

Optimización

On this page

- Parte 1: temas del cómputo Científico y del Álgebra Lineal Numérica.
- Parte 2: ecuaciones no lineales y Optimización.

Dar click en [liga](#) para el temario.

Url del repositorio: <https://github.com/ITAM-DS/analisis-numerico-computo-cientifico>

Liga para crear *issues* en repositorio: [liga para issues](#)

Nota generada a partir de [liga](#)

1.1 Análisis numérico y cómputo científico

Introducción

El cómputo científico (ver [liga](#)) es parte de lo que se conoce como ciencia de la computación, cuya investigación incluye las fases:

1. Desarrollo de un modelo matemático para un fenómeno de interés.
2. Desarrollo de un algoritmo.
3. Implementación del algoritmo en un software.
4. Simulaciones numéricas del fenómeno con el software.
5. Representación de los resultados calculados (gráficas o herramientas visuales).
6. Interpretación y validación de los resultados.

y se realizan los puntos anteriores en una forma repetida (1-6) dependiendo de los resultados obtenidos en 6.

El cómputo científico se encarga de las fases 2 a 4: desarrollo, implementación y uso de los algoritmos numéricos y software para aplicaciones. Ejemplos de aplicaciones las encontramos en análisis y ajuste de modelos a datos, optimización y machine learning, entre otras.

Análisis numérico y cómputo científico.

Nuestro curso involucra temas del análisis numérico el cual se enfoca al diseño y análisis de algoritmos para resolver problemas que surgen en la ciencia de la computación e ingeniería. Por esto, al análisis numérico se le ha llamado también cómputo científico y se distingue de otras áreas de la ciencia de la computación en que trabaja con **cantidades continuas**, cantidades como velocidad y temperatura presentes naturalmente son algunos ejemplos.

Tales cantidades las encontramos en muchos problemas de matemáticas los cuales no se resuelven en principio con un número finito de pasos, sino que deben resolverse por un proceso iterativo que converja a la solución, teóricamente infinito. Sin embargo, en la práctica, no se realizan procesos infinitos y además no se resuelven problemas de manera exacta; se obtienen respuestas aproximadamente correctas, "suficientemente cercanas o precisas" al resultado deseado. Así, el análisis numérico debe trabajar con recursos finitos y cantidades discretas.

Lo anterior subraya uno de los aspectos más importantes del cómputo científico: encontrar algoritmos iterativos que **converjan** "rápidamente", así como dar una estimación de la **exactitud** de las aproximaciones calculadas. Así, un segundo tema que distingue al análisis numérico de otras áreas de la ciencia de la computación es el relacionado con las **aproximaciones** y sus efectos. Es indispensable que tales aproximaciones sean controladas por algoritmos "buenos", esto es, que sean **eficientes** (hacer más con menos), sean **confiables** y **exactos** ante la serie de aproximaciones realizadas: sean **estables**.

La eficiencia se relaciona directamente con el **costo computacional**, el cual típicamente se mide a partir del número de operaciones que realiza el algoritmo (aunque otra componente común para medir el costo computacional, es la transferencia de datos entre las diferentes jerarquías de memoria en una máquina) y la **estabilidad** se refiere a no amplificar los **errores** generados por las aproximaciones o resultados calculados durante la ejecución del algoritmo.

Asimismo, la estabilidad se relaciona con la pregunta: ¿si se perturban los datos de entrada, el algoritmo calcula una solución “cercana” a los datos no perturbados? (esta pregunta tiene que ver con el **análisis de sensibilidad** en un algoritmo ante perturbaciones en los datos de entrada).

Fuentes de error

Algunas fuentes de error que nos encontramos al resolver un problema surgen por:

- Uso de modelos (simplificación u omisión).
- Mediciones con instrumentos.
- Cálculos previos.

Las fuentes anteriores típicamente están fuera de nuestro control. En el cómputo científico se estudian dos fuentes que influyen directamente en el algoritmo utilizado para resolver un problema y sí podemos controlarlas:

- Truncamiento: relacionado con el uso de procesos o cantidades finitas.
- Redondeo: relacionado con la representación de los números y la aritmética realizada en una máquina.

Estas fuentes de error influyen en la exactitud de un cálculo y las perturbaciones que resulten de estas fuentes, serán amplificadas (o no) por la **naturaleza del problema** y el **tipo de algoritmo utilizado**. Al estudio de la exactitud y estabilidad de un algoritmo por fuentes de error se le llama **análisis del error**.

Análisis del error

En general un problema puede verse como evaluar una función $f : \mathbb{R} \rightarrow \mathbb{R}$. Por ejemplo, sea x el dato de entrada, considérese el problema de evaluar $f(x)$, el dato de salida.

Al trabajar con la máquina, nos encontramos que la representación de un número real en general no es exacta, por ello tenemos una aproximación \hat{x} a la cantidad x (error por redondeo) y nuestra forma de calcular f es por medio de un algoritmo \hat{f} , entonces, el error al evaluar f es:

$$\text{Error total} = \hat{f}(\hat{x}) - f(x) = \underbrace{\hat{f}(\hat{x}) - f(\hat{x})}_{\text{Error Computacional}} + \underbrace{f(\hat{x}) - f(x)}_{\text{Error en datos}}.$$

El primer término es generado por el algoritmo utilizado: \hat{f} (cálculos y aproximaciones realizadas por el algoritmo utilizando \hat{x}) y el segundo es debido a mediciones o redondeos realizados por la máquina o personas o instrumentos, observa que este término no se ve influenciado por el algoritmo utilizado y es un error que no controlamos.

El error computacional tiene dos componentes: uno por **truncamiento** y otro por **redondeo**:

- El truncamiento es la diferencia entre el resultado verdadero y el resultado que se obtiene con un algoritmo usando aritmética exacta. Por ejemplo, reemplazar una serie infinita (resultado verdadero) con una serie truncada (algoritmo: truncar serie infinita) y evaluar la serie truncada con **aritmética exacta** (por ejemplo usar $\frac{1}{3}$ en los cálculos en lugar de 0.333).
- El error por redondeo es la diferencia entre el resultado obtenido por un algoritmo utilizando aritmética exacta y el resultado producido por el mismo algoritmo usando **aritmética de máquina**.

💡 Observación

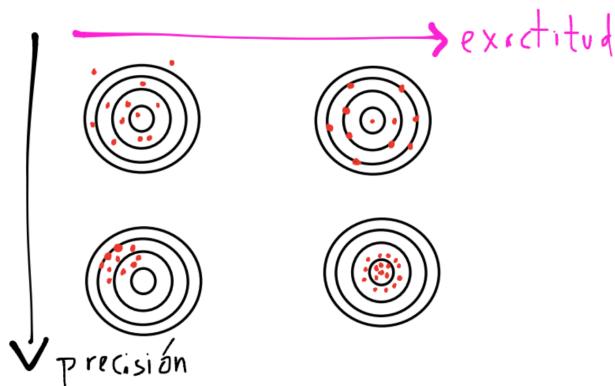
Obsérvese que en la descomposición del error total de arriba se utiliza en el error computacional \hat{x} . En la nota [Condición de un problema y estabilidad de un algoritmo](#) se define la estabilidad de un algoritmo utilizando x en lugar de \hat{x} pues se desea medir la cercanía entre x y \hat{x} determinada por el sistema de punto flotante (errores por redondeo).

ℹ️ Comentario

Aunque los errores por truncamiento y por redondeo son componentes del error total, típicamente un tipo de error domina sobre otro. Hacer la distinción entre ellos, nos ayuda a entender el comportamiento de los algoritmos y los factores que influyen en su **exactitud**.

Nota sobre exactitud y precisión.

Los errores en los cálculos y las medidas se pueden caracterizar respecto a su exactitud y precisión. La exactitud se refiere a que tan cercano está el valor calculado o medido del valor verdadero. La precisión se refiere a qué tan cercanos se encuentran unos de otros diversos valores calculados o medidos. Obsérvese el siguiente diagrama:



Preguntas de comprensión

- 1) ¿Cuáles son las características del análisis numérico y cómputo científico?
- 2) ¿Qué propiedades debe tener un buen algoritmo?
- 3) ¿Cuáles fuentes de error principalmente son estudiadas por el análisis numérico y cómputo científico al resolver un problema?
- 4) Menciona las diferencias entre los términos exactitud y precisión.

Referencias:

1. M. T. Heath, Scientific Computing. An Introductory Survey, McGraw-Hill, 2002.

Notas para contenedor de docker:

Comando de docker para ejecución de la nota de forma local:

nota: cambiar <ruta a mi directorio> por la ruta de directorio que se desea mapear a /datos dentro del contenedor de docker.

```
docker run --rm -v <ruta a mi directorio>:/datos --name jupyterlab_optimizacion -p 8888:8888 -d palmoreck/jupyterlab_optimizacion:2.1.4
```

password para jupyterlab: `qwerty`

Detener el contenedor de docker:

```
docker stop jupyterlab_optimizacion
```

Documentación de la imagen de docker `palmoreck/jupyterlab_optimizacion:2.1.4` en [liga](#).

Nota generada a partir de [liga](#)

1.2 Sistema de punto flotante

Representación de los números en la computadora

Las computadoras utilizan una determinada cantidad de cifras de un número real para realizar operaciones. Además, utilizan una representación de los números en bases no usadas por las personas para realizar cálculos comunes, ejemplos son la **2, 8 o 16**. En contraste, la mayoría de las personas utilizamos la base **10** para representar a los números y realizar cálculos.

A continuación se muestran construcciones que se han hecho para representar los números en una computadora.

Enteros

Tenemos distintos métodos para la representación de los enteros en una computadora, pero uno que es más utilizado es el de "magnitud con signo" en el que se utiliza un bit para el signo del número y los bits restantes para almacenar al número. El primer bit se le da el valor de 0 para codificar al signo + y el valor 1 codifica al bit -. Entonces el número -173 se almacena con la cadena de 16 bits:

1000000010101101

Utilizamos la notación posicional para convertir este número binario a base 10: el primer bit es 1 por lo que se tiene un signo negativo. Luego:

$$\begin{aligned} 0 * 2^{14} + 0 * 2^{13} + \dots + 1 * 2^7 + 0 * 2^6 + 1 * 2^5 + 0 * 2^4 = \\ + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 \\ 1 * 2^7 + 1 * 2^5 + 1 * 2^3 + 1 * 2^2 + 1 * 2^0 = \\ 173 \end{aligned}$$

```
/*%cflags:-lm
#include<stdio.h>
#include<math.h>
int main(){
    printf("suma: %f",pow(2,7)+pow(2,5)+pow(2,3)+pow(2,2)+pow(2,0));
    return 0;
}
```

suma: 173.000000

Ejercicio

Determina el rango de enteros de base 10 que puede representarse en una computadora de 16 bits utilizando el primer bit para el signo

Reales

Dadas las limitaciones en el almacenamiento de una computadora (hardware), sólo se representa un subconjunto de los números reales en ella, tal conjunto se denota como $\mathcal{F}l$ y contiene números racionales:

$$\mathcal{F}l \subset \mathbb{Q} \subset \mathbb{R}.$$

Sistema de punto flotante (SPF)

En un sistema de punto flotante se define:

1. Rango de un exponente definido por un límite inferior y uno superior.
2. Base del sistema.
3. Precisión.

Así, un número x en el SPF, $x \in \mathcal{F}l$, se representa de la forma:

$$\pm 0.d_1 d_2 \dots d_k \times \beta^n.$$

donde:

n es el exponente, $n \in [L, U] \cap \mathbb{Z}$ con L, U fijos.

k es la precisión.

β es la base.

$d_i \in \{0, 1, \dots, \beta - 1\} \forall i = 1, \dots, k$ son los dígitos.

Un poco de historia...

En 1985 la IEEE publicó un estándar de nombre [Binary Floating Point Arithmetic Standard 754-1985](#) y ha habido más estándares publicados, siendo el más reciente el [IEEE 754-2019](#). El estándar 754 – 1985 proveía estándares para números de punto flotante binarios y decimales, formatos para intercambio de datos, algoritmos para operaciones de redondeo y manejo de excepciones. Los formatos se especificaron para precisiones simple, doble y extendida y tales estándares son seguidos por las manufactureras de computadoras que utilizan el hardware de punto flotante: "...A family of commercially feasible ways for new systems to perform binary floating-point arithmetic is defined. This standard specifies basic and extended floating-point number formats; add, subtract, multiply, divide, square root, remainder, and compare operations; conversions between integer and floating-point formats; conversions between different floating-point formats; conversions between basic-format floating-point numbers and decimal strings; and floating-point exceptions and their handling, including nonnumbers..." (estándar 754 – 1985)

Definición

A la parte $\pm 0.d_1 d_2 \dots d_k$ se le llama **mantisa**. A la porción $d_2 \dots d_k$ se le llama **fracción** f .

Los números reales que tienen una representación exacta en el $\mathcal{F}l$ se les conoce como **números de máquina**.

Ejemplo:

Supóngase un $\mathcal{F}l$ con $\beta = 10, k = 4, n \in [-4, 3] \cap \mathbb{Z}$ entonces:

$$0.3330 \times 10^{-1}, 0.3300 \times 10^3 \in \mathcal{FL}$$

pero: $\frac{1}{3} \notin \mathcal{F}l$

por lo que $0.333 \times 10^{-1}, 0.3300 \times 10^3$ son números de máquina.

SPFN

Un Sistema de Punto Flotante Normalizado es aquel que cumple:

$$d_1 \in \{1, 2, \dots, \beta - 1\}$$

para números distintos de cero.

💡 Observación

El número cero es el único que tiene dígitos de la mantisa y exponente iguales a cero.

Ejemplos:

1. $\beta = 10, k = 3$, rango del exponente en $[-3, 3] \cap \mathbb{Z}$, entonces algunos números en el SPFN:

Notación de punto flotante Mantisa Exponente Valor de punto fijo

0.153×10^0	0.153	0	0.153
-0.990×10^2	-0.990	2	-99.0
0.343×10^{-3}	0.343	-3	0.000343

1. En un SPFN con $\beta = 2, k = 3$, rango del exponente en $[0, 2] \cap \mathbb{Z}$ se tiene:

a) El número más grande positivo que es posible representar es:

$$0.111 \times 2^2 = (11.1)_2 = 1 * 2^1 + 1 * 2^0 + 1 * 2^{-1} = (3.5)_{10}.$$

b) El más chico positivo es:

$$0.100 \times 2^0 = (0.1)_2 = 1 * 2^{-1} = (0.5)_{10}.$$

💡 Observación 1

Se utiliza $(\cdot)_{10}$ para representar un número en base 10 pero típicamente se omite escribir paréntesis y la base.

💡 Observación 2

La representación normalizada permite una representación única de los números reales en el $\mathcal{F}l$:

el número 0.343×10^{-3} como se vio en el ejemplo 1) está en $\mathcal{F}l$ pero ni 0.0343×10^{-2} ni 3.43×10^{-4} están en el $\mathcal{F}l$.

💡 Observación 3

Por la observación 2 en un SPFN con $\beta = 2$ no es necesario almacenar el primer bit pues siempre es 1 por lo que sólo se almacenará la parte fraccionaria de la forma $1.f$ (más sobre esto en la siguiente sección).

Números de máquina binarios

Un SPF de doble precisión utiliza:

- 64 bits para representar un número real. El primero es un indicador de signo denotado por s . Le siguen 11 bits para construir al exponente n y 52 bits que construyen a la parte fraccionaria f de la mantisa. La base es $\beta = 2$.

💡 Nota 1

Si es SPFN entonces aunque se tienen 52 bits, éstos se utilizan para almacenar un dígito más del número (ver observación 3 anterior). Así, en un SPF en general maneja 52 (o 53) dígitos binarios, que corresponden a aproximadamente 15 (o 16) dígitos decimales. Ver: [Double-precision floating-point format](#).

```
//%cflags:-lm
#include<stdio.h>
#include<math.h>
int main(){
    printf("52 bits corresponden aproximadamente a: %f dígitos decimales\n",
    log10(pow(2,52)));
    printf("53 bits corresponden aproximadamente a: %f dígitos decimales\n",
    log10(pow(2,53)));
    return 0;
}
```

52 bits corresponden aproximadamente a: 15.653560 dígitos decimales
53 bits corresponden aproximadamente a: 15.954590 dígitos decimales

💡 Nota 2

Los 11 bits que se utilizan para construir al exponente producen un rango de 0 a $2^{11} - 1 = 2047$:

```
//%cflags:-lm
#include<stdio.h>
#include<math.h>
int main(){
    int sum = 0;
    int n=10;
    int i;
    for(i=0;i<=n;i++)
        sum+=pow(2,i);
    printf("Suma de 2^0 + 2^1 + ... + 2^10: %d", sum);
    return 0;
}
```

Suma de $2^0 + 2^1 + \dots + 2^{10} = 2047$

💡 Continuación Nota 2

pero esto construye un exponente con signo positivo, por lo que se resta (*offset*) la cantidad de -1023 y se tiene el rango del exponente en $[-1023, 1024] \cap \mathbb{Z}$:

$$2^{n-1023}.$$

Y un valor de $n = 1023$ representa un exponente de 0.

💡 Nota 3

Los valores -1023 (todos los bits iguales a 0) y 1024 (todos los bits iguales a 1) para el exponente se reservan para números especiales. Por esto, el exponente corre en el rango de $[-1022, 1023] \cap \mathbb{Z}$.

💡 Nota 4

Todos los bits iguales a 0 para el exponente se utiliza para representar al número 0 con signo (si su fracción f es 0) y a los números subnormales (si su fracción f es $\neq 0$). Todos los bits iguales a 1 se utiliza para representar ∞ (si su fracción f es 0) y NaNs (si su fracción $f \neq 0$).

Entonces un número de máquina binario en un SPFN se representa por:

$$(-1)^s 2^{n-1023} (1 + f)$$

Y los subnormales como:

$$(-1)^s 2^{1-1023} (0 + f) = (-1)^s 2^{-1022} f.$$

💡 Observación

Los números subnormales son todos aquellos números con magnitud menor al número más chico positivo normalizado. Con estos números se sacrifica precisión por representatividad alrededor del 0. Para representar a estos números d_1 no es igual a 1.

💡 Observación

Hay dos representaciones para el 0, una con signo positivo ($s = 0, n = 0, f = 0$) y otra con signo negativo ($s = 1, n = 0, f = 0$).

Ejemplos:

1. El valor del exponente más chico para los números normales es: $2^{1-1023} = 2^{-1022}$.

2. El valor del exponente más grande es: $2^{2046-1023} = 2^{1023}$.

1. Considérese el número formado por:

primer bit: 0.

bits del exponente: 10000000011.

bits de la mantisa: 1011100100010 ... 0.

Entonces:

1. El número es positivo pues $s = 0$: $(-1)^s = 1$.

2. Los bits del exponente generan al número decimal:

$$n = 1 * 2^{10} + 0 * 2^9 + \dots + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 1024 \\ + 2 + 1 = 1027$$

por lo que el exponente es 4: $2^{1027-1023} = 2^4$.

1. Los bits de la mantisa generan al número decimal:

$$1 * 2^{-1} + 1 * 2^{-3} + 1 * 2^{-4} + 1 * 2^{-5} + 1 * 2^{-8} + 1 * 2^{-12} \\ = 0.7229.$$

```
//%cflags:-lm
#include<stdio.h>
#include<math.h>
int main(){
    printf("Suma de 2^-1 + 2^-3 + 2^-4+2^-5+2^-8+2^-12: %f",
    pow(2,-1)+pow(2,-3)+pow(2,-4)+pow(2,-5)+pow(2,-8)+pow(2,-12));
    return 0;
}
```

Suma de 2^-1 + 2^-3 + 2^-4+2^-5+2^-8+2^-12: 0.722900

Entonces el número de máquina binario 0 10000000011 1011100100010 ... 0 es el número decimal:

$$(-1)^s 2^{n-1023} (1 + f) = (-1)^0 2^4 (1 + 0.7229) = 27.5664$$

```
//%cflags:-lm
#include<stdio.h>
#include<math.h>
int main(){
    printf("2^4(1.7229): %f", pow(2,4)*1.7229);
    return 0;
}
```

2^4(1.7229): 27.566400

Valores interesantes en un SPFN de doble precisión

El número de máquina más grande positivo normalizado es:

$$2^{1023}(1 + (2^{-1} + \dots + 2^{-52})) = 2^{1023}(1 + (1 - 2^{-52})) \\ = 2^{1023}(2 - 2^{-52}) \approx 1.7977 \times 10^{308}.$$

```
//%cflags:-lm
#include<stdio.h>
#include<float.h>
int main(){
    printf("Número más grande positivo: %e\n", DBL_MAX);
    return 0;
}
```

Número más grande positivo: 1.797693e+308

Números que rebasan este límite superior resultan en un **overflow**:

```
//%cflags:-lm
#include<stdio.h>
#include<float.h>
#include<math.h>
int main(){
    printf("overflow: %e\n", DBL_MAX + 0.0000000000000001*pow(10,308));
    return 0;
}
```

overflow: inf

i Ejercicio

Da más ejemplos de ejecuciones como la anterior que resulten en overflow.

El número de máquina **normalizado** más pequeño positivo es:

$$2^{-1022}(1 + 0) \approx 2.2251 \times 10^{-308}.$$

```
#include<stdio.h>
#include<float.h>
int main(){
    printf("Número normalizado más chico positivo: %e\n", DBL_MIN);
    return 0;
}
```

Número normalizado más chico positivo: 2.225074e-308

El número de máquina no normalizado o **subnormalizado** más chico positivo es del orden de

$$2^{-1022}(2^{-52}) = 2^{-1074} \approx 10^{-324}.$$

```
//%cflags:-lm
#include<stdio.h>
#include<float.h>
#include<math.h>
int main(){
    printf("Número más chico positivo: %e\n", pow(2,-52)*DBL_MIN);
    return 0;
}
```

Número más chico positivo: 4.940656e-324

Números con magnitud más chica que el valor anterior resultan en un **underflow**:

i Ejercicio

Da más ejemplos de ejecuciones como la anterior que resulten en underflow.

```
//%cflags:-lm
#include<stdio.h>
#include<float.h>
#include<math.h>
int main(){
    printf("Underflow: %e\n", (1-.5)*pow(2,-52)*DBL_MIN);
    return 0;
}
```

Underflow: 0.000000e+00

Epsilon de la máquina ϵ_{mag}

```
//%cflags:-lm
#include<stdio.h>
#include<float.h>
#include<math.h>
int main(){
    printf("Epsilon de la máquina equivale aproximadamente a  $2^{(-53)}$ : %e\n",
DBL_EPSILON/2);
    return 0;
}
```

Epsilon de la máquina equivale aproximadamente a $2^{(-53)}$: 1.110223e-16

El valor anterior representa el máximo error relativo en la representación de un número real en su número de máquina. Como se observa en la ejecución anterior $\epsilon_{mag} \approx 2^{-53} \approx 1.11 \times 10^{-16}$ por lo que tenemos alrededor de 15 o 16 dígitos de precisión para un número real en el SPFN de doble precisión.

Definición

Si $aprox$ es mi cantidad con la que aproximo a mi objetivo obj entonces el error absoluto de $aprox$ y el error relativo de $aprox$ es:

$$ErrAbs(aprox) = |aprox - obj|.$$
$$ErrRel(aprox) = \frac{ErrAbs(aprox)}{|obj|}.$$

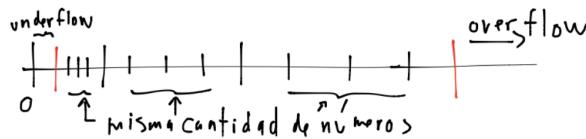
Si $ErrRel(aprox) \approx 10^{-k}$ se dice que $aprox$ aproxima a obj con alrededor de k dígitos correctos.

Observación

Otras definiciones para ϵ_{mag} son:

- ϵ_{mag} es el número más chico positivo tal que: $1 + \epsilon_{mag} \neq 1$ a precisión de la máquina.
- ϵ_{mag} es la distancia del número 1 al siguiente número de máquina.

Aunque las tres definiciones pueden diferir ligeramente, las tres pretenden dar la medida de granularidad de un SPF. Una representación gráfica para números positivos de un SPF es la siguiente:



Se tienen las siguientes afirmaciones:

- El intervalo $[1, 2]$ en un SPFN de doble precisión está formado por la secuencia de números de máquina: $1, 1 + 2^{-52}, 1 + 2 \times 2^{-52}, 1 + 3 \times 2^{-52}, \dots, 2$.
- El intervalo $[2, 4]$ en tal sistema está formado por: $2, 2 + 2^{-51}, 2 + 2 \times 2^{-51}, 2 + 3 \times 2^{-51}, \dots, 4$.

Por lo que el intervalo $[2^j, 2^{j+1}]$ se obtiene multiplicando 2^j veces la secuencia en $[1, 2]$ y los huecos entre un número de máquina y otro número de máquina no son en términos relativos más grandes que $2^{-52} \approx 2.22 \times 10^{-16}$.

Asimismo, el intervalo $[\frac{1}{2}, 1]$ en tal sistema está formado por: $\frac{1}{2}, \frac{1}{2} + 2^{-53}, \frac{1}{2} + 2 \times 2^{-53}, \frac{1}{2} + 3 \times 2^{-53}, \dots, 1$.

Entonces el espaciado entre cada número de máquina en el intervalo $[2^j, 2^{j+1}]$ siempre es menor o igual a $2^{-53} = \epsilon_{mag}$.

Una forma computacional de obtener al ϵ_{mag} de forma sencilla es con el cálculo:

```
#include<stdio.h>
#include<float.h>
int main(){
    printf("Epsilon de la máquina: %e\n", 1-3.0*(4/3.0-1));
    return 0;
}
```

Epsilon de la máquina: 2.220446e-16

Pregunta

¿Por qué funciona esto? tip: imprimase el cálculo $3.0 * (4/3.0 - 1)$ con el especificador de formato de la función `printf %.16f` o bien `%.15e`.

Reglas de corte y redondeo

Al ver el diagrama anterior de la representación gráfica de un SPF se observa que existen huecos entre cada número de máquina. Lo anterior implica que al ingresar un número real x en la computadora, ésta realiza una aproximación a x que se encuentre en el SPF. Esta aproximación genera errores conocidos con el nombre de **errores por redondeo**.

