Computación Científica II

Laboratorio 2

Victor Gonzalez Rodriguez
victor.gonzalezro@alumnos.usm.cl
2773029-9

29 de octubre de 2012

1. Introducción

El cómputo de funciones complejas, o de las cuales no se tiene certeza de su naturaleza, puede ser un completo dolor de cabeza para científicos o estudiantes de ingeniería. Para una máquina, esto no pasa a ser algo más sencillo, es por esto, y para facilitar la vida de todos, es que existen algoritmos que facilitan el cálculo. Esto resulta especialmente útil cuando se tienen muestras de un experimento o un fenómeno del cual no se sabe con certeza cómo modelar. Los siguientes algoritmos presentados en el siguiente documento, implementan computacionalmente los métodos algorítimicos para resolver parte de estas dificultades.

2. Objetivos

- Implementar en Python algoritmos numéricos y de interpolación.
- Desarrollar casos de pruebas.
- Comprobar las hipótesis y sus resultados.
- Analizar resultados.

3. Preguntas

En esta sección se esbozará parte del enunciado original para dar un contexto. Toda información vaga respecto al enunciado se puede aclarar revisando el documento de los enunciados del laboratorio.

3.1. Interpolación Polinomial

3.1.1. Función de Interpolación Polinomial

Se nos pide desarrollar una función que permita calcular la interpolación polinomial para una función $f(x)$, de manera que se pueda ejecutar como:

```
>> y_int = inter_pol(x_int, n, pol)
```

Donde `x_int` $\in [-1, 1]$, `n` un número que particiona el dominio y `pol` un string que puede ser `diff` para calcular mediante *diferecias divididas*, o `spl` para calcular mediante *splines*.

Para esto, se desarrolló el siguiente código[1], el cual puede ser encontrado en el archivo *inter_pol.py*:

```
def inter_pol(x_int, n, pol):
    ran = [-1,1]
    if x_int < ran[0] or x_int > ran[1]:
        raise ValueError("Invalid 'x_int' value")

    x = partition(ran,n) # partition the domain in n parts
    y = evaluate(x) # evaluate each point of the partition

    # execute the chosen interpolation
    if pol == "diff":
        return divided_differences(x,y,x_int)
    elif pol == "spl":
        return splines(x,y,x_int)
    else:
        raise ValueError("Unknown interpolation method")
```

3.1.2. Benchmark Comparativo

Se nos pide desarrollar un benchmark entre los dos métodos interpoladores: diferencias divididas y splines. Para realizar el benchmark, se utilizará el mismo código desarrollado en la pregunta anterior, utilizando cierta data de prueba, la cual se puede ver tabulada junto a los resultados del benchmark en la siguiente tabla:

[1]Esta porción de código y todas las que saldrán mas adelante han sido modificadas para reducir el espacio utilizado en este documento, por lo cual se han omitido comentarios e incluso algunas funciones. El código completo se puede obtener en los archivos correspondientes.

n	x	$y_k = f(x)$	y_int	pol	**Error relativo**	**Tiempo de cómputo**
	$-1/2$	0,328779511373	0,12484230553	diff	0,620285628478	0,00191378593445
	$-1/4$	0,567902536146	0,0936317291479	diff	0,835127115678	0,00162506103516
2	0	-0	0,0	diff	-0	0,00209498405457
	$1/4$	$-0{,}956250428074$	$-0{,}156052881913$	diff	0,836807516806	0,00160980224609
	$1/2$	$-1{,}06593483416$	$-0{,}374526916591$	diff	0,648639950034	0,00167298316956
	$-1/2$	0,328779511373	0,012137446371	spl	0,963083324991	0,00269913673401
	$-1/4$	0,567902536146	0,329228232814	spl	0,420273353508	0,00326800346375
2	0	-0	0,0	spl	-0	0,00271391868591
	$1/4$	$-0{,}956250428074$	$-0{,}360438809196$	spl	0,623070695066	0,00264406204224
	$1/2$	$-1{,}06593483416$	$-0{,}740384228632$	spl	0,305413234559	0,00262188911438
	$-1/2$	0,328779511373	0,328779511373	diff	0,0	0,00251698493958
	$-1/4$	0,567902536146	0,297259781507	diff	0,476565497445	0,00254106521606
4	0	-0	0,0	diff	-0	0,00262403488159
	$1/4$	$-0{,}956250428074$	$-0{,}512015531688$	diff	0,464559160806	0,00249195098877
	$1/2$	$-1{,}06593483416$	$-1{,}06593483416$	diff	-0	0,00248289108276
	$-1/2$	0,328779511373	0,328779511373	spl	0,0	0,00339508056641
	$-1/4$	0,567902536146	0,541009340544	spl	0,0473553011129	0,00333118438721
4	0	-0	0,0	spl	-0	0,00332903862
	$1/4$	$-0{,}956250428074$	$-0{,}669001678844$	spl	0,300390714396	0,00331616401672
	$1/2$	$-1{,}06593483416$	$-1{,}06593483416$	spl	-0	0,00331687927246
	$-1/2$	0,328779511373	0,328779511373	diff	0,0	0,0047550201416
	$-1/4$	0,567902536146	0,567902536146	diff	0,0	0,00483083724976
8	0	-0	$1{,}11022302463e-16$	diff	$1{,}11022302463e-16$	0,00470805168152
	$1/4$	$-0{,}956250428074$	$-0{,}956250428074$	diff	$4{,}64406808941e-16$	0,00468897819519
	$1/2$	$-1{,}06593483416$	$-1{,}06593483416$	diff	$1{,}45816815875e-15$	0,00519704818726
	$-1/2$	0,328779511373	0,328779511373	spl	0,0	0,00505805015564
	$-1/4$	0,567902536146	0,567902536146	spl	0,0	0,00494980812073
8	0	-0	0,0	spl	-0	0,00486588478088
	$1/4$	$-0{,}956250428074$	$-0{,}956250428074$	spl	-0	0,00487399101257
	$1/2$	$-1{,}06593483416$	$-1{,}06593483416$	spl	-0	0,00477004051208

