a + 2.0*h));
        break;
    case 2:
        resultado = (4.0*h/3.0)*(2.0*func(a + h) - func(a + 2.0*h) + 2.0*func(a + 3.0*h));
        break;
    case 3:
        resultado = (5.0*h/24.0)*(11.0*func(a + h) + func(a + 2.0*h) + func(a + 3.0*h) + 11.0*func
    (a + 4.0*h));
        break;
    }
    return resultado;
}
double cuadratura_gaussiana(double a, double b, double(*func)(double), int n){
    if (n < 1 || n > 4) {
        printf("\n---->Error: n debe ser un entero entre 1 y 4\n");
        return 0.0;
    }
    if (n == 1){
        return 2 * func((a + b) / 2)* (b - a) / 2;
    }
    else if (n == 2){
        double coeficientes[] = {1.0, 1.0};
        double raices[] = {0.5773502692, -0.5773502692};
        double resultado = 0.0;
        double cambio_limites;
        for(int i = 0; i < 2; i++) {
            cambio_limites = ((b-a)*raices[i] + (b+a))/2;
            resultado += coeficientes[i]*func(cambio_limites);
        }
        return resultado*(b - a) / 2.0;
    }
    else if (n == 3){
        double coeficientes[] = {0.555555555555556, 0.888888888888889, 0.555555555555556};
        double raices[] = {-0.774596669241483, 0.0, 0.774596669241483};
        double resultado = 0.0;
        double cambio_limites;
        for(int i = 0; i < 3; i++){
```

```
            cambio_limites = ((b-a)*raices[i] + (b+a))/2;
            resultado += coeficientes[i]*func(cambio_limites);
        }
        return resultado*(b - a) / 2.0;
    }
    else if (n == 4){
        double coeficientes[] = {0.652145154862546, 0.652145154862546, 0.347854845137454,
    0.347854845137454};
        double raices[] = {-0.339981043584856, 0.339981043584856, -0.861136311594053,
    0.861136311594053};
        double resultado = 0.0;
        double cambio_limites;
        for(int i = 0; i < 4; i++){
            cambio_limites = ((b-a)*raices[i] + (b+a))/2;
            resultado += coeficientes[i]*func(cambio_limites);
        }
        return resultado*(b - a) / 2.0;
    }
}
```

Con ello se evaluaron las funciones siguientes para los distintos n

$$\int_0^{\pi/4} \sin x dx \ , \ \int_1^{1.5} x^2 \ln x dx, \ \int_0^1 x^2 e^{-x} dx$$

Los resultados para cada caso se presentan a continuación

Función	Tipo	n	I
$\sin(x)$	Abierta	0	0.300559
		1	0.297988
		2	0.292859
		3	0.292869
	Cerrada	1	0.277680
		2	0.292933
		3	0.292911
		4	0.292893
$x^2 \log(x)$	Abierta	0	0.174331
		1	0.180313
		2	0.192272
		3	0.192268
	Cerrada	1	0.228074
		2	0.192245
		3	0.192253
		4	0.192259
$x^2 \exp(-x)$	Abierta	0	0.151633
		1	0.153900
		2	0.159043
		3	0.159514
	Cerrada	1	0.183940
		2	0.162402
		3	0.161410
		4	0.160611
$\sin(x)$	Cuadratura Gaussiana	1	0.300559
		2	0.292867
		3	0.292893
		4	0.292893
		5	0.292893
$x^2 \log(x)$	Cuadratura Gaussiana	1	0.174331
		2	0.192269
		3	0.192259
		4	0.192259
		5	0.192259
$x^2 \exp(-x)$	Cuadratura Gaussiana	1	0.151633
		2	0.159410
		3	0.160595
		4	0.160603
		5	0.160603

Hemos de aclarar que los valores reales de las integrales son las siguientes

$$\int_0^{\pi/4} \sin x dx = 1 - \frac{1}{\sqrt{2}} \approx 0.2928932188134524755\ldots$$

$$\int_1^{1.5} x^2 \ln x dx \approx 0.192259$$

$$\int_0^1 x^2 e^{-x} dx = 2 - \frac{5}{e} \approx 0.160602794142788392\ldots$$

3. ▪ **Extrapolación de Richarson**: Recordemos las derivadas hacia adelante y hacia atrás

$$\text{Hacia adelante} \quad f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}$$

$$\text{Hacia atrás} \quad f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{x_{i+1} - x_i}$$

En ambos caso podemos ver que existen problemas cuando $x_{i+1} - x_i \to 0$, lo cuál es contrastante debido a que su exactitud dependerá de esta diferencia a alguna potencia, por lo que entre mayor exactitud mayor este riesgo. Si suponemos que tenemos el valor real de la derivada V_r podemos encontrar la relación del error real ε_r y el error aproximado por nuestra derivada $\varepsilon_{a,h}$ para un paso $h = x_{i+1} - x_i$ es de tal forma que

$$\varepsilon_r = V_r - \varepsilon_{a,h}$$

Ahora recordemos que el error para la derivada centrada es del orden de $O(h^2)$ que esto será $O(h^2) \to ch^2$ para una c constante [1] por lo tanto $\varepsilon_r = ch^2$ así tenemos que

$$V_r = V_{a,h} + ch^2$$

Si hacemos una simple sustitución $h \to h/2$ tenemos que $4V_r = 4V_{a,h/2} + ch^2$ y restando esto a nuestra expresión anterior tenemos

$$V_r = \frac{4}{3}V_{a,h/2} - \frac{1}{3}V_{a,h}$$

Por lo tanto para $V_{a,h}$ el valor aproximado con un paso h tenemos para ${f_c}'_h$ la derivada centrada

$$f'(x_i) = \frac{4}{3}{f_c}'_{h/2} - \frac{1}{3}{f_c}'_h$$

Esta última expresión es conocida como la extrapolación de Richarson y permite un error de orden $O(h^4)$ reduciendo el riesgo de división por cero que se tenía en los primeros métodos.

