Errores

April 7, 2022

0.1 Errores de truncamiento debido a los cálculos finitos.

El término *error de truncamiento* proviene de la necesidad de truncar expresiones infinitas para obtener números concretos, aunque no exactos.

Un ejemplo de ello es si queremos calcular el seno de un número arbitrario, x. De su definición tenemos:

$$sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$$

Como no podemos hacer esa cuenta en la práctica, tomamos un número aproximado calculando:

$$sin_N(x) = \sum_{n=0}^{N} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$$

donde N será tomado de acuerdo a la precisión que necesitemos tener en el cálculo.

Esas necesidades de presición pueden ser de dos tipos:

1. el *error absoluto* $error_{abs} = |\sin(x) - \sin_N(x)|$
2. el *error relativo* $error_{rel} = \frac{|\sin(x) - \sin_N(x)|}{|\sin(x)|}$

0.1.1 Debido a la finitud de los cálculos en en una computadora aparece siempre un error de truncamiento!

0.1.2 Otros errores:

1. De la teoría (modelado de fenómenos que son idealizaciones)
2. Estocásticos (los números *al azar* no son al azar)
3. De aproximaciones (por ejemplo a funciones no algebraicas, raices, trigonométricas, etc.)
4. Redondeo

0.2 Error de redondeo debido a la finitud de la representación de números.

Está dado por la diferencia (en módulo) del valor real del número y el número más próximo en la representación:

$$error_{abs} = |x - x_{rep}| = |x - rep(x)| \qquad error_{rel} = \frac{|x - x_{rep}|}{|x|} = \frac{|x - rep(x)|}{|x|}$$

1. Cuando trabajamos con números presición fija el error absoluto que cometemos es constante.
2. Cuando trabajamos con números presición flotante el error relativo es constante.

A cualquier número lo podemos escribir como:

$$x = (-1)^s * a_1 a_2 \dots a_t a_{t+1} \dots \beta^e \qquad a_1 \neq 0$$

donde en general la secuencia $\{a_i\}$ is infinita.

Mientras que en una representación flotante solo tenemos:

$$x_{fl} = fl(x) = (-1)^s * \tilde{a}_1 \tilde{a}_2 \dots \tilde{a}_t * \beta^e \qquad t \text{ fijo} \quad L \leq e \leq U$$

Cuál es el número más cercano a x en una representación flotante? La respuesta es simple. Si el número cae entre dos números de la representación, quédese con el menor si está de la mitad para abajo del intervalo (de longitud `eps(x)`), de lo contrario quédese con el mayor.

$$\tilde{a}_i = a_i \quad i = 1 \dots t-1$$

$$\tilde{a}_t = \begin{cases} a_t & si \;\; a_{t+1} < \frac{\beta}{2} \\ a_t + 1 & si \;\; a_{t+1} \geq \frac{\beta}{2} \end{cases}$$

De esta forma nos garantizamos que:

$$|x - fl(x)| \leq \frac{\beta^{e-t+1}}{2} = \frac{eps(x)}{2}$$

y

$$\frac{|x - fl(x)|}{|x|} \leq \frac{\beta^{e-t+1}}{2\beta^e} = \frac{\beta^{1-t}}{2}$$

Therefore,

$$fl(x) = x(1+\delta) \qquad |\delta \leq \frac{eps(1)}{2}$$

0.2.1 Operaciones aritméticas y sus errores

Como hemos visto, existe $ $ tal que

$$fl(x) = x(1+\delta) \qquad |\delta| \leq \frac{eps(x)}{2}$$

Idealmente nos gustaría que:

$$x \; op_{FL} \; y := fl(x \; op \; y)(1+\delta)$$

Pero eso no es lo que hace la máquina. Las operaciones se hacen en la representación:

0.2.2 Atención: una operación aritmética puede que tenga un error mucho mayor!

Consideremos $\mathcal{F}(10, 2, L, U)$ y restemos $1.0 - 0.99 = 0.01$

$$
\begin{aligned}
& 0.10 * 10^1 && (1)\\
- & && (2)\\
& 0.09 * 10^1 && (3)\\
& \text{———————} && (4)\\
& 0.01 * 10^1 = 0.1 && (5)
\end{aligned}
$$

un error absoluto muy grande pero esperable, pero el error relativo es muy grande! En este caso el error relativo es 10!

Para este caso $eps(1) = 0.01 * 10^1 = 0.1$ mientras que $eps(0.9) = 0.01 * 10^0 = 0.01$

Por lo tanto si dos números son similares el error relativo puede ser muy grande. Para aliviarlo se puede aumentar provisoriamente t (o sea disminuir eps(1)).

Si hacemos el ejemplo anterior con $t = 3$ obtenemos:

$$
\begin{aligned}
& 0.100 * 10^1 && (6)\\
- & && (7)\\
& 0.099 * 10^1 && (8)\\
& \text{———————} && (9)\\
& 0.001 * 10^1 && (10)
\end{aligned}
$$

Las computadoras modernas usan internamente (en el co-procesador matemático) mantizas mucho más grandes (*dígitos de guarda*). Típicamente de 80 o más bits.