n	x	$y_k = f(x)$	`y_int`	`pol`	**Error relativo**	**Tiempo de cómputo**
	$-1/2$	0,328779511373	0,328779511373	`diff`	0,0	0,0108621120453
	$-1/4$	0,567902536146	0,567902536146	`diff`	0,0	0,0108370780945
16	0	-0	$-8{,}32667268469e-17$	`diff`	$8{,}32667268469e-17$	0,0106379985809
	$1/4$	$-0{,}956250428074$	$-0{,}956250428074$	`diff`	$9{,}28813617881e-16$	0,0117888450623
	$1/2$	$-1{,}06593483416$	$-1{,}06593483416$	`diff`	$2{,}22891418551e-14$	0,0129339694977
	$-1/2$	0,328779511373	0,328779511373	`spl`	0,0	0,00870490074158
	$-1/4$	0,567902536146	0,567902536146	`spl`	0,0	0,00812196731567
16	0	-0	0,0	`spl`	-0	0,00977897644043
	$1/4$	$-0{,}956250428074$	$-0{,}956250428074$	`spl`	-0	0,00809121131897
	$1/2$	$-1{,}06593483416$	$-1{,}06593483416$	`spl`	-0	0,00844788551331
	$-1/2$	0,328779511373	0,328779511373	`diff`	0,0	0,0294342041016
	$-1/4$	0,567902536146	0,567902536146	`diff`	$1{,}95495345409e-16$	0,0290868282318
32	0	-0	$-9{,}36750677027e-17$	`diff`	$9{,}36750677027e-17$	0,0290920734406
	$1/4$	$-0{,}956250428074$	$-0{,}956250428074$	`diff`	$8{,}5915259654e-15$	0,0294840335846
	$1/2$	$-1{,}06593483416$	$-1{,}06593483416$	`diff`	$1{,}42879648583e-12$	0,0296721458435
	$-1/2$	0,328779511373	0,328779511373	`spl`	0,0	0,0141451358795
	$-1/4$	0,567902536146	0,567902536146	`spl`	0,0	0,0163950920105
32	0	-0	0,0	`spl`	-0	0,0136249065399
	$1/4$	$-0{,}956250428074$	$-0{,}956250428074$	`spl`	-0	0,0135118961334
	$1/2$	$-1{,}06593483416$	$-1{,}06593483416$	`spl`	-0	0,0140430927277

3.2. Métodos de Integración Numérica

3.2.1. Estimación de la Función de Error de Gauss

Se nos pide desarrollar una función que calcule el valor estimado de la Función de Error de Gauss para un valor y cualquiera. Todo esto mediante el método de Cuadratura de Gauss utilizando el siguiente formato:

```
>> erf_teo(y, n)
```

Donde `y` $\in \mathbb{R}$ y $4 \leq n \leq 7$.

Para ello se desarrolló el siguiente código:

```
def erf_teo(y,n):
    fix = y/2
    result = 0
    roots, C = roots_weights(n)[0], roots_weights(n)[1]
    func = lambda x: e**(-x**2)

    for i in range(n):
        result += C[i] * func(fix*roots[i] + fix)
    result *= (2/sqrt(pi))*fix

    return result
```

Esta función retorna el valor estimado de la Función de Error de Gauss, calculado mediante Cuadratura de Gauss.

3.2.2. Raices de Polinomio de Legendere

Se nos pide obtener las raices para cada Polinomio de Legendre en el intervalo $4 \leq n \leq 7$. El resultado se puede ver en la siguiente tabla:

n	4
x_1	0,861136311594
x_2	$-0{,}861136311594$
x_3	0,339981043585
x_4	$-0{,}339981043585$

n	5
x_1	0,906179845939
x_2	0,538469310106
x_3	$-0{,}906179845939$
x_4	$-0{,}538469310106$
x_5	$2{,}20823512315e-17$

n	6
x_1	0,932469514203
x_2	0,661209386466
x_3	$-0{,}932469514203$
x_4	$-0{,}661209386466$
x_5	0,238619186083
x_6	$-0{,}238619186083$

n	7
x_1	$-0{,}949107912343$
x_2	$-0{,}741531185599$
x_3	0,949107912343
x_4	0,741531185599
x_5	$-0{,}405845151377$
x_6	0,405845151377
x_7	$2{,}95351022105e-17$