Entre las reglas que una computadora realiza para dar las aproximaciones a un número $x \in \mathbb{R}$ están: la regla de corte y la de redondeo y se pueden representar con funciones matemáticas:

Regla de corte: sea $f_{lc} : \mathbb{R} \rightarrow \mathcal{F}l$ una función cuya regla de correspondencia es: $x \in \mathbb{R}$ con x en el rango de valores del SPF, entonces: $x = \pm 0.d_1d_2 \dots d_kd_{k+1} \dots \times \beta^n$ y la regla de corte a k dígitos es: $f_{lc}(x) = \pm 0.d_1d_2 \dots d_k$.

Regla de redondeo: sean $\beta = 10$ y $f_{lr} : \mathbb{R} \rightarrow \mathcal{F}l$ una función cuya regla de correspondencia es: $x \in \mathbb{R}$ con x en el rango de valores del SPF, entonces: $x = \pm 0.d_1d_2 \dots d_kd_{k+1} \dots \times \beta^n$ y la regla de redondeo es:

$$f_r(x) = \begin{cases} \text{sumar uno a } d_k & \text{si } d_{k+1} \geq 5, \\ f_c(x) & \text{en otro caso} \end{cases}$$

Observación

También la regla $f_{lr}(\cdot)$ se define como antes pero se añade la restricción entre si es par o impar el último dígito en caso de empate, entonces se almacena el par

Ejemplos:

1.

$\pi = 3.141592 \dots = 0.3141592 \dots \times 10^1$. Si un SPFN usa $k = 5$ entonces:

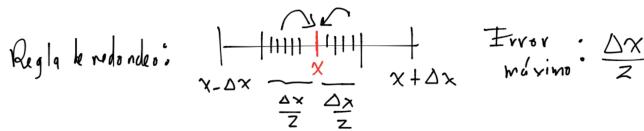
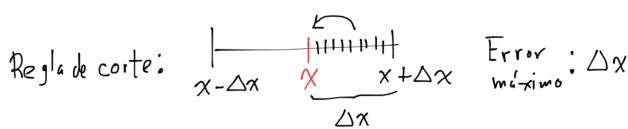
$$\begin{aligned} f_{lc}(\pi) &= 0.31415, \\ f_{lr}(\pi) &= 0.31416. \end{aligned}$$

1. Supóngase que $f_{lr}(\cdot)$ hace diferencia entre el último dígito almacenado es par e impar. Entonces para $k = 2$:

Número: x Corte: $f_{lc}(x)$ Redondeo: $f_{lr}(x)$

0.1649	0.16	0.16
0.1650	0.16	0.16
0.1651	0.16	0.17
0.1749	0.17	0.17
0.1750	0.17	0.18
0.1751	0.17	0.18

Por los ejemplos anteriores se puede comprobar que la regla de redondeo tiene más exactitud que la de corte:



i Definición

Otra expresión utilizada para la regla $fI(\cdot)$ es a partir de la definición de error relativo:

$aprox = obj(1 + ErrRel(aprox))$ (si $aprox \geq obj$ y $obj > 0$):

$$\forall x \in \mathbb{R} \exists \epsilon \in \mathbb{R} \text{ con } |\epsilon| \leq \epsilon_{mag} \text{ tal que: .}$$

$$fI(x) = x(1 + \epsilon), |\epsilon| \leq \epsilon_{mag}.$$

cuya interpretación es: la diferencia entre x con $fI(x)$ es un término (en magnitud) de a lo más ϵ_{mag} relativo a $|x|$.

i Nota

Un resultado que se puede verificar es:

$$\epsilon_{mag} = \begin{cases} \beta^{1-k} & \text{si se utiliza regla de corte,} \\ \frac{1}{2}\beta^{1-k} & \text{si se utiliza la regla de redondeo} \end{cases}$$

Ejemplo:

Considérese un SPFN con $\beta = 2, k = 3$ entonces si se utiliza la regla de corte:

$$\epsilon_{mag} = \beta^{1-k} = \beta^{-2} = 2^{-2} = 0.25.$$

si se utiliza la regla de redondeo:

$$\epsilon_{mag} = \frac{1}{2}\beta^{1-k} = \frac{1}{2}\beta^{-2} = \frac{1}{2}2^{-2} = 0.125.$$

Aritmética de máquina.

Además de representar números reales en la máquina otro objetivo es el de realizar operaciones entre ellos. Si la representación de números tiene un error asociado (error por redondeo) entonces es natural pensar que las operaciones aritméticas también tendrán errores por redondeo. Las razones de los errores son nuevamente el uso de precisión finita y la conversión entre bases: al ingresar números a la computadora se convierte a base 2 (por ejemplo), se realizan operaciones y el resultado se presenta en base 10.

Para analizar los errores por redondeo que se presentan en la aritmética de máquina es suficiente considerar la base 10, no conversiones entre bases (por ejemplo de la base 2 a la base 10) y utilizar los siguientes operadores y suposiciones:

Supongamos que se tiene un SPFN y $fI(\cdot)$ regla de corte o redondeo a una precisión k dada. Sean $a, b \in \mathbb{R}$, se definen los siguientes operadores en el SPFN:

$$\begin{aligned} \oplus : \mathbb{R}^2 &\rightarrow Fl & a \oplus b &= fI(fI(a) + fI(b)). \\ \ominus : \mathbb{R}^2 &\rightarrow Fl & a \ominus b &= fI(fI(a) - fI(b)). \\ \otimes : \mathbb{R}^2 &\rightarrow Fl & a \otimes b &= fI(fI(a) \cdot fI(b)). \\ \oslash : \mathbb{R}^2 &\rightarrow Fl & a \oslash b &= fI(fI(a) \div fI(b)). \end{aligned}$$

i Ejercicio

Considérese $x = \frac{5}{7} \approx \overline{0.714285}$, $y = \frac{1}{3} = \overline{.3}$, $u = 0.714251$, $v = 98765.9$, $w = 0.111111$ y un $\times 10^{-4}$

SPFN con $\beta = 10, k = 5$. Entonces llenar la siguiente tabla de acuerdo a las instrucciones:

1. En la columna "Aritmética de máquina con k=8" se realiza aritmética a 8 dígitos con las operaciones definidas previamente $\oplus, \ominus, \otimes, \oslash$ con la regla de corte $fI_c(\cdot)$.
2. En la columna "Aritmética de máquina con k=5" se realiza aritmética a 5 dígitos con las operaciones definidas previamente $\oplus, \ominus, \otimes, \oslash$ con la regla de corte $fI_c(\cdot)$.
3. Para el cálculo de errores absoluto y relativo tomar como valor real u objetivo el valor a 8 dígitos.

Operación	Aritmética de máquina con k=8	Aritmética de máquina con k=5	Error Absoluto de aprox	Error Relativo de aprox
-----------	----------------------------------	----------------------------------	----------------------------	----------------------------

$x \oplus y$

$x \ominus y$

$x \otimes y$

$x \oslash y$

$x \ominus u$

$(x \ominus u) \oslash w$

$(x \ominus u) \otimes v$

$u \oplus v$

Ejemplo:

El resultado del primer renglón con aritmética exacta es:

$$x + y = \frac{5}{7} + \frac{1}{3} = \frac{22}{21} \approx 0.10476190.$$

Para el llenado de las columnas:

$$\begin{aligned} 1. \text{ Aritmética de máquina a 8 dígitos usamos } k = 8: x \oplus_8 y &= fl_c(0.71428571 + 0.33333333) \\ &= 0.10476190 \times 10^1 \end{aligned}$$

$$\begin{aligned} 2. \text{ Aritmética de máquina usamos } k = 5: x \oplus_5 y &= fl_c(fl_c(x) + fl_c(y)) \\ &= fl_c(0.71428 + 0.33333) = fl_c(0.104761 \times 10^1) \\ &= 0.10476 \times 10^1 \end{aligned}$$

$$\begin{aligned} 3. \text{ Error absoluto de aprox: } ErrAbs(x \oplus_5 y) &= |(x \oplus_8 y) - (x \oplus_5 y)| = |0.10476190 \\ &\quad \times 10^1 - 0.10476 \times 10^1| = |0.00000190| = .190 \\ &\quad \times 10^{-5} \end{aligned}$$

$$\begin{aligned} 4. \text{ Error relativo de aprox: } ErrRel(x \oplus_5 y) &= \frac{ErrAbs(x \oplus_5 y)}{|x \oplus_8 y|} = \frac{.190 \times 10^{-5}}{0.10476190 \times 10^1} \approx 1.81. \\ &\quad \times 10^{-6} = 0.181 \times 10^{-5} \end{aligned}$$

```

//%cflags:-lm
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

double fl_c_k(double a, int k){
    return (double)((int)(pow(10,k)*a)*pow(10,-k));
}

double err_abs(double aprox, double obj){
    return fabs(obj-aprox);
}

double err_rel(double errabs, double obj){
    return errabs/fabs(obj);
}

int main(){
    double x = 5/7.0;
    double y = 1/3.0;
    double res1=0;
    double res2=0;
    int k;

    k=8; //precisión 8
    printf("valor x a 8 dígitos: %0.7e\n", fl_c_k(x,k));
    printf("valor y a 8 dígitos: %0.7e\n", fl_c_k(y,k));
    res1 = fl_c_k(fl_c_k(x,k)+fl_c_k(y,k),k-1); //k-1 para seguir manteniendo 5 dígitos
    pues la suma aumenta un dígito
    printf("x+y a 8 dígitos %0.7e\n", res1);

    k=5; //precisión 5
    printf("valor x a 5 dígitos: %0.4e\n", fl_c_k(x,k));
    printf("valor y a 5 dígitos: %0.4e\n", fl_c_k(y,k));
    res2 = fl_c_k(fl_c_k(x,k)+fl_c_k(y,k),k-1); //k-1 para seguir manteniendo 5 dígitos
    pues la suma aumenta un dígito
    printf("x+y a 5 dígitos %0.4e\n", res2);

    printf("-----\n");
    printf("Error absoluto de aprox: %e\n",err_abs(res2,res1));
    printf("Error relativo de aprox: %e\n",err_rel(err_abs(res2,res1),res1));
    return 0;
}

```

```

valor x a 8 dígitos: 7.1428571e-01
valor y a 8 dígitos: 3.3333333e-01
x+y a 8 dígitos 1.0476190e+00
valor x a 5 dígitos: 7.1428e-01
valor y a 5 dígitos: 3.3333e-01
x+y a 5 dígitos 1.0476e+00
-----
Error absoluto de aprox: 1.900000e-05
Error relativo de aprox: 1.813636e-05

```

Ejercicio

Finalizar la tabla con un programa en C.

Solución de la tabla: [liga](#)

Listando algunos problemas típicos que se presentan en la aritmética de máquina se tienen:

1. Problema de cancelación: pérdida de cifras significativas a partir de la resta de números similares.
2. Suma entre un número de magnitud grande y un número de magnitud pequeña.
3. Sumas con términos que involucren signos alternados.
4. Multiplicación por un número de magnitud grande.
5. División por un número de magnitud pequeña.

Possibles soluciones:

1. Usar mayor precisión.
2. Reordenar operaciones.
3. Reescribir expresiones matemáticas para obtener expresiones equivalentes.
4. Escalar las variables. También funciona estandarizarlas.

Ejercicios

1. Resuelve los ejercicios y preguntas de la nota.
2. Utiliza la notación posicional para representar al número 86409 y $(1001.1)_2$ en base 10.
3. ¿Cuáles de los siguientes números son números de máquina en un SPFN con $\beta = 2$?

a. $(2.125)_{10}$.

b. $(3.1)_{10}$.

tip: escribe los números anteriores como sumas de potencias de 2.

1. Considérese un SPFN con $\beta = 2$, $k = 3$. En este sistema se tienen 7 bits para almacenar números. El primer bit se utiliza para el signo del número, el segundo bit se utiliza para el signo del exponente, los dos siguientes bits para construir al exponente y el último bit para construir a la mantisa. Entonces el número positivo normalizado más pequeño que es posible representar en este SPFN es:

$$(0111100)_2 = (.5 \times 2^{-3})_{10} = .0625.$$

a. Escribe los siguientes números más grandes a este número que forman al SPFN hasta el valor más grande positivo que es posible representar en este SPFN.

b. Calcula las distancias entre los números con mismo valor de exponente.

c. Verifica que el error relativo para $x = 0.156249$ utilizando la regla de corte es menor o igual a $\epsilon_{mag} = .25$ y el error relativo para $x = 0.14$ utilizando la regla de redondeo es menor o igual a $\epsilon_{mag} = 0.125$.

Preguntas de comprensión.

1) ¿Cuáles componentes definen a un sistema de punto flotante?

2) Si un número tiene una representación exacta en la máquina, ¿qué nombre recibe?

3) ¿Qué es un sistema de punto flotante normalizado?

4) Menciona algunas propiedades de un sistema de punto flotante normalizado.

5) ¿Cuántos bits se utilizan en el hardware de una máquina para almacenar un número en un sistema de doble precisión?

6) ¿Qué nombre reciben los errores que se generan por utilizar un sistema de punto flotante?

7) ¿Cuáles reglas utiliza la máquina para dar aproximaciones a un número?

8) Explica con palabras la diferencia entre el epsilon de la máquina y el nivel de underflow.

a. ¿Cuál de ellos depende únicamente del número de dígitos de la mantisa?

b. ¿Cuál de ellos depende únicamente del número de dígitos del exponente?

c. ¿Cuál de ellos no depende de las reglas usadas que se preguntaron en la pregunta 7?

9) Si calculamos un error relativo para una aproximación y resulta ser del orden de 10^{-8} ¿alrededor de cuántos dígitos correctos tengo en mi aproximación?

10) Menciona algunos problemas típicos de la aritmética de máquina y algunas formas de resolverlos.

Referencias:

1. R. L. Burden, J. D. Faires, Numerical Analysis, Brooks/Cole Cengage Learning, 2005.
2. M. T. Heath, Scientific Computing. An Introductory Survey, McGraw-Hill, 2002.
3. [Lenguaje C de programación](#).

💡 Notas para contenedor de docker:

Comando de docker para ejecución de la nota de forma local:

nota: cambiar `<ruta a mi directorio>:/datos` por la ruta de directorio que se desea mapear a `/datos` dentro del contenedor de docker.

```
docker run --rm -v <ruta a mi directorio>:/datos --name jupyterlab_optimizacion -p 8888:8888 -d palmoreck/jupyterlab_optimizacion:2.1.4
```

password para jupyterlab: `qwerty`

Detener el contenedor de docker:

```
docker stop jupyterlab_optimizacion
```

Documentación de la imagen de docker `palmoreck/jupyterlab_optimizacion:2.1.4` en [liga](#).

Nota generada a partir de [liga](#)

1.3 Normas vectoriales y matriciales

Una norma define una medida de distancia en un conjunto y da nociones de tamaño, vecindad, convergencia y continuidad.

Normas vectoriales

Sea \mathbb{R}^n el conjunto de n -tuplas o vectores columna o 1-arreglo de orden 1, esto es:

$$x \in \mathbb{R}^n \iff x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \text{ con } x_i \in \mathbb{R}$$

Una norma vectorial en \mathbb{R}^n es una función $g : \mathbb{R}^n \rightarrow \mathbb{R}$ que satisface las siguientes propiedades:

- g es no negativa: $g(x) \geq 0 \forall x \in \mathbb{R}^n$.
- g es definida: $g(x) = 0 \iff x = 0$.
- g satisface la desigualdad del triángulo: $g(x + y) \leq g(x) + g(y) \forall x, y \in \mathbb{R}^n$.
- g es homogénea: $g(\alpha x) = |\alpha|g(x), \forall \alpha \in \mathbb{R}, \forall x \in \mathbb{R}^n$.

Notación: $g(x) = \|x\|$.

💡 Espacio Vectorial.

Un conjunto $V \neq \emptyset$ en el que se le han definido las operaciones $(+, \cdot)$ se le nombra **espacio vectorial** sobre \mathbb{R} si satisface las siguientes propiedades $\forall x, y, z \in V, \forall a, b \in \mathbb{R}$:

- $x + (y + z) = (x + y) + z$
- $x + y = y + x$
- $\exists 0 \in V$ tal que $x + 0 = 0 + x = x \quad \forall x \in V$.
- $\forall x \in V \exists -x \in V$ tal que $x + (-x) = 0$.
- $a(bx) = (ab)x$.
- $1x = x$ con $1 \in \mathbb{R}$.
- $a(x + y) = ax + ay$.
- $(a + b)x = ax + bx$.

Comentarios y propiedades

- Una norma es una generalización del valor absoluto de \mathbb{R} : $|x|, x \in \mathbb{R}$.
- Un espacio vectorial con una norma definida en éste se le llama **espacio vectorial normado**.
- Una norma es una medida de la longitud de un vector.
- Con una norma es posible definir conceptos como distancia entre vectores:
 $x, y \in \mathbb{R}^n : \text{dist}(x, y) = ||x - y||$.
- Existen varias normas en \mathbb{R}^n siendo las más comunes:
- La norma l_2 , Euclídea o norma 2: $||x||_2$.
- La norma l_1 o norma 1: $||x||_1$.
- La norma ∞ o de Chebyshev o norma infinito: $||x||_\infty$.

Las normas anteriores pertenecen a una familia parametrizada por una constante $p, p \geq 1$ cuyo nombre es norma l_p :

$$||x||_p = \left(\sum_{i=1}^n |x_i|^p \right)^{1/p}.$$

- Un resultado para $x \in \mathbb{R}^n$ es la **equivalencia** entre normas:

$$\begin{aligned} \exists \alpha, \beta > 0 \text{ tales que } \alpha ||x||_a &\leq ||x||_b \\ &\leq \beta ||x||_a \forall x \in \mathbb{R}^n \end{aligned}$$

donde: $||\cdot||_a, ||\cdot||_b$ son normas cualesquiera en \mathbb{R}^n . Por la propiedad anterior decimos que si se cumple convergencia en la norma $||\cdot||_a$ entonces también se cumple convergencia en la norma $||\cdot||_b$.

Ejemplos de gráficas en el plano:

Norma 2: $\{x \in \mathbb{R}^2 \text{ tales que } ||x||_2 \leq 1\}$

```
f=lambda x: np.sqrt(x[:,0]**2 + x[:,1]**2) #definición de norma2
density=1e-5
density_p=int(2.5*10**3)
x=np.arange(-1,1,density)
y1=np.sqrt(1-x**2)
y2=-np.sqrt(1-x**2)
x_p=np.random.uniform(-1,1,(density_p,2))
ind=f(x_p)<=1
x_p_subset=x_p[ind]
plt.plot(x,y1,'b',x,y2,'b')
plt.scatter(x_p_subset[:,0],x_p_subset[:,1],marker='.')
plt.title('Puntos en el plano que cumplen ||x||_2 <= 1')
plt.grid()
plt.show()
```

Norma 1: $\{x \in \mathbb{R}^2 \text{ tales que } ||x||_1 \leq 1\}$

```
f=lambda x:np.abs(x[:,0]) + np.abs(x[:,1]) #definición de norma1
density=1e-5
density_p=int(2.5*10**3)
x1=np.arange(0,1,density)
x2=np.arange(-1,0,density)
y1=1-x1
y2=1+x2
y3=x1-1
y4=-1-x2
x_p=np.random.uniform(-1,1,(density_p,2))
ind=f(x_p)<=1
x_p_subset=x_p[ind]
plt.plot(x1,y1,'b',x2,y2,'b',x1,y3,'b',x2,y4,'b')
plt.scatter(x_p_subset[:,0],x_p_subset[:,1],marker='.')
plt.title('Puntos en el plano que cumplen ||x||_1 <= 1')
plt.grid()
plt.show()
```

Norma ∞ : $\{x \in \mathbb{R}^2 \text{ tales que } ||x||_\infty \leq 1\}$

```
f=lambda x:np.max(np.abs(x),axis=1) #definición de norma infinito
density_p=int(2.5*10**3)
x_p=np.random.uniform(-1,1,(density_p,2))
ind=f(x_p)<=1
x_p_subset=x_p[ind]
plt.scatter(x_p_subset[:,0],x_p_subset[:,1],marker='.')
plt.title('Puntos en el plano que cumplen ||x||_inf <= 1')
plt.grid()
plt.show()
```

->La norma ∞ se encuentra en esta familia como límite:

$$\|x\|_{\infty} = \lim_{p \rightarrow \infty} \|x\|_p.$$

->En la norma l_2 o Euclíadiana $\|x\|_2$ tenemos una desigualdad muy importante, la desigualdad de **Cauchy-Schwartz**:

$$|x^T y| \leq \|x\|_2 \|y\|_2 \forall x, y \in \mathbb{R}^n$$

la cual relaciona el producto interno estándar para $x, y \in \mathbb{R}^n$: $\langle x, y \rangle = x^T y = \sum_{i=1}^n x_i y_i$ con la norma l_2 de x y la norma l_2 de y . Además se utiliza lo anterior para definir el ángulo (sin signo) entre x, y :

$$\angle x, y = \cos^{-1} \left(\frac{x^T y}{\|x\|_2 \|y\|_2} \right)$$

para $\cos^{-1}(u) \in [0, \pi]$ y se nombra a x, y ortogonales si $x^T y = 0$. Obsérvese que $\|x\|_2 = \sqrt{x^T x}$.

- También se utilizan matrices* para definir normas

*Matriz: arreglo 2-dimensional de datos o 2 arreglo de orden 2. Se utiliza la notación $A \in \mathbb{R}^{m \times n}$ para denotar:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-11} & a_{m-12} & \dots & a_{m-1n} \\ a_{m1} & a_{m2} & \dots & a_{mm} \end{bmatrix}$$

$*a_{ij} \in \mathbb{R} \forall i = 1, \dots, m, j = 1, \dots, n$.

$*A = (a_1, \dots, a_n), a_j \in \mathbb{R}^m (= \mathbb{R}^{m \times 1}) \forall j = 1, \dots, n$.

$*A = \begin{pmatrix} a_1^T \\ \vdots \\ a_m^T \end{pmatrix}, a_i \in \mathbb{R}^n (= \mathbb{R}^{n \times 1}) \forall i = 1, \dots, m$.

Entonces un ejemplo de norma-2 ponderada es: $\{x \in \mathbb{R}^2$ tales que $\|x\|_D \leq 1, \|x\|_D = \|Dx\|_2$; con matriz diagonal $D\}$

```
d1=1/5
d2=1/3
f=lambda x: np.sqrt((d1*x[:,0])**2 + (d2*x[:,1])**2) #definición de norma2
density=1e-5
density_p=int(2.5*10**3)
x=np.arange(-1/d1,1/d1,density)
y1=1.0/d2*np.sqrt(1-(d1*x)**2)
y2=-1.0/d2*np.sqrt(1-(d1*x)**2)
x_p=np.random.uniform(-1/d1,1/d1,(density_p,2))
ind=f(x_p)<=1
x_p_subset=x_p[ind]
plt.plot(x,y1,'b',x,y2,'b')
plt.scatter(x_p_subset[:,0],x_p_subset[:,1],marker='.')
plt.title('Puntos en el plano que cumplen ||x||_D <= 1')
plt.grid()
plt.show()
```

en este caso $D = \begin{bmatrix} \frac{1}{5} & 0 \\ 0 & \frac{1}{3} \end{bmatrix}$

Normas matriciales

La multiplicación de una matriz de tamaño $m \times n$ por un vector se define como:

$$y = Ax = \sum_{j=1}^n \alpha_j x_j$$

con $a_j \in \mathbb{R}^m, x \in \mathbb{R}^n$. Obsérvese que $x \in \mathbb{R}^n, Ax \in \mathbb{R}^m$.

Inducidas

De las normas matriciales más importantes se encuentran las **inducidas** por normas vectoriales. Estas normas matriciales se definen en términos de los vectores en \mathbb{R}^n a los que se les aplica la multiplicación Ax :

Dadas las normas vectoriales $\|\cdot\|_{(n)}$, $\|\cdot\|_{(m)}$ en \mathbb{R}^n y \mathbb{R}^m respectivamente, la norma matricial inducida $\|A\|_{(m,n)}$ para $A \in \mathbb{R}^{m \times n}$ es el **menor número** C para el cual la desigualdad:

$$\|Ax\|_{(m)} \leq C\|x\|_{(n)}$$

se cumple $\forall x \in \mathbb{R}^n$. Esto es:

$$\|A\|_{(m,n)} = \sup_{x \in \mathbb{R}^n} \frac{\|Ax\|_{(m)}}{\|x\|_{(n)}}$$

Comentarios:

- $\|A\|_{(m,n)}$ representa el **máximo** factor por el cual A puede modificar el tamaño de x sobre todos los vectores $x \in \mathbb{R}^n$, es una medida de un tipo de **worst case stretch factor**.
- Así definidas, la norma $\|\cdot\|_{(m,n)}$ es la norma matricial inducida por las normas vectoriales $\|\cdot\|_{(m)}$, $\|\cdot\|_{(n)}$.
- Son definiciones equivalentes:

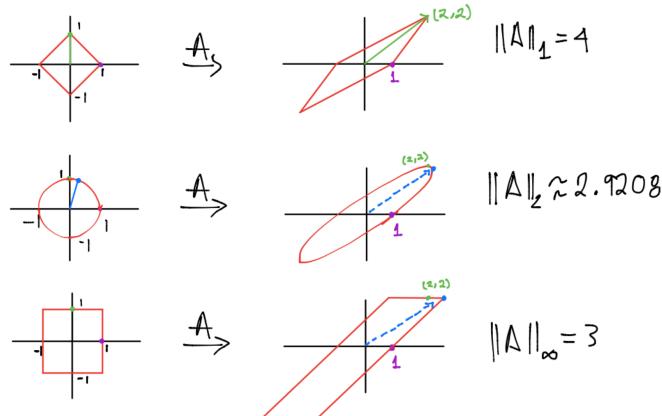
$$\begin{aligned} & \|A\|_{(m,n)} \\ &= \sup_{x \in \mathbb{R}^n} \frac{\|Ax\|_{(m)}}{\|x\|_{(n)}} = \sup_{\|x\|_{(n)} \leq 1} \frac{\|Ax\|_{(m)}}{\|x\|_{(n)}} = \sup_{\|x\|_{(n)} = 1} \|Ax\|_{(m)} \end{aligned}$$

Ejemplo:

La matriz $A = \begin{bmatrix} 1 & 2 \\ 0 & 2 \end{bmatrix}$ mapea \mathbb{R}^2 a \mathbb{R}^2 , en particular se tiene:

- A mapea $e_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ a la columna $a_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ de A .
- A mapea $e_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ a la columna $a_2 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$ de A .