Daremos un ejemplo para una función sencilla $f(x) = e^x$ cuya derivada ya es conocida. Para un paso $h = 0.01$ en $x_0 = 1$ tenemos lo siguientes errores absolutos con $f'(x_0 = 1) = e^1$ y comparándolo con el método de la derivada centrada

$$\varepsilon_{centrada} = 4.530492366905392 \times 10^{-5}$$

$$\varepsilon_{richardson} = 5.6690652172619593 \times 10^{-11}$$

¡El error es un millón de veces más pequeño!

Referencia específica: *Arévalo Ovalle, D., Bernal Yermanos, M. Á., & Posada Restrepo, J. A. (2021). Métodos numéricos con Python.*

- **Método de Romberg**: Este método consiste en tomar alguna forma de integración y mejorar su precisión con base al método de Richardson, para ello recordemos (por ejemplo) la regla compuesta trapezoidal para $h = \frac{b-a}{n}$ y $x_j = a + jh$

$$\int_a^b f(x)dx = \frac{h}{2}\left[f(a) + 2\sum_{j=1}^{n-1} f(x_j) + f(b)\right] - \frac{b-a}{12}h^2 f''(\mu)$$

Sea entonces $R_{i,j}$ donde i es el orden n para la forma del trapecio y j indica el término h^{2j} del error, utilizando la forma para $f'(x_i)$ encontrada en Richardson

$$R_{k,2} = R_{k,1} + \frac{1}{3}(R_{k,1} - R_{k-1,1}), \quad k = j, j+1, \ldots$$

[1] Esto es, que nuestro error es c veces h^2.

Con esta idea podemos reescribir el siguiente término para $j = 3$ como

$$R_{k,3} = R_{k,2} + \frac{1}{15}(R_{k,2} - R_{k-1,2}), \quad k = j, j+1, \ldots$$

Si seguimos con esta sustitución repetitivamente, podemos llegar a la forma general

$$R_{k,j} = R_{k,j-1} + \frac{1}{4^{j-1} - 1}(R_{k,j-1} - R_{k-1,j-1}), \quad k = j, j+1, \ldots$$

Notamos así que en general requerimos de $n(n+1)/2$ cálculos. Como ejemplo integremos la expresión e^x en el intervalo $[0, 1]$, para ello la solución exacta es

$$\int_0^1 e^x dx = e - 1 \approx 1.7182818285$$

Realizando el método de Romberg para 5 iteraciones tenemos los siguientes resultados

i	h_i	$R_{i,0}$	$R_{i,1}$	$R_{i,2}$	$R_{i,3}$	$R_{i,4}$	$R_{i,5}$
0	1.000000	1.859141					
1	0.500000	1.753931	1.718861				
2	0.250000	1.727222	1.718319	1.718283			
3	0.125000	1.720519	1.718284	1.718282	1.718282		
4	0.062500	1.718841	1.718282	1.718282	1.718282	1.718282	
5	0.031250	1.718282	1.718282	1.718282	1.718282	1.718282	1.718282

Tenemos así un error absoluto de 3×10^{-14}.

Referencia específica: *Richard. L. Burden y J. Douglas Faires, Análisis Numérico, 7a Edición, Editorial Thomson Learning, 2002.*

4. Conclusiones

Para los diferentes interpolaciones notamos que si bien existen diferencias entre el tipo de kernel que se escoge, la mayor diferencia suele recaer al momentos de identificar el regulador adecuado para cada sistema y kernel. En lo general pudimos encontrar algunos $\lambda = \lambda_i, \forall i$ que ayudaran al sistema pero es de aclarar que un λ que sea diferente para todo i permitiría ajustar aquellos valores de la matriz $\Phi^T\Phi$ de tal forma en que se reduzcan los errores en a menos todo el rango de los valores x.

Dentro de la sección de integración podemos observar que la cuadratura gaussiana fue aquella que más se acercó al valor real de las integrales, un resultado que es de esperarse dado a su naturaleza adaptativa de subintervalos.

5. Referencias

1. Arévalo Ovalle, D., Bernal Yermanos, M. Á., & Posada Restrepo, J. A. (2021). Métodos numéricos con Python.
2. Kong, Q., Siauw, T., & Bayen, A. (2020). Python programming and numerical methods: A guide for engineers and scientists. Academic Press.
3. Richard. L. Burden y J. Douglas Faires, Análisis Numérico, 7a Edición, Editorial Thomson Learning, 2002.
4. Samuel S M Wong, Computational Methods in Physics and Enginering, Ed. World Scientific, 3rd Edition, 1997.
5. Teukolsky, S. A., Flannery, B. P., Press, W. H., & Vetterling, W. T. (1992). Numerical recipes in C. SMR, 693(1), 59-70.
6. William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, Numerical Recipes: The Art of Scientific Computing, 3rd Edition, Cambridge University Press, 2007