3.2.3. Para $y = 10$ y los Polinomios de Legendre tal que $4 \leq n \leq 7$. Genere una tabla que contenga los c_i y los valores de erf(y) para cada Polinomio de Legendre.

n	4
c_1	0,347854845137
c_2	0,652145154863
c_3	0,652145154863
c_4	0,347854845137
y	1,2119475604

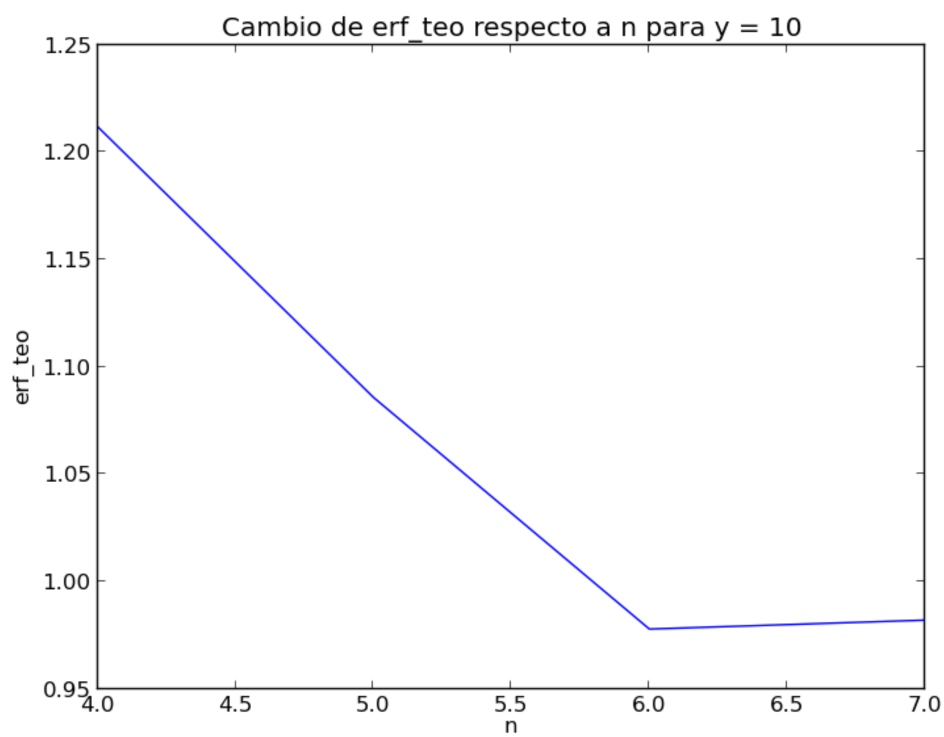
n	5
c_1	0,236926885056
c_2	0,478628670499
c_3	0,568888888889
c_4	0,478628670499
c_5	0,236926885056
y	1,08582368212

n	6
c_1	0,171324492379
c_2	0,360761573048
c_3	0,467913934573
c_4	0,467913934573
c_5	0,360761573048
c_6	0,171324492379
y	0,97790965366

n	7
c_1	0,129484966169
c_2	0,279705391489
c_3	0,381830050505
c_4	0,417959183673
c_5	0,381830050505
c_6	0,279705391489
c_7	0,129484966169
y	0,982074829295

3.2.4. ¿Cómo cambian los valores de la función conforme crece n?

Para ver como cambian los valores a medida que crece n, se puede analizar el siguiente gráfico:



3.2.5. Generación de erf_real()

Se nos pide desarrollar nuevamente una función que calcule la Función de Error de Gauss, pero que esta vez obtenga el valor real de la integración de la función.

El código fuente se puede analizar aquí:

```
def erf_real(y):
    func = lambda x: e**(-x**2)
    return (2/(pi**(0.5)))*quad(func,0,y)[0]
```

3.2.6. Cálculo del Error Relativo

Se nos pide calcular el error relativo para $y = 10$, esto es:

$$\frac{|erf_t(y,n) - erf_r(y,n)|}{erf_r(y)}$$

Lo cual entrega como resultado:

n	**erf_teo**	**erf_real**	**Error Relativo**
4	1,2119475604	1,0	0,211947560402
5	1,08582368212	1,0	0,0858236821154
6	0,97790965366	1,0	0,0220903463402
7	0,982074829295	1,0	0,0179251707053

4. Conclusiones

Podemos concluir que mediante métodos algorítmicos y gracias a la computación moderna, podemos simplificar el cálculo de data compleja, mediante métodos numéricos como las cuadraturas de Gauss, las cuales permiten obtener buenas estimaciones de los resultados reales. Esto es preciadamente útil para la computación ya que la máquina no es inteligente y es de recursos limitados.

Además, métodos como la interpolación polinomial responden adecuadamente a situcaciones donde la data puede ser difusa, y con funciones de naturaleza desconocida, las cuales se simplifican mediante métodos como splines o diferencias divididas, las cuales resultan ser aproximaciones a las curvas reales.