Considerando $\|A\|_p := \|A\|_{(p,p)}$ con $p = 1, p = 2, p = \infty$ se tiene:



Comentario: al observar la segunda gráfica se tiene la siguiente afirmación: la acción de una matriz sobre una circunferencia es una elipse con longitudes de semiejes iguales a $|d_i|$. En general la acción de una matriz sobre una hiperesfera es una hiperelipse. Por lo que los vectores unitarios en \mathbb{R}^n que son más amplificados por la acción de una matriz diagonal $D \in \mathbb{R}^{m \times n}$ con entradas iguales a d_i son aquellos que se mapean a los semiejes de una hiperelipse en \mathbb{R}^m de longitud igual a $\max\{|d_i|\}$ y así tenemos: si D es una matriz diagonal con entradas $|d_i|$ entonces $\|D\|_2 = \max_{i=1,\dots,m} \{|d_i|\}$.

Ejemplo con Python para la norma 1:

```

A=np.array([[1,2],[0,2]])
density=1e-5
x1=np.arange(0,1,density)
x2=np.arange(-1,0,density)
x1_y1 = np.column_stack((x1,1-x1))
x2_y2 = np.column_stack((x2,1+x2))
x1_y3 = np.column_stack((x1,x1-1))
x2_y4 = np.column_stack((x2,-1-x2))
apply_A = lambda vec : np.transpose(A@np.transpose(vec))
A_to_vector_1 = apply_A(x1_y1)
A_to_vector_2 = apply_A(x2_y2)
A_to_vector_3 = apply_A(x1_y3)
A_to_vector_4 = apply_A(x2_y4)
plt.subplot(1,2,1)
plt.plot(x1_y1[:,0],x1_y1[:,1],'b',
         x2_y2[:,0],x2_y2[:,1],'b',
         x1_y3[:,0],x1_y3[:,1],'b',
         x2_y4[:,0],x2_y4[:,1],'b')
e1 = np.column_stack((np.repeat(0,len(x1)),x1))
plt.plot(e1[:,0],e1[:,1],'g')
plt.xlabel('Vectores con norma 1 menor o igual a 1')
plt.grid()
plt.subplot(1,2,2)
plt.plot(A_to_vector_1[:,0],A_to_vector_1[:,1],'b',
         A_to_vector_2[:,0],A_to_vector_2[:,1],'b',
         A_to_vector_3[:,0],A_to_vector_3[:,1],'b',
         A_to_vector_4[:,0],A_to_vector_4[:,1],'b')
A_to_vector_e1 = apply_A(e1)
plt.plot(A_to_vector_e1[:,0],A_to_vector_e1[:,1],'g')
plt.grid()
plt.title('Efecto de la matriz A sobre los vectores con norma 1 menor o igual a 1')
plt.show()

```

```
np.linalg.norm(A,1)
```

Ejercicio: obtener las otras dos gráficas con Python usando norma 2 y norma ∞ . Para el caso de la norma 2 el vector en color azul está dado por la descomposición en valores singulares (SVD) de A. En específico la primer columna de la matriz U multiplicado por el primer valor singular. En el ejemplo resulta en:

$$\sigma_1 U[:,0] \approx 2.9208 * \begin{bmatrix} 0.74967 \\ 0.66180 \end{bmatrix} \approx \begin{bmatrix} 2.189 \\ 1.932 \end{bmatrix}$$

y el vector v que será multiplicado por la matriz A es la primer columna de V dada por:

$$V[:,0] \approx \begin{bmatrix} 0.2566 \\ 0.9664 \end{bmatrix}$$

Resultados computacionales que es posible probar:

$$\begin{aligned} 1. ||A||_1 &= \max_{j=1,\dots,n} \sum_{i=1}^n |a_{ij}|. \\ 2. ||A||_\infty &= \max_{i=1,\dots,n} \sum_{j=1}^n |a_{ij}|. \\ 3. ||A||_2 &= \sqrt{\lambda_{\max}(A^T A)} = \\ &\max \left\{ \sqrt{\lambda} \in \mathbb{R} \mid \lambda \text{ es eigenvalor de } A^T A \right\} \\ &= \max \{ \sigma \in \mathbb{R} \mid \sigma \text{ es valor singular de } A \} = \sigma_{\max}(A) \end{aligned}$$

por ejemplo para la matriz anterior se tiene:

```
np.linalg.norm(A,2)
```

```
_,s,_ = np.linalg.svd(A)
np.max(s)
```

Otras normas matriciales:

- Norma de Frobenius: $||A||_F = \text{tr}(A^T A)^{1/2} = \left(\sum_{i=1}^m \sum_{j=1}^n a_{ij}^2 \right)^{1/2}$.
- Norma "sum-absolute-value": $||A||_{\text{sav}} = \sum_{i=1}^m \sum_{j=1}^n |a_{ij}|$.
- Norma "max-absolute-value": $||A||_{\text{mav}} = \max \{ |a_{ij}| \text{ para } i = 1, \dots, m, j = 1, \dots, n \}$.

Comentarios:

- El producto interno estándar en $\mathbb{R}^{m \times n}$ es: $\langle A, B \rangle = \text{tr}(A^T B) = \sum_{i=1}^m \sum_{j=1}^n a_{ij} b_{ij}$.
- La norma 2 (también llamada normapectral o l_2) y la norma de Frobenius cumplen la propiedad de **consistencia**:

$$\|Ax\| \leq \|A\| \|x\| \quad \forall x \in \mathbb{R}^n, \forall A \in \mathbb{R}^{m \times n}.$$

$$\|AB\| \leq \|A\| \|B\| \quad \forall A,$$

B matrices con dimensiones correspondientes para su multiplicación

Obs: de hecho esta propiedad de consistencia también es cumplida por las normas- p matriciales.

Nota sobre sup

Si $C \subseteq \mathbb{R}$ entonces $a \subseteq \mathbb{R}$ es una **cota superior** en C si

$$x \leq a, \forall x \in C.$$

En \mathbb{R} el conjunto de cotas superiores es \emptyset , \mathbb{R} ó un intervalo de la forma $[b, \infty]$. En el último caso, b se llama **mínima cota superior o supremo del conjunto** C y se denota $\sup C$. Por convención $\sup \emptyset = -\infty$ y $\sup C = \infty$ si C no es acotado por arriba.

Obs: si C es finito, $\sup C$ es el máximo de los elementos de C y típicamente se denota como $\max C$.

Análogamente, $a \in \mathbb{R}$ es una **cota inferior** en $C \subseteq \mathbb{R}$ si $a \leq x, \forall x \in C$.

El **ínfimo o máxima cota inferior** de C es $\inf C = -\sup(-C)$. Por convención $\inf \emptyset = \infty$ y si C no es acotado por debajo entonces $\inf C = -\infty$.

Obs: si C es finito, $\inf C$ es el mínimo de sus elementos y se denota como $\min C$.

Ejercicios

1. Resuelve los ejercicios y preguntas de la nota.

Preguntas de comprensión

1)Menciona 5 propiedades que un conjunto debe cumplir para que sea considerado un espacio vectorial.

2)Menciona las propiedades que debe cumplir una función para que se considere una norma.

3)¿Qué es una norma matricial inducida?, ¿qué mide una norma matricial inducida?

4)¿La norma de Frobenius, es una norma matricial inducida?

5)¿A qué son iguales $\sup(\emptyset)$, $\inf(\emptyset)$? (el conjunto \emptyset es el conjunto vacío)

Referencias

1. L. Trefethen, D. Bau, Numerical linear algebra, SIAM, 1997.

2. G. H. Golub, C. F. Van Loan, Matrix Computations. John Hopkins University Press, 2013

Notas para contenedor de docker:

Comando de docker para ejecución de la nota de forma local:

nota: cambiar `<ruta a mi directorio>` por la ruta de directorio que se desea mapear a `/datos` dentro del contenedor de docker.

```
docker run --rm -v <ruta a mi directorio>:/datos --name jupyterlab_optimizacion -p 8888:8888 -d palmoreck/jupyterlab_optimizacion:2.1.4
```

password para jupyterlab: `qwerty`

Detener el contenedor de docker:

```
docker stop jupyterlab_optimizacion
```

Documentación de la imagen de docker `palmoreck/jupyterlab_optimizacion:2.1.4` en [liga](#).

Nota generada a partir de [liga](#)

Dos temas fundamentales en el análisis numérico son: la **condición de un problema** y **estabilidad de un algoritmo**. El condicionamiento tiene que ver con el comportamiento de un problema ante perturbaciones y la estabilidad con el comportamiento de un algoritmo (usado para resolver un problema) ante perturbaciones.

La exactitud de un cálculo dependerá finalmente de una combinación de estos términos:

$$\text{Exactitud} = \text{Condición} + \text{Estabilidad}$$

La falta de exactitud en un problema se presenta entonces por problemas mal condicionados (no importando si los algoritmos son estables o inestables) y algoritmos inestables (no importando si los problemas son mal o bien condicionados).

1.4 Condición de un problema y estabilidad de un algoritmo

Perturbaciones

La condición de un problema y estabilidad de un algoritmo hacen referencia al término **perturbación**. Tal término conduce a pensar en perturbaciones “chicas” o “grandes”. Para dar una medida de lo anterior se utiliza el concepto de **norma**. Ver final de esta nota para definición de norma y propiedades.

Condición de un problema

Pensemos a un problema como una función $f : \mathbb{X} \rightarrow \mathbb{Y}$ donde \mathbb{X} es un espacio vectorial con norma definida y \mathbb{Y} es otro espacio vectorial de soluciones con una norma definida. Llamemos instancia de un problema a la combinación entre x, f y nos interesa el comportamiento de f en x . Usamos el nombre de “problema” para referirnos al de instancia del problema.

Un problema (instancia) bien condicionado tiene la propiedad de que todas las perturbaciones pequeñas en x conducen a pequeños cambios en $f(x)$. Y es mal condicionado si perturbaciones pequeñas en x conducen a grandes cambios en $f(x)$. El uso de los términos “pequeño” o “grande” dependen del problema mismo.

Sea $\hat{x} = x + \Delta x$ con Δx una perturbación pequeña de x .

El número de condición relativo del problema f en x es:

$$\text{Cond}_f^R = \frac{\text{ErrRel}(f(\hat{x}))}{\text{ErrRel}(\hat{x})} = \frac{\frac{\|f(\hat{x}) - f(x)\|}{\|f(x)\|}}{\frac{\|\hat{x} - x\|}{\|x\|}}$$

considerando $x, f(x) \neq 0$.

Obs: si f es una función diferenciable, podemos evaluar Cond_f^R con la derivada de f , pues a primer orden (usando teorema de Taylor): $f(\hat{x}) - f(x) \approx J_f(x)\Delta x$ con igualdad para $\Delta x \rightarrow 0$ y J_f la Jacobiana de f definida como una matriz con entradas: $(J_f(x))_{ij} = \frac{\partial f_i(x)}{\partial x_j}$. Por tanto, se tiene:

$$\text{Cond}_f^R = \frac{\|J_f(x)\| \|x\|}{\|f(x)\|}$$

y $\|J_f(x)\|$ es una norma matricial inducida por las normas en \mathbb{X}, \mathbb{Y} . Ver final de esta nota para definición de norma y propiedades.

Comentario: en la práctica se considera a un problema **bien condicionado** si Cond_f^R es “pequeño”: menor a 10, **medianamente condicionado** si es de orden entre 10^1 y 10^2 y **mal condicionado** si es “grande”: mayor a 10^3 .

Ejercicio:

Calcular Cond_f^R de los siguientes problemas. Para $x \in \mathbb{R}$ usa el valor absoluto y para $x \in \mathbb{R}^n$ usa $\|x\|_\infty$.

1. $x \in \mathbb{R} - \{0\}$. Problema: realizar la operación $\frac{x}{2}$.

2. $x \geq 0$. Problema: calcular \sqrt{x} .

3. $x \approx \frac{\pi}{2}$. Problema: calcular $\cos(x)$.

4. $x \in \mathbb{R}^2$. Problema: calcular $x_1 - x_2$.

Comentario: las dificultades que pueden surgir al resolver un problema **no** siempre están relacionadas con una fórmula o un algoritmo mal diseñado sino con el problema en cuestión. En el ejercicio anterior, observamos que aún utilizando **aritmética exacta**, la solución del problema puede ser altamente sensible a perturbaciones a los datos de entrada. Por esto el número de condición relativo se define de acuerdo a perturbaciones en los datos de entrada y mide la perturbación en los datos de salida que uno espera:

$$\text{Cond}_f^R = \frac{\|\text{Cambios relativos en la solución}\|}{\|\text{Cambios relativos en los datos de entrada}\|}$$

Estabilidad de un algoritmo

Pensemos a un algoritmo \hat{f} como una función $\hat{f} : \mathbb{X} \rightarrow \mathbb{Y}$ para resolver el problema f con datos $x \in \mathbb{X}$, donde \mathbb{X} es un espacio vectorial con norma definida y \mathbb{Y} es otro espacio vectorial con una norma definida.

La implementación del algoritmo \hat{f} en una máquina conduce a considerar:

- Errores por redondeo:

$$fl(u) = u(1 + \epsilon), |\epsilon| \leq \epsilon_{mag}, \forall u \in \mathbb{R}.$$

- Operaciones en un SPFN, \mathcal{Fl} . Por ejemplo para la suma:

$$u \oplus v = fl(u + v) = (u + v)(1 + \epsilon), |\epsilon| \leq \epsilon_{mag} \forall u, v \in \mathcal{Fl}.$$

Esto es, \hat{f} depende de $x \in \mathbb{X}$ y ϵ_{mag} : representación de los números reales en una máquina y operaciones entre ellos o aritmética de máquina. Ver nota: [Sistema de punto flotante](#).

Al ejecutar \hat{f} obtenemos una colección de números en el SPFN que pertenecen a \mathbb{Y} : $\hat{f}(x)$.

Debido a las diferencias entre un problema con cantidades continuas y una máquina que trabaja con cantidades discretas, los algoritmos numéricos **no** son exactos para **cualquier** elección de datos $x \in \mathbb{X}$. Esto es, los algoritmos **no cumplen** que la cantidad:

$$\frac{\|\hat{f}(x) - f(x)\|}{\|f(x)\|}$$

dependa únicamente de errores por redondeo al evaluar $f \forall x \in \mathbb{X}$. En notación matemática:

$$\frac{\|\hat{f}(x) - f(x)\|}{\|f(x)\|} \leq K \epsilon_{mag} \forall x \in \mathbb{X}$$

con $K > 0$ no se cumple en general.

La razón de lo anterior tiene que ver con cuestiones en la implementación de \hat{f} como el número de iteraciones, la representación de x en un SPFN o el mal condicionamiento de f . Así, a los algoritmos en el análisis numérico, se les pide una condición menos estricta que la anterior y más bien satisfagan lo que se conoce como **estabilidad**. Se dice que un algoritmo \hat{f} para un problema f es **estable** si:

$$\forall x \in \mathbb{X}, \frac{\|\hat{f}(x) - f(\hat{x})\|}{\|f(\hat{x})\|} \leq K_1 \epsilon_{mag}, K_1 > 0$$

para $\hat{x} \in \mathbb{X}$ tal que $\frac{\|\hat{x} - x\|}{\|x\|} \leq K_2 \epsilon_{mag}, K_2 > 0$.

Esto es, \hat{f} resuelve un problema cercano para datos cercanos (cercano en el sentido del ϵ_{mag}) independientemente de la elección de x .

Obs: obsérvese que esta condición es más flexible y en general K_1, K_2 dependen de las dimensiones de \mathbb{X}, \mathbb{Y} .

Comentarios:

- Esta definición resulta apropiada para la mayoría de los problemas en el análisis numérico. Para otras áreas, por ejemplo en ecuaciones diferenciales, donde se tienen definiciones de sistemas dinámicos estables e inestables (cuyas definiciones no se deben confundir con las descritas para algoritmos), esta condición es muy estricta.
- Tenemos algoritmos que satisfacen una condición más estricta y simple que la estabilidad: **estabilidad hacia atrás**:

Estabilidad hacia atrás

Decimos que un algoritmo \hat{f} para el problema f es **estable hacia atrás** si:

$\forall x \in \mathbb{X}, \hat{f}(x) = f(\hat{x})$
 con $\hat{x} \in \mathbb{X}$ tal que $\frac{\|\hat{x}-x\|}{\|x\|} \leq K\epsilon_{mag}, K > 0$.

Esto es, el algoritmo \hat{f} da la solución **exacta** para datos cercanos (cercano en el sentido de ϵ_{mag}), independientemente de la elección de x .

Ejemplo: Para entender la estabilidad hacia atrás de un algoritmo, considérese el ejemplo siguiente.

Problema: evaluar $f(x) = e^x$ en $x = 1$.

Resultado: $f(1) = e^1 = 2.718281\dots$

```
import math
x=1
math.exp(x)
```

Algoritmo: truncar la serie $1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \dots$ a cuatro términos: $\hat{f}(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{6}$.

Resultado del algoritmo: $\hat{f}(1) = 2.\bar{6}$

```
algoritmo = lambda x: 1 + x + x**2/2.0 + x**3/6.0
algoritmo(1)
```

Pregunta: ¿Qué valor $\hat{x} \in \mathbb{R}$ hace que el valor calculado por el algoritmo $\hat{f}(1)$ sea igual a $f(\hat{x})$?

-> **Solución:**

Resolver la ecuación: $e^{\hat{x}} = 2.\bar{6}$, esto es: $\hat{x} = \log(2.\bar{6}) = 0.980829\dots$. Entonces $f(\hat{x}) = 2.\bar{6} = \hat{f}(x)$.

```
x_hat = math.log(algoritmo(1))
x_hat
```

Entonces, el algoritmo es estable hacia atrás sólo si la diferencia entre x y \hat{x} en términos relativos es menor a $K\epsilon_{mag}$ con $K > 0$. Además, podemos calcular **errores hacia delante** y **errores hacia atrás**:

error hacia delante: $\hat{f}(x) - f(x) = -0.05161\dots$, error hacia atrás: $\hat{x} - x = -0.01917\dots$

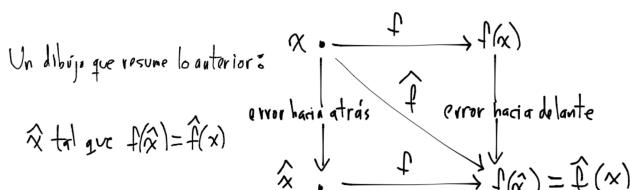
```
err_delante = algoritmo(x) - math.exp(x)
err_delante
```

```
err_atras = x_hat-x
err_atras
```

Dependiendo del problema, estos errores son pequeños o grandes, por ejemplo si consideramos tener una cifra correcta como suficiente para determinar que es una buena aproximación entonces podemos concluir: \hat{f} obtiene una respuesta correcta y cercana al valor de f (error hacia delante) y la respuesta que obtuvimos con \hat{f} es correcta para datos ligeramente perturbados (error hacia atrás).

Obs:

- Obsérvese que el error hacia delante requiere resolver el problema f (para calcular $f(x)$) y también información sobre f .
- En el ejemplo anterior se calcularon $\hat{f}(x)$ y también qué tan larga debe ser la modificación en los datos x , esto es: \hat{x} , para que $\hat{f}(x) = f(\hat{x})$ (error hacia atrás).
- Dibujo que ayuda a ver errores hacia atrás y hacia delante:



En resumen, algunas características de un método **estable** numéricamente respecto al redondeo son:

- Variaciones “pequeñas” en los datos de entrada del método generan variaciones “pequeñas” en la solución del problema.
- No amplifican errores de redondeo en los cálculos involucrados.
- Resuelven problemas “cercanos” para datos ligeramente modificados.

Número de condición de una matriz

En el curso trabajaremos con algoritmos matriciales que son numéricamente estables (o estables hacia atrás) ante errores por redondeo, sin embargo la exactitud que obtengamos con tales algoritmos dependerán de qué tan bien (o mal) condicionado esté el problema. En el caso de matrices la condición de un problema puede ser cuantificada con el **número de condición** de la matriz del problema. Aunque haciendo uso de definiciones como la pseudoinversa de una matriz es posible definir el número de condición para una matriz en general rectangular $A \in \mathbb{R}^{m \times n}$, en esta primera definición consideraremos matrices cuadradas no singulares $A \in \mathbb{R}^{n \times n}$:

$$\text{cond}(A) = \|A\| \|A^{-1}\|.$$

Obs: obsérvese que la norma anterior es una **norma matricial** y $\text{cond}(\cdot)$ puede calcularse para diferentes normas matriciales. Ver final de esta nota para definición de norma y propiedades.

¿Por qué se utiliza la expresión $\|A\| \|A^{-1}\|$ para definir el número de condición de una matriz?

Esta pregunta tiene que ver con el hecho que tal expresión aparece frecuentemente en problemas típicos de matrices. Para lo anterior considérese los siguientes problemas f :

1.Sean $A \in \mathbb{R}^{n \times n}$ no singular, $x \in \mathbb{R}^n$ y f el problema de realizar la multiplicación Ax para x fijo, esto es:

$f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ dada por $f(x) = Ax$. Considérese una perturbación en x : $\hat{x} = x + \Delta x$, entonces:

$$\text{Cond}_f^R = \frac{\text{ErrRel}(f(\hat{x}))}{\text{ErrRel}(\hat{x})} = \frac{\frac{\|f(\hat{x}) - f(x)\|}{\|f(x)\|}}{\frac{\|\hat{x} - x\|}{\|x\|}} \approx \frac{\|\mathcal{J}_f(x)\| \|x\|}{\|f(x)\|}.$$

Para este problema tenemos:

$$\frac{\|\mathcal{J}_f(x)\| \|x\|}{\|f(x)\|} = \frac{\|A\| \|x\|}{\|Ax\|}.$$

Si las normas matriciales utilizadas en el número de condición son consistentes (ver final de esta nota para definición de norma y propiedades) entonces:

$$\|x\| = \|A^{-1}Ax\| \leq \|A^{-1}\| \|Ax\| \therefore \frac{\|x\|}{\|Ax\|} \leq \|A^{-1}\|$$

y se tiene:

$$\text{Cond}_f^R \leq \|A\| \|A^{-1}\|.$$

2.Sean $f : \mathbb{R}^n \rightarrow \mathbb{R}$, $A \in \mathbb{R}^{n \times n}$ no singular. Considérese el problema de calcular $f(b) = A^{-1}b$ para $b \in \mathbb{R}^n$ fijo y la perturbación $\hat{b} = b + \Delta b$ entonces bajo las suposiciones del ejemplo anterior:

$$\text{Cond}_f^R \approx \frac{\|A^{-1}\| \|b\|}{\|A^{-1}b\|}.$$

Si las normas matriciales utilizadas en el número de condición son consistentes (ver final de esta nota para definición de norma y propiedades) entonces:

$$\|b\| = \|AA^{-1}b\| \leq \|A\| \|A^{-1}b\| \therefore \text{Cond}_f^R \leq \|A^{-1}\| \|A\|.$$

3.Sean $f : \mathbb{R}^{n \times n} \rightarrow \mathbb{R}^n$, $A \in \mathbb{R}^{n \times n}$ no singular $b \in \mathbb{R}^n$ fijo. Considérese el problema de calcular la solución x del sistema $Az = b$, esto es, calcular: $x = f(A) = A^{-1}b$. Además, considérese la perturbación $\hat{A} = A + \Delta A$ en el sistema $Az = b$. Se tiene:

$$\hat{x} = \hat{A}^{-1}b,$$

donde: $\hat{x} = x + \Delta x$ (si se perturba A entonces se perturba también x).

De la ecuación anterior como $\hat{x} = \hat{A}^{-1}b$ se tiene:

$$\begin{aligned} \hat{A}\hat{x} &= b \\ (A + \Delta A)(x + \Delta x) &= b \\ Ax + A\Delta x + \Delta Ax + \Delta A\Delta x &= b \end{aligned}$$

$$b + A\Delta x + \Delta Ax = b$$

Donde en esta última ecuación se supuso que $\Delta A\Delta x \approx 0$ y de aquí:

$$A\Delta x + \Delta Ax \approx 0 \therefore \Delta x \approx -A^{-1}\Delta Ax.$$

Entonces se tiene que la condición del problema f : calcular la solución de sistema de ecuaciones lineales $Az = b$ con A no singular ante perturbaciones en A es:

$$\text{Cond}_f^R = \frac{\frac{||x - \hat{x}||}{||x||}}{\frac{||A - \hat{A}||}{||A||}} = \frac{\frac{||\Delta x||}{||x||}}{\frac{||\Delta A||}{||A||}} \leq \frac{\frac{||A^{-1}|| ||\Delta Ax||}{||x||}}{\frac{||\Delta A||}{||A||}} \leq ||A^{-1}|| ||A||.$$

¿Qué está midiendo el número de condición de una matriz respecto a un sistema de ecuaciones lineales?

El número de condición de una matriz mide la **sensibilidad** de la solución de un sistema de ecuaciones lineales ante perturbaciones en los datos de entrada (en la matriz del sistema A o en el lado derecho b). Si pequeños cambios en los datos de entrada generan grandes cambios en la solución tenemos un **sistema mal condicionado**. Si pequeños cambios en los datos de entrada generan pequeños cambios en la solución tenemos un sistema **bien condicionado**. Lo anterior puede apreciarse con los siguientes ejemplos y gráficas:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy
import pprint
```

1. Resolver los siguientes sistemas:

$$\begin{array}{l} a) \begin{aligned} x_1 + 2x_2 &= 10 \\ 1.1x_1 + 2x_2 &= 10.4 \end{aligned} \\ b) \begin{aligned} 1.05x_1 + 2x_2 &= 10 \\ 1.1x_1 + 2x_2 &= 10.4 \end{aligned} \end{array}$$

```
print('inciso a')
A = np.array([[1, 2], [1.1, 2]])
b = np.array([10, 10.4])
print('matriz A:')
pprint.pprint(A)
print('lado derecho b:')
pprint.pprint(b)
```

```
inciso a
matriz A:
array([[1. , 2. ],
       [1.1, 2.]])
lado derecho b:
array([10. , 10.4])
```

```
x=np.linalg.solve(A,b)
print('solución x:')
pprint.pprint(x)
```

```
solución x:
array([4., 3.])
```

```
x=np.arange(0,10, .5)
recta1 = lambda x: 1/2.0*(10-1*x)
recta2 = lambda x: 1/2.0*(10.4-1.1*x)
plt.plot(x,recta1(x), 'o-', x, recta2(x), '^-')
plt.title('Sistema mal condicionado')
plt.legend(['x1+2x2=10', '1.1x1+2x2=10.4'])
plt.grid(True)
plt.show()
```

Obs: obsérvese que las dos rectas anteriores tienen una inclinación (pendiente) similar por lo que no se ve claramente el punto en el que intersectan.