Eso nos garantiza que

$$x\ op_{FL}\ y = fl(x\ op\ y)(1 + \delta)$$

0.2.3 Las operaciones no son asociativas!

Ejercicio: Con $a = 1234.567$, $b = 45.67844$, $c = 0.0004$ y utilizando aritmética *decimal* de punto flotante de 7 dígitos, calcule:

$$(a + b) + c$$

y

$$a + (b + c).$$

Por ejemplo:

$$a = 0.1234567 * 10^4 \tag{11}$$
$$b = 0.0045678 * 10^4 \tag{12}$$
$$\text{---------------------} \tag{13}$$
$$a + b = 0.1280245 * 10^4 \tag{14}$$

[16]:
```
1.0 - (1.0 - eps(0.1))
```

[16]:
```
0.0
```

[17]:
```
(1.0 - 1.0) + eps(0.1)
```

[17]:
```
1.3877787807814457e-17
```

0.2.4 Propagación de errores

Ahora vamos a tratar el caso real, es decir:

$$x\ op_{fl}\ y = fl(fl(x)\ op\ fl(y))$$

O sea nos dan dos números reales, los pasamos a una representación flotante y luego hacemos la operación como flotante. Además del error ya tratado ahora tenemos un error por haber asignado a cada número real un número (distinto en general en la representación).

Tenemos así que, por ejemplo:

$$x +_{fl} y = (x(1+\delta_x) + y(1+\delta_y))(1+\delta) \quad |\delta|,\ |\delta_x|,\ |\delta_y| \leq \frac{eps(1)}{2} := u$$

El error cometido se puede estimar entonces como:

$$|x +_{fl} y - (x+y)| = |x\delta_x + y\delta_y + (x+y)\delta + (x\delta_x + y\delta_y)\delta| \tag{15}$$
$$\leq |x||\delta_x| + |y||\delta_y| + |x+y||\delta| + (|x||\delta_x| + |y||\delta_y|)|\delta| \tag{16}$$
$$\leq (|x| + |y|)(1+u)u + |x+y|u \tag{17}$$

El error relativo es entonces:

$$\frac{|x +_{fl} y - (x+y)|}{|x+y|} \leq \frac{|x|+|y|}{|x+y|}(1+u)u + u \tag{18}$$

Si $y \approx -x$ el error relativo puede ser muy grande y es debido al error de redondeo inicial en x e y.

0.2.5 Evitar:

1. resta de dos números parecidos
2. suma/resta de números muy dispares
3. cociente de números muy grandes o muy chicos (overflow o underflow).

0.2.6 Ejemplos:

Ejemplo 1:

$$\sqrt{x+h}-\sqrt{x} = \frac{(\sqrt{x+h}-\sqrt{x})(\sqrt{x+h}+\sqrt{x})}{\sqrt{x+h}+\sqrt{x}} \tag{19}$$

$$= \frac{(\sqrt{x+h})^2-(\sqrt{x})^2}{\sqrt{x+h}+\sqrt{x}} \tag{20}$$

$$= \frac{(x+h-x}{\sqrt{x+h}+\sqrt{x}} \tag{21}$$

$$= \frac{h}{\sqrt{x+h}+\sqrt{x}} \tag{22}$$

Ejemplo 2: Raices de Ax^2+Bx+C cuando $B^2 >> |4AC|$

$$r_{\pm} = \frac{-B \pm \sqrt{B^2-4AC}}{2A}$$

Si $B>0$ la raiz positiva tendrá problemas de presición. Si $B<0$ la negativa los tendrá. Supongamos el primer caso:

$$r_{+} = \frac{-B+\sqrt{B^2-4AC}}{2A} \tag{23}$$

$$= \frac{(-B+\sqrt{B^2-4AC})(-B-\sqrt{B^2-4AC})}{2A(-B-\sqrt{B^2-4AC})} \tag{24}$$

$$= \frac{(-B^2+(\sqrt{B^2-4AC})^2)}{2A(B+\sqrt{B^2-4AC})} \tag{25}$$

$$= \frac{-2C}{B+\sqrt{B^2-4AC}} \tag{26}$$

[29]:
```
A = 1.0
B = 10.0
C = 10^(-12)

println("modo usual raiz positiva = $((-B + sqrt(B^2 - 4*A*C))/2/A)")
println("modo seguro raiz positiva = $(-2*C/(+B + sqrt(B^2 - 4*A*C)))")
```

```
modo usual raiz positiva = -9.947598300641403e-14
modo seguro raiz positiva = -1.0000000000000105e-13
```

Ejemplo 3: (ver en el otro notebook)

0.2.7 Que precisión combiene usar?

Hasta hace unos años la respuesta era que lo mejor era usar doble precisión (64 bits):

1. Nos asegurábamos más precisión.
2. No incurríamos en demasiados gastos de tiempo de cómputo pues los co-procesadores tenían más dígitos (de guarda) en cualquier caso.
3. Los cálculos estaban acotados por el procesamiento y no por el acceso a memoria.

Hoy no es tan claro:

1. Las GPU tienen co-procesadores más simples y en general son mejores para hacer cálculos en precisión simple.
2. Los cores de las CPU comparten co-procesadores.
3. La vectorización (muchas cuentas simultáneas en un solo ciclo) se beneficia de mantizas más cortas.
4. Los cálculos hoy día están limitados por el acceso a memoria. Mantizas más cortas significa mayor cantidad de datos en la memoria rápida.

Estrategia:

1. Haga su código de tal manera que pueda funcionar con ambas precisiones y compare resultados.
2. Siempre tenga cuidado en no originar acumulación de errores innecesarias.

0.2.8 Otras representaciones:

1. BigFloat (precisión arbitraria)
2. Rationals (Int/Int)
3. Irrationals (cálculo de por-ejemplo π de acuerdo a la precisión utilizada)

[31]:
```
1//2 + 5//4
```

[31]:
```
7//4
```

[32]:
```
BigFloat()
```

[32]:
```
3.14159265358979323846264338327950288419716939937510582097494459230781640628619 8
```

[ ]:
```

```