```

print('inciso b')
A = np.array([[1.05, 2], [1.1, 2]])
b = np.array([10, 10.4])
print('matriz A ligeramente modificada:')
 pprint.pprint(A)
print('lado derecho b:')
 pprint.pprint(b)

```

```

inciso b
matriz A ligeramente modificada:
array([[1.05, 2. ],
       [1.1 , 2. ]])
lado derecho b:
array([10. , 10.4])

```

```

x=np.linalg.solve(A,b)
print('solución x:')
 pprint.pprint(x)

```

```

solución x:
array([8. , 0.8])

```

```

x=np.arange(0,10,.5)
recta1 = lambda x: 1/2.0*(10-1.05*x)
recta2 = lambda x: 1/2.0*(10.4-1.1*x)
plt.plot(x,recta1(x),'o-',x,recta2(x),'^-')
plt.title('Sistema mal condicionado')
plt.legend(['1.05x1+2x2=10','1.1x1+2x2=10.4'])
plt.grid(True)
plt.show()

```

Obs: al modificar un poco las entradas de la matriz A la solución del sistema cambia drásticamente.

Comentario: otra forma de describir a un sistema mal condicionado es que un amplio rango de valores en un SPFN satisfacen tal sistema de forma aproximada.

2. Resolver los siguientes sistemas:

$$\begin{aligned} a) \quad .03x_1 + 58.9x_2 &= 59.2 \\ 5.31x_1 - 6.1x_2 &= 47 \\ a) \quad .03x_1 + 58.9x_2 &= 59.2 \\ 5.31x_1 - 6.05x_2 &= 47 \end{aligned}$$

```

print('inciso a')
A = np.array([[.03, 58.9], [5.31, -6.1]])
b = np.array([59.2, 47])
print('matriz A:')
 pprint.pprint(A)
print('lado derecho b:')
 pprint.pprint(b)

```

```

inciso a
matriz A:
array([[ 3.00e-02,  5.89e+01],
       [ 5.31e+00, -6.10e+00]])
lado derecho b:
array([59.2, 47. ])

```

```

x=np.linalg.solve(A,b)
print('solución x:')
 pprint.pprint(x)

```

```

solución x:
array([10.,  1.])

```

```

x=np.arange(4,14,.5)
recta1 = lambda x: 1/58.9*(59.2-.03*x)
recta2 = lambda x: 1/6.1*(5.31*x-47)
plt.plot(x,recta1(x),'o-',x,recta2(x),'^-')
plt.title('Sistema bien condicionado')
plt.legend(['.03x1+58.9x2=59.2','5.31x1-6.1x2=47'])
plt.grid(True)
plt.show()

```

Obs: obsérvese que la solución del sistema de ecuaciones (intersección entre las dos rectas) está claramente definido.

```

print('inciso b')
A = np.array([[.03, 58.9], [5.31, -6.05]])
b = np.array([59.2, 47])
print('matriz A ligeramente modificada:')
 pprint.pprint(A)
print('lado derecho b:')
 pprint.pprint(b)

```

```

inciso b
matriz A ligeramente modificada:
array([[ 3.00e-02,  5.89e+01],
       [ 5.31e+00, -6.05e+00]])
lado derecho b:
array([59.2, 47. ])

```

```

x=np.linalg.solve(A,b)
print('solución x:')
 pprint.pprint(x)

```

```

solución x:
array([9.99058927, 1.00000479])

```

```

x=np.arange(4,14,.5)
recta1 = lambda x: 1/58.9*(59.2-.03*x)
recta2 = lambda x: 1/6.05*(5.31*x-47)
plt.plot(x,recta1(x), 'o-',x,recta2(x), '^-')
plt.title('Sistema bien condicionado')
plt.legend(['.03x1+58.9x2=59.2', '5.31x1-6.05x2=47'])
plt.grid(True)
plt.show()

```

Obs: al modificar un poco las entradas de la matriz A la solución **no** cambia mucho.

Comentarios:

1.¿Por qué nos interesa considerar perturbaciones en los datos de entrada? -> recuérdese que los números reales se representan en la máquina mediante el sistema de punto flotante (SPF), entonces al ingresar datos a la máquina tenemos perturbaciones y por tanto errores de redondeo. Ver nota: [Sistema de punto flotante](#).

2.Las matrices anteriores tienen número de condición distinto:

```

print('matriz del ejemplo 1')
A = np.array([[1, 2], [1.1, 2]])
 pprint.pprint(A)

```

```

matriz del ejemplo 1
array([[1. , 2. ],
       [1.1, 2. ]])

```

su número de condición es:

```

np.linalg.cond(A)

```

```

print('matriz del ejemplo 2')
A = np.array([[.03, 58.9], [5.31, -6.1]])
 pprint.pprint(A)

```

```

matriz del ejemplo 2
array([[ 3.00e-02,  5.89e+01],
       [ 5.31e+00, -6.10e+00]])

```

su número de condición es:

```

np.linalg.cond(A)

```

Las matrices del ejemplo 1 y 2 son **medianamente** condicionadas. Una matriz se dice **bien condicionada** si $\text{cond}(A)$ es cercano a 1.

Algunas propiedades del número de condición de una matriz

- Si $A \in \mathbb{R}^{n \times n}$ es no singular entonces:

$$\frac{1}{\text{cond}(A)} = \min \left\{ \frac{\|A - B\|}{\|A\|} \mid \begin{array}{l} \text{tal que } B \text{ es singular, } \|} \\ \cdot \| \text{ es una norma inducida} \end{array} \right\}.$$

esto es, una matriz mal condicionada (número de condición grande) se le puede aproximar muy bien por una matriz singular. Sin embargo, el mal condicionamiento no necesariamente se relaciona con singularidad. Una matriz singular es mal condicionada pero una matriz mal condicionada no necesariamente es singular. Considérese por ejemplo la matriz de **Hilbert**:

```
from scipy.linalg import hilbert
```

```
hilbert(4)
```

```
np.linalg.cond(hilbert(4))
```

la cual es una matriz mal condicionada pero es no singular:

```
np.linalg.inv(hilbert(4))@hilbert(4)
```

y otro ejemplo de una matriz singular:

```
print('matriz singular')
A = np.array([[1, 2], [1, 2]])
pprint(A)
```

```
matriz singular
array([[1, 2],
       [1, 2]])
```

```
np.linalg.inv(A)
```

```
LinAlgError                                         Traceback (most recent call last)
<ipython-input-29-ae645f97e1f8> in <module>
----> 1 np.linalg.inv(A)

<__array_function__ internals> in inv(*args, **kwargs)

~/local/lib/python3.7/site-packages/numpy/linalg/linalg.py in inv(a)
    544     signature = 'D->D' if isComplexType(t) else 'd->d'
    545     extobj = get_linalg_error_extobj(_raise_linalgerror_singular)
--> 546     ainv = _umath_linalg.inv(a, signature=signature, extobj=extobj)
    547     return wrap(ainv.astype(result_t, copy=False))
    548

~/local/lib/python3.7/site-packages/numpy/linalg/linalg.py in
_raise_linalgerror_singular(err, flag)
    86
    87 def _raise_linalgerror_singular(err, flag):
--> 88     raise LinAlgError("Singular matrix")
    89
    90 def _raise_linalgerror_nonposdef(err, flag):

LinAlgError: Singular matrix
```

```
np.linalg.cond(A)
```

- Para las normas matriciales inducidas se tiene:

- $\text{cond}(A) \geq 1, \forall A \in \mathbb{R}^{n \times n}$.
- $\text{cond}(\gamma A) = \text{cond}(A), \forall \gamma \in \mathbb{R} - \{0\}, \forall A \in \mathbb{R}^{n \times n}$.
- $\text{cond}_2(A) = \|A\|_2 \|A^{-1}\|_2 = \frac{\sigma_{\max}}{\sigma_{\min}}, \sigma_{\min} \neq 0$.

- En el problema: resolver $Ax = b$ se cumple:

$$\text{ErrRel}(\hat{x}) = \frac{\|x^* - \hat{x}\|}{\|x^*\|} \leq \text{cond}(A) \left(\frac{\|\Delta A\|}{\|A\|} + \frac{\|\Delta b\|}{\|b\|} \right), b \\ \neq 0.$$

donde: x^* es solución de $Ax = b$ y \hat{x} es solución aproximada que se obtiene por algún método numérico (por ejemplo factorización LU). $\frac{\|\Delta A\|}{\|A\|}, \frac{\|\Delta b\|}{\|b\|}$ son los errores relativos en las entradas de A y b respectivamente.

Comentario: la desigualdad anterior se puede interpretar como sigue: si sólo tenemos perturbaciones en A de modo que se tienen errores por redondeo del orden de 10^{-k} y por lo tanto k dígitos de precisión en A y $\text{cond}(A)$ es del orden de 10^c entonces $\text{ErrRel}(\hat{x})$ puede llegar a tener errores de redondeo de a lo más del orden de 10^{c-k} y por tanto $k - c$ dígitos de precisión:

$$\text{ErrRel}(\hat{x}) \leq \text{cond}(A) \frac{\|\Delta A\|}{\|A\|}.$$

- Supongamos que x^* es solución del sistema $Ax = b$ y obtenemos \hat{x} por algún método numérico (por ejemplo factorización LU) entonces ¿qué condiciones garantizan que $\|x^* - \hat{x}\|$ sea cercano a cero (del orden de $\epsilon_{mag} = 10^{-16}$), ¿de qué depende esto?

Para responder las preguntas anteriores definimos el residual de $Ax = b$ como $r = A\hat{x} - b$ con \hat{x} aproximación a x^* obtenida por algún método numérico. Asimismo, el residual relativo a la norma de b como:

$$\frac{\|r\|}{\|b\|}.$$

Obs: típicamente x^* (solución exacta) es desconocida y por ello no podríamos calcular $\|x^* - \hat{x}\|$, sin embargo sí podemos calcular el residual relativo a la norma de b : $\frac{\|r\|}{\|b\|}$. ¿Se cumple que $\frac{\|r\|}{\|b\|}$ pequeño implica $\text{ErrRel}(\hat{x})$ pequeño?

El siguiente resultado nos ayuda a responder esta y las preguntas anteriores:

Sea $A \in \mathbb{R}^{n \times n}$ no singular, x^* solución de $Ax = b$, \hat{x} aproximación a x^* , entonces para las normas matriciales inducidas se cumple:

$$\frac{\|r\|}{\|b\|} \frac{1}{\text{cond}(A)} \leq \frac{\|x^* - \hat{x}\|}{\|x^*\|} \leq \text{cond}(A) \frac{\|r\|}{\|b\|}.$$

Por la desigualdad anterior, si $\text{cond}(A) \approx 1$ entonces $\frac{\|r\|}{\|b\|}$ es una buena estimación de $\text{ErrRel}(\hat{x}) = \frac{\|x^* - \hat{x}\|}{\|x^*\|}$ por lo que \hat{x} es una buena estimación de x^* . Si $\text{cond}(A)$ es grande no podemos decir **nada** acerca de $\text{ErrRel}(\hat{x})$ ni de \hat{x} .

Ejemplos:

1.

$$a) \begin{array}{rcl} x_1 + x_2 & = & 2 \\ 10.05x_1 + 10x_2 & = & 21 \end{array}$$

$$b) \begin{array}{rcl} x_1 + x_2 & = & 2 \\ 10.1x_1 + 10x_2 & = & 21 \end{array}$$

```
print('inciso a')
A_1 = np.array([[1, 1], [10.05, 10]])
b_1 = np.array([2, 21])
print('matriz A_1:')
 pprint.pprint(A_1)
print('lado derecho b_1:')
 pprint.pprint(b_1)
```

```
inciso a
matriz A_1:
array([[ 1. ,  1. ],
       [10.05, 10. ]])
lado derecho b_1:
array([ 2, 21])
```

```
x_est=np.linalg.solve(A_1,b_1)
print('solución x_est:')
 pprint.pprint(x_est)
```

```
solución x_est:
array([ 20., -18.])
```

```

print('inciso b')
A_2 = np.array([[1, 1], [10.1, 10]])
b_2 = np.array([2, 21])
print('matriz A_2:')
 pprint.pprint(A_2)
print('lado derecho b_2:')
 pprint.pprint(b_2)

```

```

inciso b
matriz A_2:
array([[ 1. ,  1. ],
       [10.1, 10. ]])
lado derecho b_2:
array([ 2, 21])

```

```

x_hat=np.linalg.solve(A_2,b_2)
print('solución x_hat:')
 pprint.pprint(x_hat)

```

```

solución x_hat:
array([10., -8.])

```

```

print('residual relativo:')
r_rel = np.linalg.norm(A_1@x_hat-b_1)/np.linalg.norm(b_1)
r_rel

```

```

residual relativo:
```

```

print('error relativo:')
err_rel = np.linalg.norm(x_hat-x_est)/np.linalg.norm(x_est)
 pprint.pprint(err_rel)

```

```

error relativo:
0.5255883312276278

```

no tenemos una buena estimación del error relativo a partir del residual relativo pues:

```

np.linalg.cond(A_1)

```

De acuerdo a la cota del resultado el error relativo se encuentra en el intervalo:

```

(r_rel*1/np.linalg.cond(A_1), r_rel*np.linalg.cond(A_1))

```

1.

$$\begin{aligned}
 a) \quad & 4.1x_1 + 2.8x_2 = 4.1 \\
 & 9.7x_1 + 6.6x_2 = 9.7 \\
 b) \quad & 4.1x_1 + 2.8x_2 = 4.11 \\
 & 9.7x_1 + 6.6x_2 = 9.7
 \end{aligned}$$

```

print('inciso a')
A_1 = np.array([[4.1, 2.8], [9.7, 6.6]])
b_1 = np.array([4.1, 9.7])
print('matriz A_1:')
 pprint.pprint(A_1)
print('lado derecho b_1:')
 pprint.pprint(b_1)

```

```

inciso a
matriz A_1:
array([[4.1, 2.8],
       [9.7, 6.6]])
lado derecho b_1:
array([4.1, 9.7])

```

```

x_est=np.linalg.solve(A_1,b_1)
print('solución x_est:')
 pprint.pprint(x_est)

```

```

solución x_est:
array([1., 0.])

```

```

print('inciso b')
A_2 = np.array([[4.1, 2.8], [9.7, 6.6]])
b_2 = np.array([4.11, 9.7])
print('matriz A_2:')
 pprint.pprint(A_2)
print('lado derecho b_2:')
 pprint.pprint(b_2)

```

```

inciso b
matriz A_2:
array([[4.1, 2.8],
       [9.7, 6.6]])
lado derecho b_2:
array([4.11, 9.7])

```

```

x_hat=np.linalg.solve(A_2,b_2)
print('solución x_hat:')
 pprint.pprint(x_hat)

```

```

solución x_hat:
array([0.34, 0.97])

```

```

print('residual relativo:')
r_rel = np.linalg.norm(A_1@x_hat-b_1)/np.linalg.norm(b_1)
r_rel

```

```

residual relativo:
```

```

print('error relativo:')
err_rel = np.linalg.norm(x_hat-x_est)/np.linalg.norm(x_est)
 pprint.pprint(err_rel)

```

```

error relativo:
1.1732433677631406

```

no tenemos una buena estimación del error relativo a partir del residual relativo pues:

```

np.linalg.cond(A_1)

```

```

(r_rel*1/np.linalg.cond(A_1), r_rel*np.linalg.cond(A_1))

```

1.

$$\begin{aligned}
 a) \quad & 3.9x_1 + 11.6x_2 = 5.5 \\
 & 12.8x_1 + 2.9x_2 = 9.7 \\
 b) \quad & 3.95x_1 + 11.6x_2 = 5.5 \\
 & 12.8x_1 + 2.9x_2 = 9.7
 \end{aligned}$$

```

print('inciso a')
A_1 = np.array([[3.9, 11.6], [12.8, 2.9]])
b_1 = np.array([5.5, 9.7])
print('matriz A_1:')
 pprint.pprint(A_1)
print('lado derecho b_1:')
 pprint.pprint(b_1)

```

```

inciso a
matriz A_1:
array([[ 3.9, 11.6],
       [12.8,  2.9]])
lado derecho b_1:
array([5.5, 9.7])

```

```

x_est=np.linalg.solve(A_1,b_1)
print('solución x_est:')
 pprint.pprint(x_est)

```

```

solución x_est:
array([0.70401691, 0.23744259])

```

```

print('inciso b')
A_2 = np.array([[3.95, 11.6], [12.8, 2.9]])
b_2 = np.array([5.5, 9.7])
print('matriz A_2:')
 pprint.pprint(A_2)
print('lado derecho b_2:')
 pprint.pprint(b_2)

```

```

inciso b
matriz A_2:
array([[ 3.95, 11.6 ],
       [12.8 ,  2.9 ]])
lado derecho b_2:
array([5.5, 9.7])

```

```

x_hat=np.linalg.solve(A_2,b_2)
print('solución x_hat:')
 pprint.pprint(x_hat)

```

```

solución x_hat:
array([0.7047619 , 0.23415435])

```

```

print('residual relativo:')
r_rel = np.linalg.norm(A_1@x_hat-b_1)/np.linalg.norm(b_1)
r_rel

```

```

residual relativo:
```

```

print('error relativo:')
err_rel = np.linalg.norm(x_hat-x_est)/np.linalg.norm(x_est)
 pprint.pprint(err_rel)

```

```

error relativo:
0.004537910940159858

```

sí tenemos una buena estimación del error relativo a partir del residual relativo pues:

```
np.linalg.cond(A_1)
```

```
(r_rel*1/np.linalg.cond(A_1), r_rel*np.linalg.cond(A_1))
```

1.

$$\theta = \frac{\pi}{3}$$

```
theta_1=math.pi/3
```

```
(math.cos(theta_1),math.sin(theta_1))
```

```
theta_2 = math.pi/3 + .00005
```

```
theta_2
```

```
(math.cos(theta_2),math.sin(theta_2))
```

$$\begin{aligned}
 a) \quad & \cos(\theta_1)x_1 - \sin(\theta_1)x_2 = -1.5 \\
 & \sin(\theta_1)x_1 + \cos(\theta_1)x_2 = 2.4 \\
 b) \quad & \cos(\theta_2)x_1 - \sin(\theta_2)x_2 = -1.5 \\
 & \sin(\theta_2)x_1 + \cos(\theta_2)x_2 = 2.4 \\
 c) \quad & \cos(\theta_2)x_1 - \sin(\theta_2)x_2 = -1.7 \\
 & \sin(\theta_2)x_1 + \cos(\theta_2)x_2 = 2.4
 \end{aligned}$$

```

print('inciso a')
A_1 = np.array([[math.cos(theta_1), -math.sin(theta_1)], [math.sin(theta_1),
math.cos(theta_1)]])
b_1 = np.array([-1.5, 2.4])
print('matriz A_1:')
 pprint.pprint(A_1)
print('lado derecho b_1:')
 pprint.pprint(b_1)

```

```

inciso a
matriz A_1:
array([[ 0.5        , -0.8660254],
       [ 0.8660254,  0.5        ]])
lado derecho b_1:
array([-1.5,  2.4])

```

```

x_est=np.linalg.solve(A_1,b_1)
print('solución x_est:')
 pprint.pprint(x_est)

```

```

solución x_est:
array([1.32846097, 2.49903811])

```

```

print('inciso b')
A_2 = np.array([[math.cos(theta_2), -math.sin(theta_2)], [math.sin(theta_2),
math.cos(theta_2)]])
b_2 = np.array([-1.5, 2.4])
print('matriz A_2:')
 pprint.pprint(A_2)
print('lado derecho b_2:')
 pprint.pprint(b_2)

```

```

inciso b
matriz A_2:
array([[ 0.4999567, -0.8660504],
       [ 0.8660504,  0.4999567]])
lado derecho b_2:
array([-1.5,  2.4])

```

```

x_hat=np.linalg.solve(A_2,b_2)
print('solución x_hat:')
 pprint.pprint(x_hat)

```

```

solución x_hat:
array([1.32858592, 2.49897168])

```

```

print('residual relativo:')
r_rel = np.linalg.norm(A_1@x_hat-b_1)/np.linalg.norm(b_1)
'{:0.10e}'.format(r_rel)

```

```

residual relativo:

```

```

print('error relativo:')
err_rel = np.linalg.norm(x_hat-x_est)/np.linalg.norm(x_est)
'{:0.10e}'.format(err_rel)

```

```

error relativo:

```

sí tenemos una buena estimación del error relativo a partir del residual relativo pues:

```

np.linalg.cond(A_1)

```

```

('{:0.10e}'.format(r_rel*1/np.linalg.cond(A_1)),
 '{:0.10e}'.format(r_rel*np.linalg.cond(A_1)))

```

```

print('inciso c')
A_2 = np.array([[math.cos(theta_2), -math.sin(theta_2)], [math.sin(theta_2),
math.cos(theta_2)]])
b_2 = np.array([-1.7, 2.4])
print('matriz A_2:')
pprint.pprint(A_2)
print('lado derecho b_2:')
pprint.pprint(b_2)

```

```

inciso c
matriz A_2:
array([[ 0.4999567, -0.8660504],
       [ 0.8660504,  0.4999567]])
lado derecho b_2:
array([-1.7,  2.4])

```

```

x_hat=np.linalg.solve(A_2,b_2)
print('solución x_hat:')
pprint.pprint(x_hat)

```

```

solución x_hat:
array([1.22859458, 2.67218176])

```

```

print('residual relativo:')
r_rel = np.linalg.norm(A_1@x_hat-b_1)/np.linalg.norm(b_1)
'{:0.14e}'.format(r_rel)

```

```

residual relativo:
```

```

print('error relativo:')
err_rel = np.linalg.norm(x_hat-x_est)/np.linalg.norm(x_est)
'{:0.14e}'.format(err_rel)

```

```

error relativo:
```

sí tenemos una buena estimación del error relativo a partir del residual relativo pues:

```

np.linalg.cond(A_1)

```

```

('{:0.14e}'.format(r_rel*1/np.linalg.cond(A_1)),
 '{:0.14e}'.format(r_rel*np.linalg.cond(A_1)))

```

Así, $\text{cond}(A)$ nos da una calidad (mediante $\frac{\|r\|}{\|b\|}$) de la solución \hat{x} en el problema inicial (resolver $Ax = b$) obtenida por algún método numérico respecto a la solución x^* de $Ax = b$.

Obs: Por último obsérvese que la condición del problema inicial (resolver $Ax = b$) **no depende del método numérico** que se elige para resolverlo.

Ejercicio: proponer sistemas de ecuaciones lineales con distinto número de condición, perturbar matriz del sistema o lado derecho (o ambos) y revisar números de condición y residuales relativos de acuerdo a la cota:

$$\frac{\|r\|}{\|b\|} \frac{1}{\text{cond}(A)} \leq \frac{\|x^* - \hat{x}\|}{\|x^*\|} \leq \text{cond}(A) \frac{\|r\|}{\|b\|}.$$

Verificar que si el número de condición del sistema es pequeño entonces el residual relativo estima bien al error relativo.

Número de condición de una matriz $A \in \mathbb{R}^{m \times n}$

Para este caso se utiliza la **pseudoinversa** de A definida a partir de la descomposición en valores singulares compacta (compact SVD, ver [Factorizaciones matriciales SVD, Cholesky, QR](#)) y denotada como A^\dagger :

$$A^\dagger = V\Sigma^\dagger U^T$$

donde: Σ^\dagger es la matriz transpuesta de Σ y tiene entradas σ_i^+ :

$$\sigma_i^+ = \begin{cases} \frac{1}{\sigma_i} & \text{si } \sigma_i \neq 0, \\ 0 & \text{en otro caso} \end{cases}$$

$\forall i = 1, \dots, r$ con $r = \text{rank}(A)$.

Comentarios y propiedades:

- A^\dagger se le conoce como pseudoinversa de *Moore – Penrose*.
- Si $\text{rank}(A) = n$ entonces $A^\dagger = (A^T A)^{-1} A^T$, si $\text{rank}(A) = m$, $A^\dagger = A^T (A A^T)^{-1}$, si $A \in \mathbb{R}^{n \times n}$ no singular, entonces $A^\dagger = A^{-1}$.
- Con A^\dagger se define $\text{cond}(A)$ para $A \in \mathbb{R}^{m \times n}$:

$$\text{cond}(A) = ||A|| ||A^\dagger||$$

de hecho, se tiene:

$$\text{cond}_2(A) = \frac{\sigma_{\max}}{\sigma_{\min}} = \frac{\sigma_1}{\sigma_r}.$$

Ejercicios

1. Resuelve los ejercicios y preguntas de la nota.

Preguntas de comprensión

- 1) ¿Qué factores influyen en la falta de exactitud de un cálculo?
- 2) Si f es un problema mal condicionado, ¿a qué nos referimos? Da ejemplos de problemas bien y mal condicionados.
- 3) Si f es un problema que resolvemos con un algoritmo g , ¿qué significa:

- a. que g sea estable?
- b. que g sea estable hacia atrás?
- c. que g sea inestable?

- 4) ¿Qué ventaja(s) se tiene(n) al calcular un error hacia atrás vs calcular un error hacia delante?

Referencias

1. Nota [Sistema de punto flotante](#).
2. L. Trefethen, D. Bau, Numerical linear algebra, SIAM, 1997.
3. G. H. Golub, C. F. Van Loan, Matrix Computations. John Hopkins University Press, 2013

💡 Notas para contenedor de docker:

Comando de docker para ejecución de la nota de forma local:

nota: cambiar `<ruta a mi directorio>` por la ruta de directorio que se desea mapear a `/datos` dentro del contenedor de docker.

```
docker run --rm -v <ruta a mi directorio>:/datos --name jupyterlab_optimizacion -p  
8888:8888 -d palmoreck/jupyterlab_optimizacion:2.1.4
```

password para jupyterlab: `qwerty`

Detener el contenedor de docker:

```
docker stop jupyterlab_optimizacion
```

Documentación de la imagen de docker `palmoreck/jupyterlab_optimizacion:2.1.4` en [liga](#).

Nota generada a partir de la [liga1](#), [liga2](#) e inicio de [liga3](#).

1.5 Definición de función y derivada de una función

Notación: $f : A \rightarrow B$ es una función de un conjunto $\text{dom } f \subseteq A$ en un conjunto B .

💡 Observación

$\text{dom } f$ (el dominio de f) podría ser un subconjunto propio de A , esto es, algunos elementos de A y otros no, son mapeados a elementos de B .

En lo que sigue se considera al espacio \mathbb{R}^n y se asume que conjuntos y subconjuntos están en este espacio.

Un punto x se nombra **punto límite** de un conjunto X , si existe una sucesión $\{x_k\} \subset X$ que converge a x . El conjunto de puntos límites se nombra **cerradura** o *closure* de X y se denota como $\text{cl } X$.

Un conjunto X se nombra **cerrado** si es igual a su cerradura.

En lo siguiente $\text{intdom } f$ es el **interior*** del dominio de f .

*El interior es el conjunto de **puntos interiores**: un punto x de un conjunto X se llama interior si existe una **vecindad** de x (conjunto abierto* que contiene a x) contenida en X .

*Un conjunto X se dice que es **abierto** si $\forall x \in X$ existe una bola abierta* centrada en x y contenida en X . Es equivalente escribir que X es **abierto** si su complemento $\mathbb{R}^n \setminus X$ es cerrado.

*Una **bola abierta** con radio $\epsilon > 0$ y centrada en x es el conjunto: $B_\epsilon(x) = \{y \in \mathbb{R}^n : \|y - x\| < \epsilon\}$.

Continuidad

$f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ es continua en $x \in \text{dom } f$ si $\forall \epsilon > 0 \exists \delta > 0$ tal que:

$$y \in \text{dom } f, \|y - x\|_2 \leq \delta \implies \|f(y) - f(x)\|_2 \leq \epsilon$$

Comentarios

- f continua en un punto x del dominio de f entonces $f(y)$ es arbitrariamente cercana a $f(x)$ para y en el dominio de f cercana a x .
- Otra forma de definir que f sea continua en $x \in \text{dom } f$ es con sucesiones y límites: si $\{x_i\}_{i \in \mathbb{N}} \subseteq \text{dom } f$ es una sucesión de puntos en el dominio de f que converge a $x \in \text{dom } f$, $\lim_{i \rightarrow \infty} x_i = x$, f es continua en x entonces la sucesión $\{f(x_i)\}_{i \in \mathbb{N}}$ converge a $f(x)$:

$$\lim_{i \rightarrow \infty} f(x_i) = f(x) = f\left(\lim_{i \rightarrow \infty} x_i\right).$$

Notación: $C([a, b])$

$= \{\text{funciones } f : \mathbb{R} \rightarrow \mathbb{R} \text{ continuas en el intervalo } [a, b]\}$

$C(\text{dom } f)$

$= \{\text{funciones } f : \mathbb{R}^n \rightarrow \mathbb{R}^m \text{ continuas en su dominio}\}$

Función Diferenciable

Caso $f : \mathbb{R} \rightarrow \mathbb{R}$

f es diferenciable en $x_0 \in (a, b)$ si $\lim_{x \rightarrow x_0} \frac{f(x) - f(x_0)}{x - x_0}$ existe y escribimos:

$$f^{(1)}(x_0) = \lim_{x \rightarrow x_0} \frac{f(x) - f(x_0)}{x - x_0}.$$

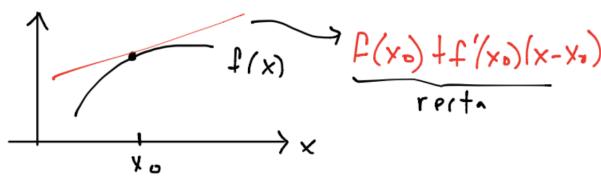
f es diferenciable en $[a, b]$ si es diferenciable en cada punto de $[a, b]$. Análogamente definiendo la variable $h = x - x_0$ se tiene:

$f^{(1)}(x_0) = \lim_{h \rightarrow 0} \frac{f(x_0 + h) - f(x_0)}{h}$ que típicamente se escribe como:

$$f^{(1)}(x) = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}.$$

💡 Comentario

Si f es diferenciable en x_0 entonces $f(x) \approx f(x_0) + f'(x_0)(x - x_0)$. Gráficamente:



Notación: $C^n([a, b])$

= {funciones $f : \mathbb{R}$

$\rightarrow \mathbb{R}$ con n derivadas continuas en el intervalo $[a, b]$ }

Ejemplo:

En Python podemos utilizar el paquete [SymPy](#) para calcular límites y derivadas:

```
import sympy
```

Límite de $\frac{\cos(x+h)-\cos(x)}{h}$ para $h \rightarrow 0$:

```
x, h = sympy.symbols("x, h")
```

```
quotient = (sympy.cos(x+h) - sympy.cos(x))/h
```

```
sympy.limit(quotient, h, 0)
```

Derivada de $\cos(x)$:

```
x = sympy.Symbol("x")
```

```
sympy.cos(x).diff(x)
```

Si queremos evaluar la derivada podemos usar:

```
sympy.cos(x).diff(x).subs(x, sympy.pi/2)
```

```
sympy.Derivative(sympy.cos(x), x).doit_numerically(sympy.pi/2)
```

Caso $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$

f es diferenciable en $x \in \text{intdom } f$ si existe una matriz $Df(x) \in \mathbb{R}^{m \times n}$ tal que:

$$\lim_{z \rightarrow x, z \neq x} \frac{\|f(z) - f(x) - Df(x)(z - x)\|_2}{\|z - x\|_2} = 0, z \in \text{dom } f$$

en este caso $Df(x)$ se llama la derivada de f en x .

💡 Observación

Sólo puede existir a lo más una matriz que satisfaga el límite anterior.

Comentarios:

- $Df(x)$ también es llamada la **Jacobiana** de f .
- Se dice que f es diferenciable si $\text{dom } f$ es abierto y es diferenciable en cada punto de $\text{dom } f$.
- La función: $f(x) + Df(x)(z - x)$ es afín y se le llama **aproximación de orden 1** de f en x (o también cerca de x). Para z cercana a x esta aproximación es cercana a $f(z)$.
- $Df(x)$ puede encontrarse con la definición de límite anterior o con las derivadas parciales:
$$Df(x)_{ij} = \frac{\partial f_i(x)}{\partial x_j}, i = 1, \dots, m, j = 1, \dots, n$$
 definidas como:

$$\frac{\partial f_i(x)}{\partial x_j} = \lim_{h \rightarrow 0} \frac{f_i(x + he_j) - f_i(x)}{h}$$

donde: $f_i : \mathbb{R}^n \rightarrow \mathbb{R}$, $i = 1, \dots, m$, $j = 1, \dots, n$ y e_j j -ésimo vector canónico que tiene un número 1 en la posición j y 0 en las entradas restantes.

- Si $f : \mathbb{R}^n \rightarrow \mathbb{R}$, $Df(x) \in \mathbb{R}^{1 \times n}$, su transpuesta se llama **gradiente**, el cual es un vector columna, se denota $\nabla f(x)$ y sus componentes son derivadas parciales:

$$\nabla f(x) = Df(x)^T = \begin{bmatrix} \frac{\partial f(x)}{\partial x_1} \\ \vdots \\ \frac{\partial f(x)}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \lim_{h \rightarrow 0} \frac{f(x + he_1) - f(x)}{h} \\ \vdots \\ \lim_{h \rightarrow 0} \frac{f(x + he_n) - f(x)}{h} \end{bmatrix} \in \mathbb{R}^{n \times 1}.$$

Comentarios

- En este contexto, la aproximación de primer orden a f en x es: $f(x) + \nabla f(x)^T(z - x)$ para z cercana a x .
- $C^n(\text{dom } f) = \{\text{funciones } f : \mathbb{R}^n \rightarrow \mathbb{R}^m \text{ con } n \text{ derivadas continuas en su dominio}\}$

Ejemplo:

$f : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ dada por:

$$f(x) = \begin{bmatrix} x_1 x_2 + x_2^2 \\ x_1^2 + 2x_1 x_2 + x_2^2 \end{bmatrix}$$

con $x = (x_1, x_2)^T$.

```
x1, x2 = sympy.symbols("x1, x2")
```

```
f1 = x1*x2 + x2**2
```

```
f1
```

```
f2 = x1**2 + x2**2 + 2*x1*x2
```

```
f2
```

Derivadas parciales:

Para $f_1(x) = x_1 x_2 + x_2^2$:

```
df1_x1 = f1.diff(x1)
```

```
df1_x1
```

```
df1_x2 = f1.diff(x2)
```

```
df1_x2
```

Para $f_2(x) = x_1^2 + 2x_1 x_2 + x_2^2$:

```
df2_x1 = f2.diff(x1)
```

```
df2_x1
```

```
df2_x2 = f2.diff(x2)
```

```
df2_x2
```

Derivada parcial de f_1 respecto a x_1 .

Derivada parcial de f_1 respecto a x_2 .

Derivada parcial de f_2 respecto a x_1 .

Derivada parcial de f_2 respecto a x_2 .

Entonces la derivada es:

$$Df(x) = \begin{bmatrix} x_2 & x_1 + 2x_2 \\ 2x_1 + 2x_2 & 2x_1 + 2x_2 \end{bmatrix}$$

Otra opción más fácil:

```
f = sympy.Matrix([f1, f2])
```

```
f
```

```
f.jacobian([x1, x2])
```

Para evaluar por ejemplo en $(x_1, x_2)^T = (0, 1)^T$:

```
d = f.jacobian([x1, x2])
```

```
d.subs([(x1, 0), (x2, 1)])
```

Regla de la cadena

Si $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ es diferenciable en $x \in \text{intdom } f$ y $g : \mathbb{R}^m \rightarrow \mathbb{R}^p$ es diferenciable en $f(x) \in \text{intdom } g$, se define la composición $h : \mathbb{R}^n \rightarrow \mathbb{R}^p$ por $h(z) = g(f(z))$, la cual es diferenciable en x , con derivada:

$$Dh(x) = Dg(f(x))Df(x) \in \mathbb{R}^{p \times n}.$$

Ejemplo: Sean $g : \mathbb{R} \rightarrow \mathbb{R}$, $f : \mathbb{R}^n \rightarrow \mathbb{R}$, $h : \mathbb{R}^n \rightarrow \mathbb{R}$ con $h(z) = g(f(z))$ entonces:

$$Dh(x) = Dg(f(x))Df(x) = \frac{dg(f(x))}{dx}(\nabla f(x))^T \in \mathbb{R}^{1 \times n}$$

y la transpuesta de $Dh(x)$ es: $\nabla h(x) = (Dh(x))^T = \frac{dg(f(x))}{dx}\nabla f(x) \in \mathbb{R}^{n \times 1}$.

Ejemplo:

1. $f(x) = \cos(x)$, $g(x) = \sin(x)$ por lo que $h(x) = \sin(\cos(x))$:

```
x = sympy.Symbol("x")
```

```
f = sympy.cos(x)
```

```
f
```

```
g = sympy.sin(x)
```

```
g
```

```
h = g.subs(x, f)
```

```
h
```

```
h.diff(x)
```

Otras formas:

```
g = sympy.sin
```

```
h = g(f)
```

```
h.diff(x)
```

```
h = sympy.sin(f)
```

```
h.diff(x)
```

1. $f(x) = x_1 + \frac{1}{x_2}$, $g(x) = e^x$ por lo que $h(x) = e^{x_1 + \frac{1}{x_2}}$:

```
x1, x2 = sympy.symbols("x1, x2")
```

```
f = x1 + 1/x2
```

```
f
```

```
g = sympy.exp
```

```
g
```

```
h = g(f)
```

```
h
```

```
h.diff(x1)
```

```
h.diff(x2)
```

Derivada parcial de f respecto a x_1 .

Derivada parcial de f respecto a x_2 .

Lo siguiente basado en [how-to-get-the-gradient-and-hessian-sympy](#):

```
from sympy.tensor.array import derive_by_array
```

```
derive_by_array(h, (x1, x2))
```

Caso particular:

Si $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, $f(x) = Ax + b$ con $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $g : \mathbb{R}^m \rightarrow \mathbb{R}^p$ y $h(z) = g(f(z)) = g(Az + b)$ con $\text{dom } h = \{x \in \mathbb{R}^n \mid Ax + b \in \text{dom } g\}$ entonces la derivada de h en x es:

$$Dh(x) = Dg(f(x))Df(x) = Dg(Ax + b)A$$

Obs: Si $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, $g : \mathbb{R}^m \rightarrow \mathbb{R}$, $h : \mathbb{R}^n \rightarrow \mathbb{R}$ entonces:

$$\nabla h(x) = (Dh(x))^T = A^T(Dg(Ax + b))^T = A^T \nabla g(Ax + b).$$

$$\in \mathbb{R}^{n \times 1}$$

Ejemplo:

1. Sean $g : \mathbb{R}^n \rightarrow \mathbb{R}$, $x, v \in \mathbb{R}^n$ y $\hat{g} : \mathbb{R} \rightarrow \mathbb{R}$ con $\hat{g}(t) = g(x + tv)$, esto es, \hat{g} es g restringida a la línea $\{x + tv \mid t \in \mathbb{R}\}$, entonces:

$$(\nabla \hat{g}(t))^T = D\hat{g}(t) = \hat{g}'(t) = (\nabla g(x + tv))^T v.$$

Comentario: el escalar $\hat{g}'(0) = (\nabla g(x))^T v$ se llama **derivada direccional** de g en x en la dirección v .

1. Sea $h : \mathbb{R}^n \rightarrow \mathbb{R}$, $h(x) = \log \left(\sum_{i=1}^m \exp(a_i^T x + b_i) \right)$ con $x \in \mathbb{R}^n$, $a_i \in \mathbb{R}^n \forall i = 1, \dots, m$ y

$b_i \in \mathbb{R} \forall i = 1, \dots, m$ entonces: $Dh(x) = Dg(f(x))Df(x)$ con $g : \mathbb{R}^m \rightarrow \mathbb{R}$ dada por

$g(y) = \log \left(\sum_{i=1}^m \exp(y_i) \right)$, $f(x) = Ax + b$, A y $h(x) = g(f(x))$. Entonces:

$$= (a_i)_{i=1}^m \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m.$$

$$Dh(x) = \left(\sum_{i=1}^m \exp(a_i^T x + b_i) \right)^{-1} \begin{bmatrix} \exp(a_1^T x + b_1) \\ \vdots \\ \exp(a_m^T x + b_m) \end{bmatrix} A \\ = (1^T z)^{-1} z^T A$$

donde: $z = \begin{bmatrix} \exp(a_1^T x + b_1) \\ \vdots \\ \exp(a_m^T x + b_m) \end{bmatrix}$. Por lo tanto $\nabla h(x) = (1^T z)^{-1} A^T z$.

Segunda derivada de una función $f : \mathbb{R}^n \rightarrow \mathbb{R}$.

Sea $f : \mathbb{R}^n \rightarrow \mathbb{R}$. La segunda derivada o matriz **Hessiana** de f en $x \in \text{intdom } f$ existe si f es dos veces diferenciable en x , se denota $\nabla^2 f(x)$ y sus componentes son segundas derivadas parciales:

$$\nabla^2 f(x) = \begin{bmatrix} \frac{\partial^2 f(x)}{\partial x_1^2} & \frac{\partial^2 f(x)}{\partial x_2 \partial x_1} & \cdots & \frac{\partial^2 f(x)}{\partial x_n \partial x_1} \\ \frac{\partial^2 f(x)}{\partial x_1 \partial x_2} & \frac{\partial^2 f(x)}{\partial x_2^2} & \cdots & \frac{\partial^2 f(x)}{\partial x_n \partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(x)}{\partial x_1 \partial x_n} & \frac{\partial^2 f(x)}{\partial x_2 \partial x_n} & \cdots & \frac{\partial^2 f(x)}{\partial x_n^2} \end{bmatrix}$$

Comentarios:

- La aproximación de segundo orden a f en x (o también para puntos cercanos a x) es la función cuadrática:

$$f(x) + (\nabla f(x))^T(z - x) + \frac{1}{2}(z - x)^T \nabla^2 f(x)(z - x)$$

la cual es una función de z .

- Se cumple:

$$\lim_{z \rightarrow x, z \neq x} \frac{|f(z) - [f(x) + (\nabla f(x))^T(z - x) + \frac{1}{2}(z - x)^T \nabla^2 f(x)(z - x)]|}{\|z - x\|_2} = 0, z \in \text{dom } f$$

- Se tiene lo siguiente:

- ∇f es una función llamada gradient mapping (o simplemente gradiente).
- $\nabla f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ tiene regla de correspondencia $\nabla f(x)$ (evaluar en x la matriz $(Df(\cdot))^T$).
- Si f es dos veces diferenciable entonces: $D\nabla f(x) = \nabla^2 f(x)$.
- **Importante:** si $f \in C^2(\text{dom } f)$ entonces la Hessiana es una matriz simétrica.

Regla de la cadena para la segunda derivada

Caso 1: Sean $f : \mathbb{R}^n \rightarrow \mathbb{R}$, $g : \mathbb{R} \rightarrow \mathbb{R}$, $h : \mathbb{R}^n \rightarrow \mathbb{R}$ con $h(x) = g(f(x))$, entonces: $\nabla^2 h(x) = D\nabla h(x)$ y $\nabla h(x) = (Dh(x))^T = (Dg(f(x))Df(x))^T = \frac{dg(x)}{dx} \nabla f(x)$ por lo que:

$$\begin{aligned} \nabla^2 h(x) &= D\nabla h(x) = D \left(\frac{dg(f(x))}{dx} \nabla f(x) \right) = \frac{dg(f(x))}{dx} \nabla^2 f(x) \\ &+ \left(\frac{d^2 g(f(x))}{dx^2} \nabla f(x) (\nabla f(x))^T \right)^T = \frac{dg(f(x))}{dx} \nabla^2 f(x) \\ &+ \frac{d^2 g(f(x))}{dx^2} \nabla f(x) (\nabla f(x))^T \end{aligned}$$

Caso 2: Sean $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, $f(x) = Ax + b$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ y $h(x) = g(f(x))$ con $g : \mathbb{R}^m \rightarrow \mathbb{R}^p$, $h : \mathbb{R}^n \rightarrow \mathbb{R}^p$, entonces:

$$\begin{aligned} \nabla h(x) &= (Dh(x))^T = (Dg(f(x))Df(x))^T \\ &= (Dg(Ax + b)A)^T = A^T Dg(Ax + b). \end{aligned}$$

Obs: si $p = 1$ se tiene: $\nabla^2 h(x) = D\nabla h(x) = A^T \nabla^2 g(Ax + b)A$.

Ejemplos:

- Sea $\hat{g}(t) = g(x + tv)$ con $x, v \in \mathbb{R}^n$, $t \in \mathbb{R}$ y $g : \mathbb{R}^n \rightarrow \mathbb{R}$, $\hat{g} : \mathbb{R} \rightarrow \mathbb{R}$ es decir, \hat{g} es g restringida a la línea $\{x + tv | t \in \mathbb{R}\}$, entonces:

$$(\nabla \hat{g}(t))^T = D\hat{g}(t) = \frac{\hat{g}(t)}{dt} = Dg(x + tv)v = \nabla g(x + tv)^T v$$

y:

$$\nabla^2 \hat{g}(t) = \frac{d^2 \hat{g}(t)}{dt^2} = D\nabla g(x + tv)v = v^T \nabla^2 g(x + tv)v.$$

- Sean $h : \mathbb{R}^n \rightarrow \mathbb{R}$, $h(x) = \log\left(\sum_{i=1}^m \exp(a_i^T x + b_i)\right)$ con $x \in \mathbb{R}^n$, $a_i \in \mathbb{R}^n \forall i = 1, \dots, m$ y $b_i \in \mathbb{R} \forall i = 1, \dots, m$. Entonces: $Dh(x) = Dg(f(x))Df(x)$ con $g(y) = \log\left(\sum_{i=1}^m \exp(y_i)\right)$, $f(x) = Ax + b$, A .
 $= (a_i)_{i=1}^m \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $h(x) = g(f(x))$

Como se desarrolló anteriormente $\nabla h(x) = (1^T z)^{-1} A^T z$ con $z = \begin{bmatrix} \exp(a_1^T x + b_1) \\ \vdots \\ \exp(a_m^T x + b_m) \end{bmatrix}$.

$\therefore \nabla^2 h(x) = D\nabla h(x) = A^T \nabla^2 g(Ax + b)A$ (por caso 2)
donde: $\nabla^2 g(y) = (1^T y)^{-1} \text{diag}(y) - (1^T y)^{-2} yy^T$ (por caso 1 tomando $\log : \mathbb{R} \rightarrow \mathbb{R}$, $\sum \exp : \mathbb{R}^m \rightarrow \mathbb{R}$)

$$\therefore \nabla^2 h(x) = A^T [(1^T z)^{-1} \text{diag}(z) - (1^T z)^{-2} zz^T] A$$

con $\text{diag}(c)$ matriz diagonal con elementos en su diagonal iguales a las entradas del vector c .

Referencias

1. S. P. Boyd, L. Vandenberghe, Convex Optimization. Cambridge University Press, 2004.

Notas para contenedor de docker:

Comando de docker para ejecución de la nota de forma local:

nota: cambiar `<ruta a mi directorio>:/datos` por la ruta de directorio que se desea mapear a `/datos` dentro del contenedor de docker.

```
docker run --rm -v <ruta a mi directorio>:/datos --name jupyterlab_optimizacion -p 8888:8888 -d palmoreck/jupyterlab_optimizacion:2.1.4
```

password para jupyterlab: `qwerty`

Detener el contenedor de docker:

```
docker stop jupyterlab_optimizacion
```

Documentación de la imagen de docker `palmoreck/jupyterlab_optimizacion:2.1.4` en [liga](#).

Nota generada a partir de la [liga1](#), [liga2](#) e inicio de [liga3](#).

1.6 Polinomios de Taylor y diferenciación numérica

Problema: ¿Cómo aproximar una función f en un punto x_1 ?

Si f es continuamente diferenciable en x_0 y $f^{(1)}, f^{(2)}$ existen y están acotadas en x_0 entonces:

$$f(x_1) \approx f(x_0) + f^{(1)}(x_0)(x_1 - x_0)$$

y se llama **aproximación de orden 1**. Ver [Definición de función, derivada de una función](#) para definición de continuidad, diferenciabilidad y propiedades

Obs: obsérvese que lo anterior requiere de los valores: $x_0, x_1, f(x_0), f^{(1)}(x_0)$. Esta aproximación tiene un error de **orden 2** pues su error es **proporcional** al cuadrado del ancho del intervalo: $h = x_1 - x_0$, esto es, si reducimos a la mitad h entonces el error se reduce en una cuarta parte.

Otra aproximación más simple sería:

$$f(x_1) \approx f(x_0)$$

lo cual sólo requiere del conocimiento de $f(x_0)$ y se llama **aproximación de orden 0**, sin embargo esta aproximación tiene un error de **orden 1** pues este es proporcional a h , esto es, al reducir a la mitad h se reduce a la mitad el error.

Estos errores los llamamos errores por **truncamiento**. Utilizamos la notación “O grande” $\mathcal{O}(\cdot)$ para escribir lo anterior:

$$f(x) - f(x_0) = \mathcal{O}(h)$$

con la variable $h = x - x_0$. Análogamente:

$$f(x) - (f(x_0) + f^{(1)}(x_0)(x - x_0)) = \mathcal{O}(h^2).$$

Obs: no confundir órdenes de una aproximación con órdenes de error.

Otras aproximaciones a una función se pueden realizar con:

- Interpoladores polinomiales (representación por Vandermonde, Newton, Lagrange).

Aproximación a una función por el teorema de Taylor

En esta sección se presenta el teorema de Taylor, el cual, bajo ciertas hipótesis nos proporciona una expansión de una función alrededor de un punto. Este teorema será utilizado en **diferenciación e integración numérica**. El teorema es el siguiente:

Sea $f : \mathbb{R} \rightarrow \mathbb{R}$, $f \in C^n([a, b])$ tal que $f^{(n+1)}$ existe en $[a, b]$. Si $x_0 \in [a, b]$ entonces $\forall x \in [a, b]$ se tiene:

$f(x) = P_n(x) + R_n(x)$ donde:

$$P_n(x) = \sum_{k=0}^n \frac{f^{(k)}(x_0)(x - x_0)^k}{k!} \quad (f^{(0)} = f)$$

Comentarios:

- El teorema de Taylor nos dice que cualquier función suave (función en C^n) se le puede aproximar por un polinomio en el intervalo $[a, b]$, de hecho $f(x) \approx P_n(x)$.
- El teorema de Taylor es una generalización del [teorema del valor medio para derivadas](#)
- $P_n(x)$ se le llama polinomio de Taylor alrededor de x_0 de orden n y $R_n(x)$ es llamado residuo de Taylor alrededor de x_0 de orden $n + 1$, tiene otras expresiones para representarlo y la que se utiliza en el enunciado anterior es en su forma de Lagrange (ver [liga](#) para otras expresiones del residuo).
- ξ_x es un punto entre x_0, x desconocido y está en función de x (por eso se le escribe un subíndice).
- Una forma del teorema de Taylor es escribirlo definiendo a la variable $h = x - x_0$:

$$\begin{aligned} f(x) &= f(x_0 + h) = P_n(h) + R_n(h) \\ &= \sum_{k=0}^n \frac{f^{(k)}(x_0)h^k}{k!} + \frac{f^{(n+1)}(\xi_h)h^{n+1}}{(n+1)!} \end{aligned}$$

y si $f^{(n+1)}$ es acotada, escribimos: $R_n(h) = \mathcal{O}(h^{n+1})$.

Ejemplo: graficar la función y los polinomios de Taylor constante, lineal y cuadrático en una sola gráfica con **ggplot2** en el intervalo $[1, 2]$ para la función $\frac{1}{x}$ con centro en $x_0 = 1.5$. ¿Cuánto es la aproximación de los polinomios en $x=1.9$? (para esta pregunta calcula el error relativo de tus aproximaciones).

Obtengamos los polinomios de Taylor de orden n con $n \in \{0, 1, 2\}$ y centro en $x_0 = 1.5$ para la función $\frac{1}{x}$ en el intervalo $[1, 2]$. Los polinomios de Taylor son:

$$\begin{aligned} P_0(x) &= f(x_0) = \frac{2}{3} \quad (\text{constante}) \\ P_1(x) &= f(x_0) + f^{(1)}(x_0)(x - x_0) = \frac{2}{3} - \frac{1}{x_0^2}(x - x_0) \\ &\quad (\text{lineal}) \\ P_2(x) &= f(x_0) + f^{(1)}(x_0)(x - x_0) + \frac{f^{(2)}(x_0)(x - x_0)^2}{2} = \frac{2}{3} \\ &\quad - \frac{1}{x_0^2}(x - x_0) + \frac{2}{x_0^3}(x - x_0)^2 \quad (\text{cuadrático}) \end{aligned}$$

```
library(ggplot2)
```

```
options(repr.plot.width=6, repr.plot.height=6) #esta línea sólo se ejecuta para jupyterlab con R
```

```

Aprox_Taylor <- function(x,centro,n){
  length_x = length(x)
  evaluacion = rep(0,length_x)
  for(j in 1:length_x){
    constante = centro^(-1)
    evaluacion[j] = constante
    for(k in 1:n)
      constante = -1*centro^(-1)*(x[j]-centro)*constante
      evaluacion[j] = evaluacion[j]+(k+1)*constante
  }
  evaluacion
}

```

```

x0<-1.5
x<- seq(from=1,to=2,by=.005)
n<-c(0,1,2)
f<-function(z)1/z
y<-f(x)
y_Taylor_0<-1/x0*rep(1,length(x))
y_Taylor_1<-Aprox_Taylor(x,x0,1)
y_Taylor_2<-Aprox_Taylor(x,x0,2)

```

```
gg <- ggplot()
```

```

gg+
geom_line(aes(x=x,y=y,color='f(x)')) +
geom_line(aes(x=x,y=y_Taylor_0,color='constante'))+
geom_line(aes(x=x,y=y_Taylor_1,color='lineal')) +
geom_line(aes(x=x,y=y_Taylor_2,color='cuadratico')) +
geom_point(aes(x=x0, y=f(x0)), color='blue', size=3)

```

```

err_relativo<-function(aprox,obj){
  abs(aprox-obj)/abs(obj)
}

```

```

x_obj=1.9
f_x_obj=f(x_obj)
print('error relativo polinomio constante')
err_relativo(1/x0,f_x_obj)
print('error relativo polinomio lineal')
err_relativo(Aprox_Taylor(x_obj,x0,1),f_x_obj)
print('error relativo polinomio cuadrático')
err_relativo(Aprox_Taylor(x_obj,x0,2),f_x_obj)

```

```
[1] "error relativo polinomio constante"
```

```
[1] "error relativo polinomio lineal"
```

```
[1] "error relativo polinomio cuadrático"
```

Pregunta: ¿por qué se tiene mejor aproximación con un polinomio constante o lineal que con un polinomio cuadrático? ¿no deberíamos tener menor error al añadir más y más términos en el polinomio de Taylor? ¿de qué depende tener una buena aproximación al utilizar polinomios de Taylor para una función f en general?.

Ejercicio: Aproximar $f(1)$ con polinomios de Taylor de orden 0, 1, 2, 3, 4 si

$f(x) = -0.1x^4 - 0.15x^3 - 0.5x^2 - 0.25x + 1.2$ con centro en $x_0 = 0$. Calcula errores relativos de tus aproximaciones. Realiza las gráficas de cada polinomio en el intervalo $[0, 1]$ con `ggplot2`. Observa que $R_5(x)$ es cero.

Teorema de Taylor para una función $f : \mathbb{R}^n \rightarrow \mathbb{R}$

Sea $f : \mathbb{R}^n \rightarrow \mathbb{R}$ diferenciable en $\text{dom } f$. Si $x_0, x \in \text{dom } f$ y $x_0 + t(x - x_0) \in \text{dom } f, \forall t \in (0, 1)$, entonces $\forall x \in \text{dom } f$ se tiene $f(x) = P_0(x) + R_0(x)$ donde:

$$P_0(x) = f(x_0)$$

$$R_0(x) = \nabla f(x_0 + t_x(x - x_0))^T(x - x_0)$$

para alguna $t_x \in (0, 1)$ y $\nabla f(\cdot)$ gradiente de f (ver al final de esta nota la definición de gradiente). Esta se llama **aproximación de orden 0** para f con centro en x_0 . Si $\nabla f(\cdot)$ es acotado en $\text{dom } f$ entonces se escribe:

$$R_0(x) = \mathcal{O}(\|x - x_0\|).$$

Si además f es continuamente diferenciable en $\text{dom } f$ (su derivada es continua, ver al final de esta nota definición de continuidad), $f^{(2)}$ existe en $\text{dom } f$, se tiene $f(x) = P_1(x) + R_1(x)$ donde:

$$P_1(x) = f(x_0) + \nabla f(x_0)^T(x - x_0)$$

$$R_1(x) = \frac{1}{2}(x - x_0)^T \nabla^2 f(x_0 + t_x(x - x_0))(x - x_0)$$

para alguna $t_x \in (0, 1)$ y $\nabla^2 f(\cdot)$ Hessiana de f (ver al final de esta nota definición de la matriz Hessiana). Esta se llama **aproximación de orden 1** para f con centro en x_0 . Si $\nabla^2 f(\cdot)$ es acotada en $\text{dom } f$ entonces se escribe:
 $R_1(x) = \mathcal{O}(\|x - x_0\|^2)$.

Si $f^{(2)}$ es continuamente diferenciable y $f^{(3)}$ existe y es acotada en $\text{dom } f$, se tiene $f(x) = P_2(x) + R_2(x)$ donde:

$$P_2(x) = f(x_0) + \nabla f(x_0)^T(x - x_0)$$

$$+ \frac{1}{2}(x - x_0)^T \nabla^2 f(x_0)(x - x_0)$$

$$R_2(x) = \mathcal{O}(\|x - x_0\|^3).$$

Esta se llama **aproximación de orden 2** para f con centro en x_0 .

Obs: en este caso $f^{(3)}$ es un tensor.

Comentarios:

- Tomando $h = x - x_0$, se reescribe el teorema como sigue, por ejemplo para la aproximación de orden 1 incluyendo su residuo:

$$f(x) = f(x_0 + h) = \underbrace{f(x_0) + \nabla f(x_0)^T h}_{P_1(h)} + \underbrace{\frac{1}{2}h^T \nabla^2 f(x_0 + t_x h)h}_{R_1(h)}$$

Si $f^{(2)}$ es acotada en $\text{dom } f$, escribimos: $R_1(h) = \mathcal{O}(\|h\|^2)$.

Diferenciación numérica por diferencias finitas

Las fórmulas de aproximación a las derivadas por diferencias finitas pueden obtenerse con los polinomios de Taylor, presentes en el teorema del mismo autor, por ejemplo:

Sea $f \in C^1([a, b])$ y $f^{(2)}$ existe y está acotada $\forall x \in [a, b]$ entonces, si $x + h \in [a, b]$ con $h > 0$ por el teorema de Taylor:

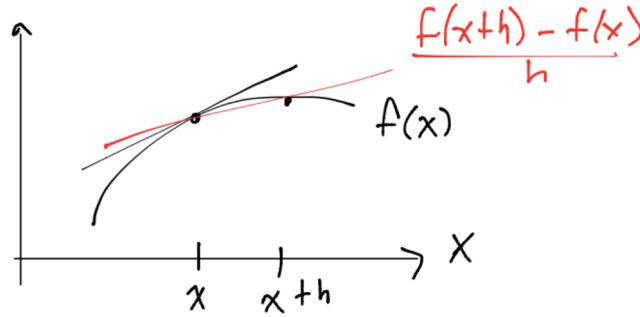
$$f(x + h) = f(x) + f^{(1)}(x)h + f^{(2)}(\xi_{x+h}) \frac{h^2}{2}$$

$$f^{(1)}(x) = \frac{f(x + h) - f(x)}{h} - f^{(2)}(\xi_{x+h}) \frac{h}{2}.$$

y escribimos:

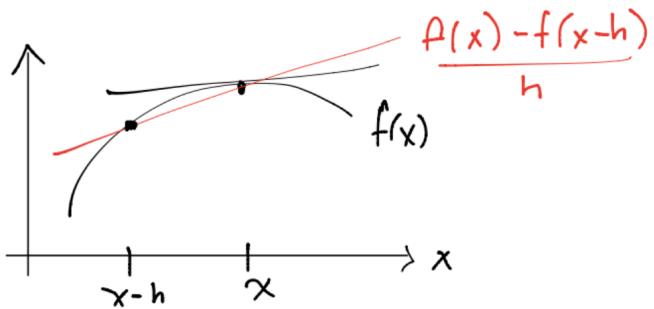
$$f^{(1)}(x) = \frac{f(x + h) - f(x)}{h} + \mathcal{O}(h).$$

La aproximación $\frac{f(x+h) - f(x)}{h}$ es una fórmula por diferencias hacia delante con error de orden 1. Gráficamente se tiene:



Con las mismas suposiciones es posible obtener la fórmula para la aproximación por diferencias hacia atrás:

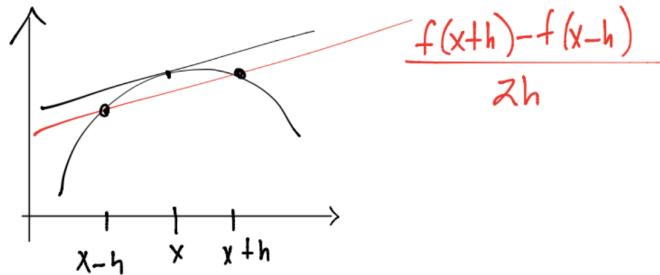
$$f^{(1)}(x) = \frac{f(x) - f(x - h)}{h} + \mathcal{O}(h), h > 0.$$



Considerando $f \in C^2([a, b])$, $f^{(3)}$ existe y está acotada $\forall x \in [a, b]$ si $x - h, x + h \in [a, b]$ y $h > 0$ entonces:

$$f^{(1)}(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2), h > 0.$$

y el cociente $\frac{f(x+h)-f(x-h)}{2h}$ es la aproximación por diferencias centradas con error de orden 2. Gráficamente:



Obs: la aproximación por diferencias finitas a la primera derivada de la función tiene un error de orden $\mathcal{O}(h)$ por lo que una elección de h igual a $.1 = 10^{-1}$ generará aproximaciones con alrededor de un dígito correcto.

Obs: la diferenciación numérica por diferencias finitas no es un proceso con una alta exactitud pues los problemas del redondeo de la aritmética en la máquina se hacen presentes en el mismo (ver nota [Sistema de punto flotante](#)). Como ejemplo de esta situación realicemos el siguiente ejemplo.

Ejemplo: realizar una gráfica de $\log(\text{error relativo})$ vs $\log(h)$ (h en el eje horizontal) con [ggplot2](#) para aproximar la primera derivada de $f(x) = e^{-x}$ en $x = 1$ con $h \in \{10^{-16}, 10^{-14}, \dots, 10^{-1}\}$ y diferencias hacia delante. Valor a aproximar: $f^{(1)}(1) = -e^{-1}$

```

f<-function(x){
  exp(-x)
}
aprox_1a_derivada<-function(f,x,h){
  (f(x+h)-f(x))/h
}

df<-function(x){
  -exp(-x)
}

err_absoluto<-function(aprox,obj){
  abs(aprox-obj)
}

err_relativo<-function(aprox,obj){
  abs(aprox-obj)/abs(obj)
}

x<-1

h<-10^(-1*(1:16))

err_absoluto_res<-err_absoluto(aprox_1a_derivada(f,x,h),df(x))
err_relativo_res<-err_relativo(aprox_1a_derivada(f,x,h),df(x))

gf<-ggplot()

```

```

gf+
  geom_line(aes(x=log(h),y=log(err_absoluto_res)))+
  ggtitle('Aproximación a la primera derivada por diferencias finitas')

```

```

gf+
  geom_line(aes(x=log(h),y=log(err_relativo_res)))+
  ggtitle('Aproximación a la primera derivada por diferencias finitas')

```

Ejercicio: realizar una gráfica de $\log(\text{error relativo})$ vs $\log(h)$ (h en el eje horizontal) con `ggplot2` para aproximar la segunda derivada de $f(x) = e^{-x}$ en $x = 1$ con $h \in \{10^{-16}, 10^{-14}, \dots, 10^{-1}\}$ y diferencias hacia delante. Valor a aproximar: $f''(1) = e^{-1}$. Usar:

$$\frac{d^2 f(x)}{dx^2} = \frac{f(x+2h) - 2f(x+h) + f(x)}{h^2} + \mathcal{O}(h)$$

Encontrar valor(es) de h que minimiza(n) al error absoluto y relativo.

La aproximación anterior es una aproximación a la segunda derivada por diferencias hacia delante. Las versiones hacia atrás y centradas en general para una función $f : \mathbb{R} \rightarrow \mathbb{R}$ son respectivamente:

$$\begin{aligned}\frac{d^2 f(x)}{dx^2} &\approx \frac{f(x) - 2f(x-h) + f(x-2h)}{h^2} + \mathcal{O}(h) \\ \frac{d^2 f(x)}{dx^2} &\approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + \mathcal{O}(h^2)\end{aligned}$$

Análisis del error por redondeo y truncamiento en aproximación por diferencias finitas hacia delante

El ejemplo anterior muestra (vía una gráfica) que el método numérico de diferenciación numérica no es estable numéricamente respecto al redondeo (ver nota [Condición de un problema y estabilidad de un algoritmo](#) para definición de estabilidad de un algoritmo) y también se puede corroborar realizando un análisis del error. En esta sección consideraremos la aproximación a la primer derivada por diferencias finitas hacia delante:

$$\frac{f(x+h) - f(x)}{h}$$

Suponemos que $\hat{f}(x)$ aproxima a $f(x)$ y por errores de redondeo $\hat{f}(x) = f(x)(1 + \epsilon_{f(x)})$ con $|\epsilon_{f(x)}| \leq \epsilon_{mag}$ error de redondeo al evaluar f en x . $\hat{f}(x)$ es la aproximación en un SPFN (ver nota [Sistema de punto flotante](#)). Además supóngase que $x, x+h, h \in \mathcal{F}$. Entonces en la aproximación a la primer derivada por diferencias hacia delante:

$$f^{(1)}(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h) \text{ y calculando el error absoluto:}$$

$$\begin{aligned}\text{ErrAbs}\left(\frac{\hat{f}(x+h) - \hat{f}(x)}{h}\right) &= \left|f^{(1)}(x) - \frac{\hat{f}(x+h) - \hat{f}(x)}{h}\right| \\ &= \left|\frac{f(x+h) - f(x)}{h} + \mathcal{O}(h) - \left(\frac{f(x+h)(1 + \epsilon_{f(x+h)}) - f(x)(1 + \epsilon_{f(x)})}{h}\right)\right| \\ &= \left|\mathcal{O}(h) - \frac{f(x+h)\epsilon_{f(x+h)} - f(x)\epsilon_{f(x)}}{h}\right| \\ &\leq \mathcal{O}(h) + \frac{C\epsilon_{mag}}{h}\end{aligned}$$

suponiendo en el último paso que $|f(x+h)\epsilon_{f(x+h)} - f(x)\epsilon_{f(x)}| \leq C\epsilon_{mag}$ con $C > 0$ constante que acota a la función f en el intervalo $[a, b]$. Obsérvese que $\frac{\hat{f}(x+h) - \hat{f}(x)}{h}$ es la aproximación a la primer derivada por diferencias hacia delante que se obtiene en la computadora, por lo que la cantidad $\left|f^{(1)}(x) - \frac{\hat{f}(x+h) - \hat{f}(x)}{h}\right|$ es el error absoluto de la aproximación por diferencias hacia delante.

El error relativo es:

$$\begin{aligned}\text{ErrRel}\left(\frac{\hat{f}(x+h) - \hat{f}(x)}{h}\right) &= \frac{\text{ErrAbs}\left(\frac{\hat{f}(x+h) - \hat{f}(x)}{h}\right)}{|f^{(1)}(x)|} \\ &\leq \frac{\mathcal{O}(h) + \frac{C\epsilon_{mag}}{h}}{|f^{(1)}(x)|} = K_1 h + K_2 \frac{1}{h}\end{aligned}$$

con $K_1, K_2 > 0$ constantes.

Entonces la función $g(h) = \mathcal{O}(h) + \mathcal{O}\left(\frac{1}{h}\right)$ acota al error absoluto y al error relativo y se tiene:

- Si $h \rightarrow 0$ la componente $\mathcal{O}\left(\frac{1}{h}\right)$ domina a la componente $\mathcal{O}(h)$, la cual tiende a 0.
- Si $h \rightarrow \infty$ la componente $\mathcal{O}(h)$ domina a $\mathcal{O}\left(\frac{1}{h}\right)$, la cual tiende a 0.

Por las dos observaciones anteriores, existe un valor de h que minimiza a los errores. Tal valor se observa en las gráficas anteriores y es igual a:

```
print(c(h[which.min(err_absoluto_res)],h[which.min(err_relativo_res)]))
```

[1] 1e-08 1e-08

Ejercicio: obtener de forma analítica el valor de h que minimiza la función $g(h)$ anterior. Tip: utilizar criterio de primera y segunda derivada para encontrar mínimo global.

Conclusiones y comentarios:

- La componente $\mathcal{O}(h)$ es el error por truncamiento, la cual resulta del teorema de Taylor. El teorema de Taylor nos indica que añadir términos en el polinomio de Taylor si la x a aproximar es cercana al centro, las derivadas de f son acotadas y $h \rightarrow 0$ entonces el error por truncamiento debe tender a 0. Lo anterior no ocurre en la implementación numérica (corroborado de forma analítica y visual) del método por diferenciación numérica para la primera derivada por la presencia de la componente $\mathcal{O}(\frac{1}{h})$ en los errores. Tal componente proviene del error por redondeo.
- Obsérvese que el error relativo máximo es del 100% lo que indica que no se tiene ninguna cifra correcta en la aproximación:

```
max(err_relativo_res)
```

y esto ocurre para un valor de h igual a:

```
h[which.max(err_relativo_res)]
```

pregunta: ¿por qué se alcanza el máximo error relativo en el valor de $h = 10^{-16}$?

Con esto se tiene que la diferenciación numérica es un método **inestable numéricamente respecto al redondeo**. Ver nota [Condición de un problema y estabilidad de un algoritmo](#).

- Un análisis de error similar se utiliza para el método de diferencias finitas por diferencias centradas para aproximar la primera derivada. En este caso el valor de h que minimiza a los errores es del orden $h^* = 10^{-6}$.

Diferenciación numérica para una función $f : \mathbb{R}^n \rightarrow \mathbb{R}$

Supongamos f es dos veces diferenciable en $\text{dom } f$. Si $f : \mathbb{R}^n \rightarrow \mathbb{R}$ entonces su derivada se llama **gradiente**, el cual es una función $\nabla f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ y su segunda derivada se llama **Hessiana**, la cual es una función $f' : \mathbb{R}^n \rightarrow \mathbb{R}^{n \times n}$ (ver [Definición de función, derivada de una función](#) para definición de derivadas en funciones $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$). Ambas funciones al evaluarse resultan en un vector en \mathbb{R}^n y en una matriz en $\mathbb{R}^{n \times n}$ respectivamente.

Podemos utilizar las fórmulas de aproximación en diferenciación numérica con diferencias finitas para el caso $f : \mathbb{R} \rightarrow \mathbb{R}$ revisadas anteriormente para aproximar al gradiente y a la Hessiana.

Para el caso del gradiente se tiene por **diferenciación hacia delante**:

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f(x)}{\partial x_1} \\ \vdots \\ \frac{\partial f(x)}{\partial x_n} \end{bmatrix} \approx \nabla \hat{f}(x) = \begin{bmatrix} \frac{f(x+he_1)-f(x)}{h} \\ \vdots \\ \frac{f(x+he_n)-f(x)}{h} \end{bmatrix} \in \mathbb{R}^n$$

con e_j j -ésimo vector canónico que tiene un número 1 en la posición j y 0 en las entradas restantes para $j = 1, \dots, n$. Se cumple $\|\nabla f(x) - \nabla \hat{f}(x)\| = \mathcal{O}(h)$. Y para el caso de la Hessiana:

$$\nabla^2 f(x) = \begin{bmatrix} \frac{\partial^2 f(x)}{\partial x_1^2} & \frac{\partial^2 f(x)}{\partial x_2 \partial x_1} & \cdots & \frac{\partial^2 f(x)}{\partial x_n \partial x_1} \\ \frac{\partial^2 f(x)}{\partial x_1 \partial x_2} & \frac{\partial^2 f(x)}{\partial x_2^2} & \cdots & \frac{\partial^2 f(x)}{\partial x_n \partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(x)}{\partial x_1 \partial x_n} & \frac{\partial^2 f(x)}{\partial x_2 \partial x_n} & \cdots & \frac{\partial^2 f(x)}{\partial x_n^2} \end{bmatrix},$$

$$\nabla^2 \hat{f}(x) = \begin{bmatrix} \frac{f(x+2he_1)-2f(x+he_1)+f(x)}{h^2} & \frac{f(x+he_1+he_2)-f(x+he_1)-f(x+he_2)+f(x)}{h^2} & \dots & \frac{f(x+he_1+he_n)-f(x+he_1)-f(x+he_n)+f(x)}{h^2} \\ \frac{f(x+he_1+he_2)-f(x+he_2)-f(x+he_1)+f(x)}{h^2} & \frac{f(x+2he_2)-2f(x+he_2)+f(x)}{h^2} & \dots & \frac{f(x+he_2+he_n)-f(x+he_2)-f(x+he_n)+f(x)}{h^2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{f(x+he_1+he_n)-f(x+he_n)-f(x+he_1)+f(x)}{h^2} & \frac{f(x+he_2+he_n)-f(x+he_n)-f(x+he_2)+f(x)}{h^2} & \dots & \frac{f(x+2he_n)-2f(x+he_n)+f(x)}{h^2} \end{bmatrix}$$

Se cumple: $\|\nabla^2 f(x) - \nabla \hat{f}^2(x)\| = \mathcal{O}(h)$.

Ejemplo:

Aproximar $\nabla f(x), \nabla^2 f(x)$ con diferencias hacia delante y $h \in \{10^{-16}, 10^{-14}, \dots, 10^{-1}\}$ para $f : \mathbb{R}^4 \rightarrow \mathbb{R}$, dada por $f(x) = (x_1^2 - x_2^2)^2 + x_1^2 + (x_3^2 - x_4^2)^2 + x_3^2$ en el punto $x_0 = (1.5, 1.5, 1.5, 1.5)^T$. Realizar una gráfica de $\log(\text{Err_rel})$ vs $\log(h)$

Para esta función se tiene:

$$\begin{aligned} \nabla f(x) &= \begin{bmatrix} 4x_1(x_1^2 - x_2^2) + 2x_1 \\ -4x_2(x_1^2 - x_2^2) \\ 4x_3(x_3^2 - x_4^2) + 2x_3 \\ -4x_4(x_3^2 - x_4^2) \end{bmatrix}, \\ &= \begin{bmatrix} 12x_1^2 - 4x_2^2 + 2 & -8x_1x_2 & 0 & 0 \\ -8x_1x_2 & -4x_1^2 + 12x_2^2 & 0 & 0 \\ 0 & 0 & 12x_3^2 - 4x_4^2 + 2 & -8x_3x_4 \\ 0 & 0 & -8x_3x_4 & -4x_3^2 + 12x_4^2 \end{bmatrix} \end{aligned}$$

Y evaluando en x_0 :

```
#gradiente de f calculado de forma simbólica
gf<-function(x){
  c(4*x[1]^(x[1]^2-x[2]^2)+2*x[1],
    -4*x[2]^(x[1]^2-x[2]^2),
    4*x[3]^(x[3]^2-x[4]^2)+2*x[3],
    -4*x[4]^(x[3]^2-x[4]^2))
}
```

```
x_0<-c(1.5,1.5,1.5,1.5)
```

```
print(gf(x_0))
```

```
[1] 3 0 3 0
```

$$\nabla f(x_0) = \begin{bmatrix} 3 \\ 0 \\ 3 \\ 0 \end{bmatrix},$$

para la Hessiana:

```
#Hessiana de f calculada de forma simbólica
gf2<-function(x){
  matrix(c(12*x[1]^2-4*x[2]^2,-8*x[1]*x[2],0,0,
         -8*x[1]*x[2],-4*x[1]^2+12*x[2]^2,0,0,
         0,0,12*x[3]^2-4*x[4]^2+2,-8*x[3]*x[4],
         0,0,-8*x[3]*x[4],-4*x[3]^2+12*x[4]^2),nrow=4,ncol=4)
}
```

```
print(gf2(x_0))
```

[,1]	[,2]	[,3]	[,4]
[1,]	20	-18	0
[2,]	-18	18	0
[3,]	0	0	20 -18
[4,]	0	0	-18 18

$$\nabla^2 f(x_0) = \begin{bmatrix} 20 & -18 & 0 & 0 \\ -18 & 18 & 0 & 0 \\ 0 & 0 & 20 & -18 \\ 0 & 0 & -18 & 18 \end{bmatrix}$$

Calculando el gradiente y la Hessiana de forma numérica con la aproximación por diferencias hacia delante:

```
#definición de función y punto en el que se calculan las aproximaciones
f_ej<-function(x){
  (x[1]^2-x[2]^2)^2+x[1]^2+(x[3]^2-x[4]^2)^2+x[3]^2
}
x0<-rep(1.5,4)
```

```
#Función auxiliar para incrementar una entrada de un vector
#por una cantidad h
inc_index<-function(vec,index,h){
  vec[index]<-vec[index]+h
  vec}
```

```
#Función para aproximar el gradiente de f
#f: función a la que se le aproximarán su gradiente
#x: punto en el que se aproximarán el gradiente de f
#h: valor de h para diferencias hacia delante
#gf: aproximación al gradiente de f por diferencias hacia delante
gf_numeric<-function(f,x,h){
  n<-length(x)
  gf<-rep(0,n)
  for(i in 1:n){
    gf[i]=(f(inc_index(x,i,h))-f(x))
  }
  gf/h
}
```

```
#Función para aproximar la Hessiana de f
#f: función a la que se le aproximarán su Hessiana
#x: punto en el que se aproximarán la Hessiana de f
#h: valor de h para diferencias hacia delante
#Hf: aproximación a la Hessiana de f por diferencias hacia delante
gf2_numeric<-function(f,x,h){
  n<-length(x)
  Hf<-matrix(rep(0,n^2),nrow=n,ncol=n)
  f_x<-f(x)
  for(i in 1:n){
    x_inc_in_i<-inc_index(x,i,h)
    f_x_inc_in_i<-f(x_inc_in_i)
    for(j in i:n{
      #otra forma para dif:
      #dif<-f(inc_index(inc_index(x,j,h),i,h))-f_x_inc_in_i-
      f(inc_index(x,j,h))+f_x
      dif<-f(inc_index(x_inc_in_i,j,h))-f_x_inc_in_i-f(inc_index(x,j,h))+f_x
      Hf[i,j]<-dif
      if(j!=i)
        Hf[j,i]<-dif
    }
  }
  Hf/h^2
}
```

```
h<-10^(-1*(1:16))#un conjunto de valores de h para diferencias hacia delante
err_absoluto<-function(aprox,obj){
  norm(aprox-obj,"2") #cálculo de norma vectorial o matricial, depende de lo que
recibe
  #la función en sus parámetros aprox y obj
}

err_relativo<-function(aprox,obj){
  norm(aprox-obj,"2")/norm(obj,"2")#cálculo de norma vectorial o matricial, depende
de lo que recibe
  #la función en sus parámetros aprox y obj
}

gf_numeric_list<-lapply(h,gf_numeric,f=f_ej,x=x0)
gf2_numeric_list<-lapply(h,gf2_numeric,f=f_ej,x=x0)

err_absoluto_gf_res<-sapply(gf_numeric_list,err_absoluto,obj=gf(x_0))
err_relativo_gf_res<-sapply(gf_numeric_list,err_relativo,obj=gf(x_0))

err_absoluto_gf2_res<-sapply(gf2_numeric_list,err_absoluto,obj=gf2(x_0))
err_relativo_gf2_res<-sapply(gf2_numeric_list,err_relativo,obj=gf2(x_0))
```

```
gg<-ggplot()
```

```
gg+
geom_line(aes(x=log(h),y=log(err_relativo_gf_res)))+
ggtitle('Aproximación al gradiente por diferencias finitas')
```

```
print(c(h[which.min(err_relativo_gf_res)],h[which.min(err_absoluto_gf_res)]))
```

```
[1] 1e-08 1e-08
```

```
gg+
geom_line(aes(x=log(h),y=log(err_relativo_gf2_res)))+
ggtitle('Aproximación a la Hessiana por diferencias finitas')
```

```
print(c(h[which.min(err_relativo_gf2_res)],h[which.min(err_absoluto_gf2_res)]))
```

```
[1] 1e-06 1e-06
```

Ejercicio: aproximar $\nabla f(x)$, $\nabla^2 f(x)$ con diferencias hacia delante y $h \in \{10^{-16}, 10^{-14}, \dots, 10^{-1}\}$ para $f : \mathbb{R}^3 \rightarrow \mathbb{R}$, dada por $f(x) = x_1 x_2 \exp(x_1^2 + x_3^2 - 5)$ en el punto $x_0 = (1, 3, -2)^T$. Realizar una gráfica de $\log(\text{Err_rel})$ vs $\log(h)$.

Ejercicios

1. Resuelve los ejercicios y preguntas de la nota.

Referencias

1. R. L. Burden, J. D. Faires, Numerical Analysis, Brooks/Cole Cengage Learning, 2005.
2. M. T. Heath, Scientific Computing. An Introductory Survey, McGraw-Hill, 2002.
3. S. P. Boyd, L. Vandenberghe, Convex Optimization. Cambridge University Press, 2004.
4. Nota [Sistema de punto flotante](#).
5. Nota [Definición de función, derivada de una función](#).
6. Nota [Condición de un problema y estabilidad de un algoritmo](#).

💡 Notas para contenedor de docker:

Comando de docker para ejecución de la nota de forma local:

nota: cambiar <ruta a mi directorio> por la ruta de directorio que se desea mapear a /datos dentro del contenedor de docker.

```
docker run --rm -v <ruta a mi directorio>:/datos --name jupyterlab_optimizacion -p  
8888:8888 -d palmoreck/jupyterlab_optimizacion:2.1.4
```

password para jupyterlab: `qwerty`

Detener el contenedor de docker:

```
docker stop jupyterlab_optimizacion
```

Documentación de la imagen de docker `palmoreck/jupyterlab_optimizacion:2.1.4` en [liga](#).

Nota generada a partir de la [liga1](#) y [liga2](#).

Los métodos revisados en esta nota de integración numérica serán utilizados más adelante para revisión de herramientas en Python para **perfilamiento de código: uso de cpu y memoria**. También serán referidos en el capítulo de **cómputo en paralelo**.

```
import math
import numpy as np

from scipy.integrate import quad
import matplotlib.pyplot as plt
```

1.7 Integración Numérica

En lo siguiente consideramos que las funciones del integrando están en C^2 en el conjunto de integración (ver [Definición de función, derivada de una función](#) para definición de C^2).

Las reglas o métodos por cuadratura nos ayudan a aproximar integrales con sumas de la forma:

$$\int_a^b f(x)dx \approx \sum_{i=0}^n w_i f(x_i)$$

donde: w_i es el peso para el nodo x_i , f se llama integrando y $[a, b]$ intervalo de integración. Los valores $f(x_i)$ se asumen conocidos.

Todas las reglas o métodos por cuadratura se obtienen con interpoladores polinomiales del integrando (por ejemplo usando la representación de Lagrange) o también con el teorema Taylor (ver nota [Polinomios de Taylor y diferenciación numérica](#) para éste teorema).

Se realizan aproximaciones numéricas por:

- Desconocimiento de la función en todo el intervalo $[a, b]$ y sólo se conoce en los nodos su valor.
- Inexistencia de antiderivada o primitiva del integrando. Por ejemplo:

$$\int_a^b e^{-\frac{x^2}{2}} dx$$

Dependiendo de la ubicación de los nodos y pesos es el método de cuadratura que resulta:

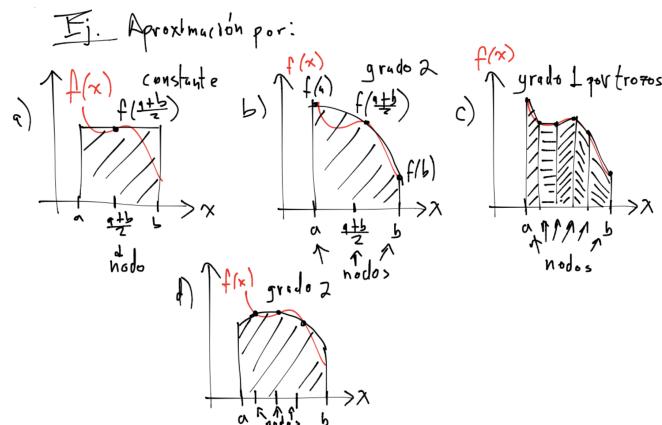
- Newton-Cotes si los nodos y pesos son equidistantes como la regla del rectángulo, trapezo y Simpson (con el teorema de Taylor o interpolación es posible obtener tales fórmulas).
- Cuadratura Gaussiana si se desea obtener reglas o fórmulas que tengan la mayor exactitud posible. Ejemplos de este tipo de cuadratura se tiene la regla por cuadratura Gauss-Legendre en $[-1, 1]$ o Gauss-Hermite para el caso de integrales en $[-\infty, \infty]$ con integrando $e^{-x^2} f(x)$.

Comentario:

Los métodos de integración numérica por Newton-Cotes o cuadratura Gaussiana pueden extenderse a más dimensiones, sin embargo incurren en lo que se conoce como la **maldición de la dimensionalidad** que para el caso de integración numérica consiste en la gran cantidad de evaluaciones que deben realizarse de la función del integrando para tener una exactitud pequeña, por ejemplo con un número de nodos igual a 10^4 , una distancia entre ellos de .1 y una integral en 4 dimensiones para la regla por Newton Cotes del rectángulo, se obtiene una exactitud de 2 dígitos. Como alternativa a los métodos por cuadratura anteriores para las integrales de más dimensiones se tienen los **métodos de integración por el método Monte Carlo** que generan aproximaciones con una exactitud moderada (del orden de $\mathcal{O}(n^{-1/2})$ con n número de nodos) para un número de puntos moderado independiente de la dimensión.

Newton-Cotes

Si el conjunto de nodos $x_i, i = 0, 1, \dots, n$ cumple $x_{i+1} - x_i = h, \forall i = 0, 1, \dots, n - 1$ con h (espaciado) constante y se approxima la función del integrando f con un polinomio en $(x_i, f(x_i)) \forall i = 0, 1, \dots, n$ entonces se tiene un método de integración numérica por Newton-Cotes (o reglas o fórmulas por Newton-Cotes).



En el dibujo: a),b) y c) se integra numéricamente por Newton-Cotes. d) es por cuadratura Gaussiana.

Comentario: si la fórmula por Newton-Cotes involucra el valor de la función en los extremos se llama cerrada, si no los involucra se les llama abiertas. En el dibujo d) es abierta.

El dibujo anterior muestra que puede subdividir el intervalo de integración en una mayor cantidad de subintervalos, lo cual para la función f mostrada es benéfico pues se tiene mejor aproximación (en la práctica esto será bueno? recuérdese los errores de redondeo de la nota [Sistema de punto flotante](#)). Los métodos que utilizan la idea anterior de dividir en subintervalos se les conoce como **métodos de integración numérica compuestos** en contraste con los simples.

Comentario: Para las reglas compuestas se divide el intervalo $[a, b]$ en n subintervalos $[x_{i-1}, x_i], i = 1, \dots, n$ con $x_0 = a < x_1 < \dots < x_{n-1} < x_n = b$ y se considera una partición regular, esto es: $x_i - x_{i-1} = \hat{h}$ con $\hat{h} = \frac{h}{n}$ y $h = b - a$. En este contexto se realiza la aproximación:

$$\int_a^b f(x)dx = \sum_{i=1}^n \int_{x_{i-1}}^{x_i} f(x)dx.$$

Para las siguientes reglas se considerará la función $f(x) = e^{-x^2}$ la cual tiene una forma:

```
f=lambda x: np.exp(-x**2)
```

```
x=np.arange(-1, 1, .01)
plt.plot(x,f(x))
plt.title('f(x)=exp(-x^2)')
plt.show()
```

El valor de la integral $\int_0^1 e^{-x^2} dx$ es:

```
obj, err = quad(f, 0, 1)
```

```
(obj,err)
```

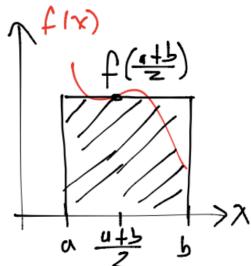
Obs: el segundo valor regresado `err`, es una cota superior del error.

Regla simple del rectángulo

Denotaremos a esta regla como Rf . En este caso se approxima el integrando f por un polinomio de grado **cero** con nodo en $x_1 = \frac{a+b}{2}$. Entonces:

$$\begin{aligned} \int_a^b f(x)dx &\approx \int_a^b f(x_1)dx = (b-a)f(x_1) \\ &= (b-a)f\left(\frac{a+b}{2}\right) = hf(x_1) \end{aligned}$$

con $h = b - a, x_1 = \frac{a+b}{2}$.



Ejemplo de implementación de regla simple de rectángulo: usando math

Utilizar la regla simple del rectángulo para aproximar la integral $\int_0^1 e^{-x^2} dx$.

```
f=lambda x: math.exp(-x**2) #using math library
```

```
def Rf(f,a,b):
    nodo=a+(b-a)/2.0 #mid point formula to minimize rounding errors
    return f(nodo) #polinomio de grado cero
```

```
Rf(f,0,1)
```

Regla compuesta del rectángulo

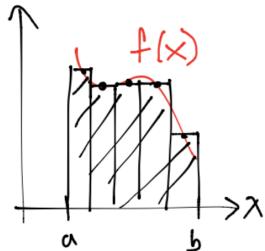
En cada subintervalo construido como $[a_{i-1}, a_i]$ con $i = 1, \dots, n$ se aplica la regla simple Rf , esto es:

$$\int_{a_{i-1}}^{a_i} f(x)dx \approx R_i(f) \forall i = 1, \dots, n.$$

De forma sencilla se puede ver que la regla compuesta del rectángulo $R_c(f)$ se escribe:

$$R_c(f) = \sum_{i=1}^n (a_i - a_{i-1}) f\left(\frac{a_i + a_{i-1}}{2}\right) = \frac{h}{n} \sum_{i=1}^n f\left(\frac{a_i + a_{i-1}}{2}\right)$$

con $h = b - a$ y n número de subintervalos.



Nota: Los nodos para el caso del rectángulo se obtienen con la fórmula:

$$x_i = a + (i + \frac{1}{2})\hat{h}, \forall i = 0, \dots, n-1, \hat{h} = \frac{h}{n}$$

Ejemplo de implementación de regla compuesta de rectángulo: usando math

Utilizar la regla compuesta del rectángulo para aproximar la integral $\int_0^1 e^{-x^2} dx$.

```
f=lambda x: math.exp(-x**2) #using math library
```

```
def Rcf(f,a,b,n): #Rcf: rectángulo compuesto para f
    """
    Compute numerical approximation using rectangle or mid-point method in
    an interval.
    Nodes are generated via formula: x_i = a+(i+1/2)h_hat for i=0,1,...,n-1 and h_hat=
    (b-a)/n
    Args:
        f (lambda expression): lambda expression of integrand
        a (int): left point of interval
        b (int): right point of interval
        n (int): number of subintervals
    Returns:
        Rcf (float)
    """
    h_hat=(b-a)/n
    nodes=[a+(i+1/2)*h_hat for i in range(0,n)]
    sum_res=0
    for node in nodes:
        sum_res=sum_res+f(node)
    return h_hat*sum_res
```

```
aprox_1=Rcf(f,0,1,1)
aprox_1
```

```
aprox_2=Rcf(f,0,1,2)
aprox_2
```

```
aprox_2=Rcf(f,0,1,2)
aprox_2
```

```
aprox_3=Rcf(f,0,1,10***3)
aprox_3
```

Y se puede evaluar el error de aproximación con el error relativo:

```
def err_relativo(aprox, obj):
    return math.fabs(aprox-obj)/math.fabs(obj) #obsérvese el uso de la librería math
```

```
obj, err = quad(f, 0, 1)
(err_relativo(aprox_1,obj), err_relativo(aprox_2,obj), err_relativo(aprox_3,obj))
```

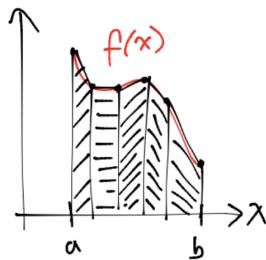
Pregunta: será el método del rectángulo un método estable numéricamente bajo el redondeo? Ver nota [Condición de un problema y estabilidad de un algoritmo](#) para definición de estabilidad numérica de un algoritmo.

```
aprox_4=Rcf(f, 0, 1, 10**5)
aprox_4
```

```
err_relativo(aprox_4,obj)
```

Al menos para este ejemplo con 10^5 nodos parece ser numéricamente estable...

Regla compuesta del trapecio



En cada subintervalo se aplica la regla simple T_f , esto es:

$$\int_{x_{i-1}}^{x_i} f(x)dx \approx T_i(f) \quad \forall i = 1, \dots, n.$$

Con $T_i(f) = \frac{(x_i - x_{i-1})}{2}(f(x_i) + f(x_{i-1}))$ para $i = 1, \dots, n$.

De forma sencilla se puede ver que la regla compuesta del trapecio $T_c(f)$ se escribe como:

$$T_c(f) = \frac{h}{2n} \left[f(x_0) + f(x_n) + 2 \sum_{i=1}^{n-1} f(x_i) \right]$$

con $h = b - a$ y n número de subintervalos.

Nota: Los nodos para el caso del trapecio se obtienen con la fórmula: $x_i = a + i\hat{h}$, $\forall i = 0, \dots, n$, $\hat{h} = \frac{h}{n}$.

Ejemplo de implementación de regla compuesta del trapecio: usando numpy

Con la regla compuesta del trapecio se aproximarán la integral $\int_0^1 e^{-x^2} dx$. Se calculará el error relativo y graficará n vs Error relativo para $n = 1, 10, 100, 1000, 10000$.

```
f=lambda x: np.exp(-x**2) #using numpy library
```

```
def Tcf(n,f,a,b): #Tcf: trapecio compuesto para f
    """
    Compute numerical approximation using trapezoidal rule in
    an interval.
    Nodes are generated via numpy
    Args:
        f (lambda expression): lambda expression of integrand
        a (int): left point of interval
        b (int): right point of interval
        n (int): number of subintervals
    Returns:
        Tcf (float)
    """
    h=b-a
    nodes=np.linspace(a,b,n+1)
    sum_res=sum(f(nodes[1:-1]))
    return h/(2**n)*(f(nodes[0])+f(nodes[-1])+2*sum_res)
```

Graficamos:

```
numb_of_subintervals=(1,10,100,1000,10000)
```

```
approx = np.array([Tcf(n,f,0,1) for n in numb_of_subintervals])
```

```
def err_relativo(aprox, obj):
    return np.abs(aprox-obj)/np.abs(obj) #obsérvese el uso de la librería numpy
```

```
err_relativo_res = err_relativo(approx,obj)
```

```
err_relativo_res
```

```
plt.plot(numb_of_subintervals,err_relativo_res, '.')
plt.xlabel('number of subintervals')
plt.ylabel('Relative error')
plt.title('Error relativo en la regla del Trapecio')
plt.show()
```

Si no nos interesa el valor de los errores relativos y sólo la gráfica podemos utilizar la siguiente opción:

```
from functools import partial
```

Ver [functools.partial](#) para documentación, [liga](#) para una explicación de `partial` y [liga2](#), [liga3](#) para ejemplos de uso.

```
approx=map(partial(Tcf,f=f,a=0,b=1),numb_of_subintervals) #map regresa un iterator
```

```
def err_relativo(aprox_map, obj):
    for ap in aprox_map:
        yield math.fabs(ap-obj)/math.fabs(obj) #obsérvese el uso de la librería math
```

Obs: la función `err_relativo` anterior es un [generator](#), ver [liga](#) para conocer el uso de `yield`.

```
err_relativo_res = err_relativo(approx,obj)
```

```
plt.plot(numb_of_subintervals,list(err_relativo_res), '*')
plt.xlabel('number of subintervals')
plt.ylabel('Relative error')
plt.title('Error relativo en la regla del Trapecio')
plt.show()
```

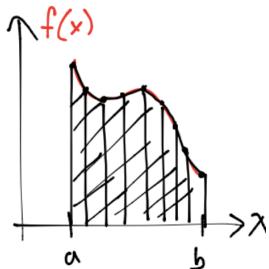
usando scatter:

```
approx=map(partial(Tcf,f=f,a=0,b=1),numb_of_subintervals) #map regresa un iterator
```

```
err_relativo_res = err_relativo(approx,obj)
```

```
[plt.scatter(n,err_rel) for n,err_rel in zip(numb_of_subintervals,err_relativo_res)]
plt.xlabel('number of subintervals')
plt.ylabel('Relative error')
plt.title('Error relativo en la regla del Trapecio')
plt.show()
```

Regla compuesta de Simpson



En cada subintervalo se aplica la regla simple Sf , esto es:

$$\int_{a_{i-1}}^{a_i} f(x)dx \approx S_i(f) \quad \forall i = 1, \dots, n$$

con $S_i(f) = \frac{h}{6}[f(x_{2i}) + f(x_{2i-2}) + 4f(x_{2i-1})]$ para el subintervalo $[a_{i-1}, a_i]$ con $i = 1, \dots, n$.

De forma sencilla se puede ver que la regla compuesta de Simpson compuesta $S_c(f)$ se escribe como:

$$S_c(f) = \frac{h}{3(2n)} \left[f(x_0) + f(x_{2n}) + 2 \sum_{i=1}^{n-1} f(x_{2i}) + 4 \sum_{i=1}^n f(x_{2i-1}) \right]$$

con $h = b - a$ y n número de subintervalos.

Nota: Los nodos para el caso de Simpson se obtienen con la fórmula: $x_i = a + \frac{i}{2}\hat{h}$, $\forall i = 0, \dots, 2n$, $\hat{h} = \frac{h}{n}$.

En esta [liga](#) está un apoyo visual para la regla Scf.

Ejercicio: implementar la regla compuesta de Simpson para aproximar la integral $\int_0^1 e^{-x^2} dx$. Calcular error relativo y realizar una gráfica de n vs Error relativo para $n = 1, 10, 100, 1000, 10000$ utilizando numpy e iterators.

La forma de los errores de las reglas del rectángulo, trapezio y Simpson se pueden obtener con interpolación o con el teorema de Taylor. Ver [Diferenciación e Integración](#) para detalles y [Polinomios de Taylor y diferenciación numérica](#) para el teorema. Suponiendo que f cumple con condiciones sobre sus derivadas, tales errores son:

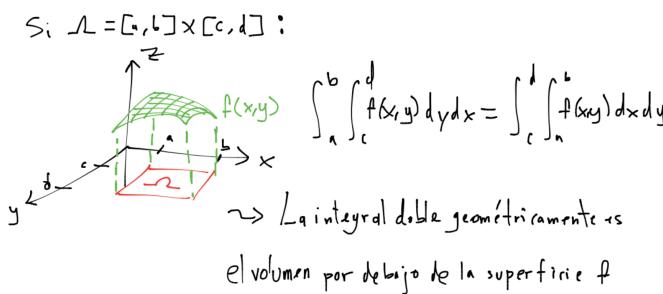
$$\begin{aligned} \text{Err}Rc(f) &= \frac{b-a}{6} f^{(2)}(\xi_r)\hat{h}^2, \xi_r \in [a, b] \\ \text{Err}Tc(f) &= -\frac{b-a}{12} f^{(2)}(\xi_t)\hat{h}^2, \xi_t \in [a, b] \\ \text{Err}Sc(f) &= -\frac{b-a}{180} f^{(4)}(\xi_S)\hat{h}^4, \xi_S \in [a, b]. \end{aligned}$$

Integración por el método de Monte Carlo

Los métodos de integración numérica por Monte Carlo son similares a los métodos por cuadratura en el sentido que se eligen puntos en los que se evaluará el integrando para sumar sus valores. La diferencia esencial con los métodos por cuadratura es que en el método de integración por Monte Carlo los puntos son seleccionados de una forma *aleatoria* (de hecho es pseudo-aleatoria pues se generan con un programa de computadora) en lugar de generarse con una fórmula.

Problema: Aproximar numéricamente la integral $\int_{\Omega} f(x)dx$ para $x \in \mathbb{R}^N$, $\Omega \subseteq \mathbb{R}^N$, $f : \mathbb{R}^N \rightarrow \mathbb{R}$ función tal que la integral esté bien definida en Ω .

Por ejemplo para $N = 2$:



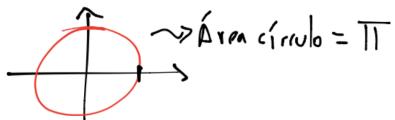
Para resolver el problema anterior con Ω un rectángulo, podemos utilizar las reglas por cuadratura por Newton-Cotes o cuadratura Gaussiana en una dimensión manteniendo fija la otra dimensión. Sin embargo considérese la siguiente situación:

La regla del rectángulo (o del punto medio) y del trapecio tienen un error de orden $\mathcal{O}(h^2)$ independientemente de si se está aproximando integrales de una o más dimensiones. Supóngase que se utilizan n nodos para tener un valor de espacio igual a h , entonces para N dimensiones se requerirían $P = n^N$ evaluaciones del integrando, o bien, si se tiene un valor de P igual a 10,000 y $N = 4$ dimensiones el error sería del orden $\mathcal{O}(P^{-2/N})$ lo que implicaría un valor de $h = .1$ para aproximadamente sólo **dos dígitos** correctos en la aproximación (para el enunciado anterior recuérdese que h es proporcional a n^{-1} y $n = P^{1/N}$). Este esfuerzo enorme de evaluar P veces el integrando para una exactitud pequeña se debe al problema de generar puntos para llenar un espacio N -dimensional y se conoce con el nombre de la maldición de la dimensionalidad, **the curse of dimensionality**.

Una opción para resolver la situación anterior si no se desea una exactitud grande (por ejemplo con una precisión de 10^{-4} o 4 dígitos es suficiente) es con el método de integración por Monte Carlo (tal nombre por el uso de números aleatorios). La integración por el método de Monte Carlo está basada en la interpretación geométrica de las integrales: calcular la integral del problema inicial implica calcular el **hipervolumen** de Ω .

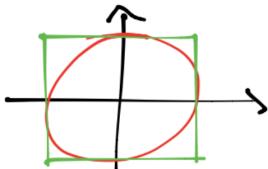
Ejemplo:

Supóngase que se desea aproximar el área de un círculo centrado en el origen de radio igual a 1:

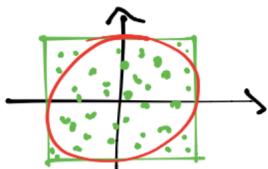


entonces el área de este círculo es $\pi r^2 = \pi$.

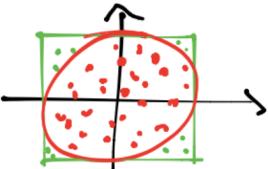
Para lo anterior **encerramos** al círculo con un cuadrado de lado 2:



Si tenemos n puntos en el cuadrado:



y consideraremos los m puntos que están dentro del círculo:



Entonces: $\frac{\text{Área del círculo}}{\text{Área del cuadrado}} \approx \frac{m}{n}$ y se tiene: $\text{Área del círculo} \approx \text{Área del cuadrado} \cdot \frac{m}{n}$ y si n crece entonces la aproximación es mejor.

prueba numérica:

```
density_p=int(2.5*10**3)
x_p=np.random.uniform(-1,1,(density_p,2))
plt.scatter(x_p[:,0],x_p[:,1],marker='.',color='g')
density=1e-5
x=np.arange(-1,1,density)
y1=np.sqrt(1-x**2)
y2=-np.sqrt(1-x**2)
plt.plot(x,y1,'r',x,y2,'r')
plt.title('Integración por Monte Carlo')
plt.grid()
plt.show()
```

```
f=lambda x: np.sqrt(x[:,0]**2 + x[:,1]**2) #definición de norma2
ind=f(x_p)<=1
x_p_subset=x_p[ind]
plt.scatter(x_p_subset[:,0],x_p_subset[:,1],marker='.',color='r')
plt.title('Integración por Monte Carlo')
plt.grid()
plt.show()
```

Área del círculo es aproximadamente:

```
Area_cuadrado=4
Area_cuadrado*len(x_p_subset)/len(x_p)
```

Si aumentamos el número de puntos...

```
density_p=int(10**4)
x_p=np.random.uniform(-1,1,(density_p,2))
ind=f(x_p)<=1
x_p_subset=x_p[ind]
Area_cuadrado*len(x_p_subset)/len(x_p)
```

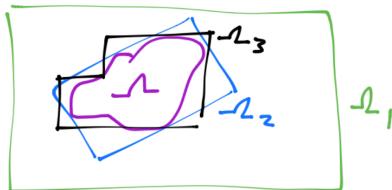
```

density_p=int(10**5)
x_p=np.random.uniform(-1,1,(density_p,2))
ind=f(x_p)<=1
x_p_subset=x_p[ind]
Area_cuadrado*len(x_p_subset)/len(x_p)

```

Comentarios:

- El método de Monte Carlo revisado en el ejemplo anterior nos indica que debemos encerrar a la región de integración Ω . Por ejemplo para una región Ω más general:



entonces la integración por el método de Monte Carlo será:

$$\int_{\Omega} f d\Omega \approx V \bar{f}$$

donde: V es el hipervolumen de Ω_E que encierra a Ω , esto es $\Omega \subseteq \Omega_E$, $\{x_1, \dots, x_n\}$ es un conjunto de puntos

distribuidos uniformemente en Ω_E y $\bar{f} = \frac{1}{n} \sum_{i=1}^n f(x_i)$

- Consideraremos \bar{f} pues $\sum_{i=1}^n f(x_i)$ representa el valor de m si pensamos a f como una restricción que deben cumplir los n puntos en el ejemplo de aproximación al área del círculo: Área del círculo \approx Área del cuadrado $\frac{m}{n}$ (en este caso Área del cuadrado es el hipervolumen V).
- Algunas características para regiones que encierran a Ω es que:
 - Sea sencillo generar números aleatorios uniformes.
 - Sea sencillo obtener su hipervolumen.

Ejemplos: aproximar las siguientes integrales:

```

density_p=int(10**4)

```

$$\bullet \int_0^1 \frac{4}{1+x^2} dx = \pi$$

```

f=lambda x: 4/(1+x**2)
x_p=np.random.uniform(0,1,density_p)
obj=math.pi
a=0
b=1
vol=b-a
approx=vol*np.mean(f(x_p))
err_rel=lambda ap,obj: math.fabs(ap-obj)/math.fabs(obj)
"error relativo: {:.0e}".format(err_rel(approx,obj))

```

$$\bullet \int_1^2 \frac{1}{x} dx = \log 2.$$

```

f=lambda x: 1/x
x_p=np.random.uniform(1,2,density_p)
obj=math.log(2)
a=1
b=2
vol=b-a
approx=vol*np.mean(f(x_p))
"error relativo: {:.0e}".format(err_rel(approx,obj))

```

$$\bullet \int_{-1}^1 \int_0^1 x^2 + y^2 dx dy = \frac{4}{3}.$$

```

f=lambda x,y:x**2+y**2
a1=-1
b1=1
a2=0
b2=1
x_p=np.random.uniform(a1,b1,density_p)
y_p=np.random.uniform(a2,b2,density_p)
obj=4/3
vol=(b1-a1)*(b2-a2)
approx=vol*np.mean(f(x_p,y_p))
"error relativo: {:.0e}".format(err_rel(approx,obj))

```

$$\bullet \int_0^{\frac{\pi}{2}} \int_0^{\frac{\pi}{2}} \cos(x) \sin(y) dx dy = 1.$$

```

f=lambda x,y:np.cos(x)*np.sin(y)
a1=0
b1=math.pi/2
a2=0
b2=math.pi/2
x_p=np.random.uniform(a1,b1,density_p)
y_p=np.random.uniform(a2,b2,density_p)
obj=1
vol=(b1-a1)*(b2-a2)
approx=vol*np.mean(f(x_p,y_p))
"error relativo: {:.0e}".format(err_rel(approx,obj))

```

$$\bullet \int_0^1 \int_{-\frac{1}{2}}^0 \int_0^{\frac{1}{3}} (x + 2y + 3z)^2 dx dy dz = \frac{1}{12}.$$

```

f=lambda x,y,z:(x+2*y+3*z)**2
a1=0
b1=1
a2=-1/2
b2=0
a3=0
b3=1/3
x_p=np.random.uniform(a1,b1,density_p)
y_p=np.random.uniform(a2,b2,density_p)
z_p=np.random.uniform(a3,b3,density_p)
obj=1/12
vol=(b1-a1)*(b2-a2)*(b3-a3)
approx=vol*np.mean(f(x_p,y_p,z_p))
"error relativo: {:.0e}".format(err_rel(approx,obj))

```

Cuál es el error en la aproximación por el método de integración por Monte Carlo?

Para obtener la expresión del error en esta aproximación supóngase que x_1, x_2, \dots, x_n son variables aleatorias independientes uniformemente distribuidas. Entonces:

$$\begin{aligned} \text{Err}(\bar{f}) &= \sqrt{\text{Var}(\bar{f})} = \sqrt{\text{Var}\left(\frac{1}{n} \sum_{i=1}^n f(x_i)\right)} = \dots \\ &= \sqrt{\frac{\text{Var}(f(x))}{n}} \end{aligned}$$

con x variable aleatoria uniformemente distribuida.

Un estimador de $\text{Var}(f(x))$ es: $\frac{1}{n} \sum_{i=1}^n (f(x_i) - \bar{f})^2 = \bar{f}^2 - \bar{f}^2$ por lo que $\hat{\text{Err}}(\bar{f}) = \sqrt{\frac{\bar{f}^2 - \bar{f}^2}{n}}$.

Se tiene entonces:

$$\begin{aligned} \int_{\Omega} f d\Omega &\approx V(\bar{f} \pm \text{Err}(\bar{f})) \approx V(\bar{f} \pm \hat{\text{Err}}(\bar{f})) = V\bar{f} \\ &\quad \pm V\sqrt{\frac{\bar{f}^2 - \bar{f}^2}{n}} \end{aligned}$$

Ejemplo:

Para el ejemplo anterior $\int_0^1 \frac{4}{1+x^2} dx = \pi$ se tiene:

```

f=lambda x: 4/(1+x**2)
x_p=np.random.uniform(0,1,density_p)
obj=math.pi
a=0
b=1
vol=b-a
f_barra=np.mean(f(x_p))
approx=vol*f_barra
err_rel=lambda ap,obj: math.fabs(ap-ob)/math.fabs(ob)
"error relativo: {:.0e}".format(err_rel(approx,obj))

```

```
error_est = math.sqrt(sum((f(x_p)-f_barra)**2)/density_p**2)
```

```
error_est
```

intervalo:

```
(approx-vol*error_est, approx+vol*error_est)
```

Ejercicios: Aproximar, reportar errores relativos e intervalo de estimación en una tabla:

- $\int_0^1 \int_0^1 \sqrt{x+y} dy dx = \frac{2}{3} \left(\frac{2}{5} 2^{5/2} - \frac{4}{5} \right).$
- $\int_D \int \sqrt{x+y} dy dx = 8 \frac{\sqrt{2}}{15}$ donde: $D = \{(x, y) \in \mathbb{R}^2 | 0 \leq x \leq 1, -x \leq y \leq x\}$.
- $\int_D \int \exp(x^2 + y^2) dy dx = \pi(e^9 - 1)$ donde $D = \{(x, y) \in \mathbb{R}^2 | x^2 + y^2 \leq 9\}$.
- $\int_0^2 \int_{-1}^1 \int_0^1 (2x + 3y + z) dz dy dx = 10.$

Comentarios:

- Los signos \pm en el error de aproximación **no** representan una cota rigurosa, es una desviación estándar.
- A diferencia de la aproximación por las reglas por cuadratura tenemos una precisión con n puntos independientemente de la dimensión N .
- Si $N \rightarrow \infty$ entonces $\hat{\text{Err}}(\bar{f}) = \mathcal{O}\left(\frac{1}{\sqrt{n}}\right)$ por lo que para ganar un decimal extra de precisión en la integración por el método de Monte Carlo se requiere incrementar el número de puntos por un factor de 10^2 .

Obs: obsérvese que si f es constante entonces $\hat{\text{Err}}(\bar{f}) = 0$. Esto implica que si f es casi constante y Ω_E encierra muy bien a Ω entonces se tendrá una estimación muy precisa de $\int_{\Omega} f d\Omega$, por esto en la integración por el método de Monte Carlo se realizan cambios de variable de modo que transformen a f en aproximadamente constante y que esto resulte además en regiones Ω_E que encierran a Ω casi de manera exacta (y que además sea sencillo generar números pseudo aleatorios en ellas!).

Comentario:

La integración por el método de Monte Carlo se utiliza para aproximar características de variables aleatorias continuas. Por ejemplo, si x es variable aleatoria continua, entonces su media está dada por:

$$E_f[h(X)] = \int_{S_X} h(x) f(x) dx$$

donde f es función de densidad de X , S_X es el soporte de X y h es una transformación. Entonces:

$$E_f[h(X)] \approx \frac{1}{n} \sum_{i=1}^n h(x_i) = \bar{h}_n$$

con $\{x_1, x_2, \dots, x_n\}$ muestra de f . Y por la ley de los grandes números se tiene:

$$\frac{\bar{h}_n - E_f[h(X)]}{\hat{\text{Err}}(\bar{h})} \xrightarrow{n \rightarrow \infty} N(0, 1)$$

con $N(0, 1)$ una distribución Normal con $\mu = 0, \sigma = 1$ ∴ si $n \rightarrow \infty$ un intervalo de confianza al 95% para $E_f[h(X)]$ es: $(\bar{h}_n \pm z_{.975} \hat{\text{Err}}(\bar{h}))$.

Obs: uno de los pasos complicados en el desarrollo anterior es obtener una muestra de f que para el caso de variables continuas se puede utilizar el teorema de transformación inversa o integral de probabilidad.

Ejercicios

1. Resuelve los ejercicios y preguntas de la nota.

Referencias

1. R. L. Burden, J. D. Faires, Numerical Analysis, Brooks/Cole Cengage Learning, 2005.
2. M. T. Heath, Scientific Computing. An Introductory Survey, McGraw-Hill, 2002.
3. Nota [Sistema de punto flotante](#).
4. Nota [Definición de función, derivada de una función](#).
5. Nota [Polinomios de Taylor y diferenciación numérica](#).
6. Nota [Condición de un problema y estabilidad de un algoritmo](#).

💡 Notas para contenedor de docker:

Comando de docker para ejecución de la nota de forma local:

nota: cambiar `<ruta a mi directorio>:/datos` por la ruta de directorio que se desea mapear a `/datos` dentro del contenedor de docker.

```
docker run --rm -v <ruta a mi directorio>:/datos --name jupyterlab_optimizacion -p  
8888:8888 -d palmoreck/jupyterlab_optimizacion:2.1.4
```

password para jupyterlab: `qwerty`

Detener el contenedor de docker:

```
docker stop jupyterlab_optimizacion
```

Documentación de la imagen de docker `palmoreck/jupyterlab_optimizacion:2.1.4` en [liga](#).

Nota generada a partir de [liga1](#), [liga2](#) y [liga3](#).

2.1 Operaciones y transformaciones básicas del Álgebra Lineal Numérica

Las operaciones básicas del Álgebra Lineal Numérica podemos dividirlas en vectoriales y matriciales.

Vectoriales

- **Transponer:** $\mathbb{R}^{n \times 1} \rightarrow \mathbb{R}^{1 \times n}$: $y = x^T$ entonces

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

y se tiene:

$$y = x^T = [x_1, x_2, \dots, x_n].$$

- **Suma:** $\mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^n$: $z = x + y$ entonces $z_i = x_i + y_i$
- **Multiplicación por un escalar:** $\mathbb{R} \times \mathbb{R}^n \rightarrow \mathbb{R}^n$: $y = \alpha x$ entonces $y_i = \alpha x_i$.
- **Producto interno estándar o producto punto:** $\mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$: $c = x^T y$ entonces $c = \sum_{i=1}^n x_i y_i$.
- **Multiplicación point wise:** $\mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^n$: $z = x.*y$ entonces $z_i = x_i y_i$.
- **División point wise:** $\mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^n$: $z = x./y$ entonces $z_i = x_i/y_i$ con $y_i \neq 0$.
- **Producto exterior o outer product:** $\mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^{n \times n}$: $A = xy^T$ entonces $A[i, :] = x_i y^T$ con $A[i, :]$ es el i -ésimo renglón de A .

Matriciales

- **Transponer:** $\mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{n \times m}$: $C = A^T$ entonces $c_{ij} = a_{ji}$.

- **Sumar:** $\mathbb{R}^{m \times n} \times \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{m \times n}: C = A + B$ entonces $c_{ij} = a_{ij} + b_{ij}$.
- **Multiplicación por un escalar:** $\mathbb{R} \times \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{m \times n}: C = \alpha A$ entonces $c_{ij} = \alpha a_{ij}$
- **Multiplicación por un vector:** $\mathbb{R}^{m \times n} \times \mathbb{R}^n \rightarrow \mathbb{R}^m: y = Ax$ entonces $y_i = \sum_{j=1}^n a_{ij}x_j$.
- **Multiplicación entre matrices:** $\mathbb{R}^{m \times k} \times \mathbb{R}^{k \times n} \rightarrow \mathbb{R}^{m \times n}: C = AB$ entonces $c_{ij} = \sum_{r=1}^k a_{ir}b_{rj}$.
- **Multiplicación point wise:** $\mathbb{R}^{m \times n} \times \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{m \times n}: C = A.*B$ entonces $c_{ij} = a_{ij}b_{ij}$.
- **División point wise:** $\mathbb{R}^{m \times n} \times \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{m \times n}: C = A./B$ entonces $c_{ij} = a_{ij}/b_{ij}$ con $b_{ij} \neq 0$.

Como ejemplos de transformaciones básicas del Álgebra Lineal Numérica se encuentran:

```
import numpy as np
```

Transformaciones de Gauss

En esta sección suponemos que $A \in \mathbb{R}^{n \times n}$ y A es una matriz con entradas $a_{ij} \in \mathbb{R}^{n \times n} \forall i, j = 1, 2, \dots, n$.

Considérese al vector $a \in \mathbb{R}^n$ y $e_k \in \mathbb{R}^n$ el k -ésimo vector canónico: vector con un 1 en la posición k y ceros en las entradas restantes.

Una transformación de Gauss está definida de forma general como $L_k = I_n - \ell_k e_k^T$ con $\ell_k = (0, 0, \dots, \ell_{k+1,k}, \dots, \ell_{n,k})^T$ y $\ell_{i,k} = \frac{a_{ik}}{a_{kk}} \forall i = k+1, \dots, n$.

Las transformaciones de Gauss se utilizan para hacer ceros por debajo del **pivote**.

Ejemplo aplicando transformaciones de Gauss a un vector

Considérese al vector $a = (-2, 3, 4)^T$. Definir una transformación de Gauss para hacer ceros por debajo de a_1 y otra transformación de Gauss para hacer cero la entrada a_3

Solución:

a) Para hacer ceros por debajo del **pivote** $a_1 = -2$:

```
a = np.array([-2, 3, 4])
```

```
pivot = a[0]
```

```
l1 = np.array([0, a[1]/pivot, a[2]/pivot])
```

```
e1 = np.array([1, 0, 0])
```

```
L1_a = a-l1*(e1.dot(a))
```

```
L1_a
```

Recuerda

La definición de $l_1 = (0, \frac{a_2}{a_1}, \frac{a_3}{a_1})^T$

A continuación se muestra que el producto $L_1 a$ si se construye L_1 es equivalente a lo anterior:

```
L1 = np.eye(3) - np.outer(l1, e1)
```

```
L1
```

```
L1@a
```

$L_1 = I_3 - \ell_1 e_1^T$.

b) Para hacer ceros por debajo del **pivote** $a_2 = 3$:

```
a = np.array([-2, 3, 4])
```

```
pivot = a[1]
```

Observa que por la definición de la transformación de Gauss, no necesitamos construir a la matriz L_1 , directamente se tiene $L_1 a = a - \ell_1 e_1^T a$.

```
l2 = np.array([0, 0, a[2]/pivot])
```

```
e2 = np.array([0, 1, 0])
```

```
L2_a = a-l2*(e2.dot(a))
```

```
L2_a
```

A continuación se muestra que el producto $L_2 a$ si se construye L_2 es equivalente a lo anterior:

```
L2 = np.eye(3) - np.outer(l2, e2)
```

```
L2
```

```
L2@a
```

Recuerda

La definición de $l_2 = (0, 0, \frac{a_3}{a_2})^T$

Observa que por la definición de la transformación de Gauss, no necesitamos construir a la matriz L_2 , directamente se tiene $L_2 a = a - \ell_2 e_2^T a$.

$$L_2 = I_3 - \ell_2 e_2^T.$$

Ejemplo aplicando transformaciones de Gauss a una matriz

Si tenemos una matriz $A \in \mathbb{R}^{3 \times 3}$ y queremos hacer ceros por debajo de su diagonal, realizamos los productos matriciales:

$$L_2 L_1 A$$

donde: L_1, L_2 son transformaciones de Gauss.

Ejemplo:

a) Utilizando L_1

```
A = np.array([[[-1, 2, 5],  
              [4, 5, -7],  
              [3, 0, 8]])
```

```
A
```

Para hacer ceros por debajo del **pivote** $a_{11} = -1$:

```
pivote = A[0, 0]
```

```
l1 = np.array([0, A[1, 0]/pivote, A[2, 0]/pivote])
```

```
e1 = np.array([1, 0, 0])
```

```
L1_A_1 = A[:, 0]-l1*(e1.dot(A[:, 0]))
```

```
L1_A_1
```

Y se debe aplicar ℓ_1 a las columnas número 2 y 3 de A :

```
L1_A_2 = A[:, 1]-l1*(e1.dot(A[:, 1]))
```

```
L1_A_2
```

```
L1_A_3 = A[:, 2]-l1*(e1.dot(A[:, 2]))
```

```
L1_A_3
```

Recuerda

La definición de $l_1 = (0, \frac{a_{21}}{a_{11}}, \frac{a_{31}}{a_{11}})^T$

Observa que por la definición de la transformación de Gauss, no necesitamos construir a la matriz L_1 , directamente se tiene $L_1 A[:, 1] = A[:, 1]$.
 $-\ell_1 e_1^T A[:, 1]$

A continuación se muestra que el producto $L_1 A$ si se construye L_1 es equivalente a lo anterior:

```
L1 = np.eye(3) - np.outer(l1, e1)
```

$$L_1 = I_3 - \ell_1 e_1^T.$$

```
L1
```

```
L1 @ A
```

💡 Observación 1

Al aplicar ℓ_1 a la primer columna de A **siempre** obtenemos ceros por debajo del pivote que en este caso es a_{11} .

💡 Observación 2

Después de hacer la multiplicación $L_1 A$ en cualquiera de los dos casos (construyendo o no explícitamente L_1) no se modifica el primer renglón de A :

```
A
```

```
A[0, :]
```

```
(L1 @ A)[0, :]
```

💡 Continuación observación 2

por lo que la multiplicación $L_1 A$ entonces modifica del segundo renglón de A en adelante y de la segunda columna de A en adelante.

💡 Observación 3

Dada la forma de $L_1 = I_n - \ell_1 e_1^T$, al hacer la multiplicación por la segunda y tercera columna de A se tiene:

$$e_1^T A[:, 1] = A[0, 1]$$
$$e_1^T A[:, 2] = A[0, 2]$$

```
e1.dot(A[:, 1])
```

```
e1.dot(A[:, 2])
```

y puede escribirse de forma compacta:

$$e_1^T A[:, 1 : 2] = A[0, 1 : 2]$$

```
A[0, 1:3] #observe that we have to use 2+1=3 as the second number in 1:3
```

```
A[0, 1:] #also we could have use this statement
```

Entonces los productos $\ell_1 e_1^T A[:, 1]$ y $\ell_1 e_1^T A[:, 2]$ quedan respectivamente como:

$$\ell_1 A[0, 1]$$

```
11*A[0, 1]
```

$$\ell_1 A[0, 2]$$

```
11*A[0, 2]
```

ℹ️ Observación 4

En los dos cálculos anteriores, las primeras entradas son iguales a 0 por lo que es consistente con el hecho que únicamente se modifican dos entradas de la segunda y tercera columna de A .

De forma compacta se puede calcular lo anterior como:

El resultado de este producto es un escalar.

El resultado de este producto es un escalar.

```
np.outer(l1[1:3],A[0,1:3])
```

```
np.outer(l1[1:],A[0,:]) #also we could have use this statement
```

Y finalmente la aplicación de L_1 al segundo renglón y segunda columna en adelante de A queda:

```
A[1:3, 1:3] - np.outer(l1[1:3],A[0,1:3])
```

Practicando combinar columnas y renglones en *numpy* con [column_stack](#) y [row_stack](#):

```
A_aux = A[1:3, 1:3] - np.outer(l1[1:3],A[0,1:3])
```

```
m, n = A.shape  
number_of_zeros = m-1  
A_aux_2 = np.column_stack((np.zeros(number_of_zeros), A_aux)) # stack two zeros
```

```
A_aux_2
```

```
A_aux_3 = np.row_stack((A[0, :], A_aux_2))
```

```
A_aux_3
```

que es el resultado de:

```
L1 @ A
```

Ejercicio

Calcular el producto $L_2 L_1 A$ para la matriz anterior y para la matriz:

$$A = \begin{bmatrix} 1 & 4 & -2 \\ -3 & 9 & 8 \\ 5 & 1 & -6 \end{bmatrix}$$

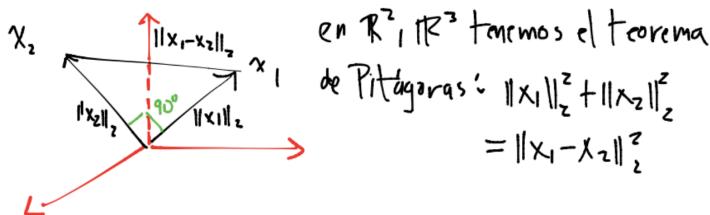
Comentario

Las transformaciones de Gauss se utilizan para la fase de eliminación del método de eliminación Gaussiana o también llamada factorización LU. Ver [Gaussian elimination](#).

Transformaciones de reflexión

Matriz ortogonal y matriz con columnas ortonormales

Un conjunto de vectores $\{x_1, \dots, x_p\}$ en \mathbb{R}^m ($x_i \in \mathbb{R}^m$) es ortogonal si $x_i^T x_j = 0 \forall i \neq j$. Por ejemplo, para un conjunto de 2 vectores x_1, x_2 en \mathbb{R}^3 esto se visualiza:



Observa que por la definición de la transformación de Gauss, no necesitamos construir a la matriz L_1 , directamente se tiene $L_1 A = A - \ell_1 e_1^T A$.

Comentarios

- Si el conjunto $\{x_1, \dots, x_p\}$ en \mathbb{R}^m satisface $x_i^T x_j = \delta_{ij} = \begin{cases} 1 & \text{si } i = j, \\ 0 & \text{si } i \neq j \end{cases}$ se le nombra conjunto **ortonormal**, esto es, constituye un conjunto ortogonal y cada elemento del conjunto tiene norma Euclídea igual a 1: $\|x_i\|_2 = 1, \forall i = 1, \dots, p$.
- Si definimos a la matriz X con columnas dadas por cada uno de los vectores del conjunto $\{x_1, \dots, x_p\}: X = (x_1, \dots, x_p) \in \mathbb{R}^{m \times p}$ entonces la propiedad de que cada par de columnas satisaga $x_i^T x_j = \delta_{ij}$ se puede escribir en notación matricial como $X^T X = I_p$ con I_p la matriz identidad de tamaño p o bien $XX^T = I_m$. A la matriz X se le nombra **matriz con columnas ortonormales**.
- Si cada x_i está en \mathbb{R}^p (en lugar de \mathbb{R}^m) entonces construimos a la matriz X como el punto anterior con la diferencia que $X \in \mathbb{R}^{p \times p}$. En este caso X se le nombra **matriz ortogonal**.

Reflectores de Householder

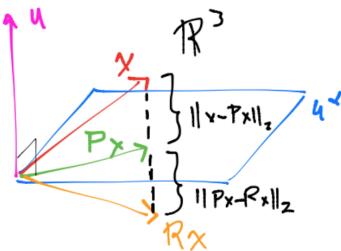
Las reflexiones de Householder son **matrices** simétricas, ortogonales y se construyen a partir de un vector $v \neq 0$ definiendo:

$$R = I_m - \beta v v^T$$

con $v \in \mathbb{R}^m - \{0\}$ y $\beta = \frac{2}{v^T v}$. El vector v se llama **vector de Householder**. La multiplicación Rx representa la reflexión del vector $x \in \mathbb{R}^m$ a través del hiperplano $\text{span}\{v\}^\perp$.

Algunas propiedades de las reflexiones de Householder son: $R^T R = R^2 = I_m$, $R^{-1} = R$, $\det(R) = -1$.

Un dibujo que representa lo anterior es el siguiente en el que se utiliza $u \in \mathbb{R}^m - \{0\}$, $\|u\|_2 = 1$ y $R = I_m - 2uu^T$, el reflector elemental alrededor de u^\perp :



Las reflexiones de Householder pueden utilizarse para hacer ceros por debajo de una entrada de un vector. Por ejemplo:

```
x = np.array([1, 2, 3])
```

```
x
```

Utilizamos la definición $v = x - \|x\|_2 e_1$ con $e_1 = (1, 0, 0)^T$ vector canónico para construir al vector de Householder:

```
v = x-np.linalg.norm(x)*np.array([1, 0, 0]) #uno en la primera entrada pues se desea
#hacer ceros en las entradas restantes
```

```
beta = 2/v.dot(v)
```

Hacemos ceros por debajo de la primera entrada de x haciendo la multiplicación matriz-vector Rx . Pero para aplicar Rx no construimos $R = I_3 - \beta v v^T$, en lugar de eso hacemos $Rx = x - \beta v v^T x$:

```
x-beta*v*(v.dot(x))
```

obsérvese que el resultado de Rx es $(\|x\|_2, 0, 0)^T$:

```
np.linalg.norm(x)
```

$\text{span}\{v\}$ es el conjunto generado por v . Se define como el conjunto de combinaciones lineales de v :
 $\text{span}\{v\} = \left\{ \sum_{i=1}^m k_i v_i | k_i \in \mathbb{R} \forall i = 1, \dots, m \right\}$

Proyector ortogonal elemental

En este dibujo se utiliza el **proyector ortogonal elemental** sobre el complemento ortogonal de u : $u^\perp = \{x \in \mathbb{R}^m | u^T x = 0\}$ (que es un subespacio de \mathbb{R}^m de dimensión $m - 1$) definido como:

$P = I_m - uu^T$ y Px es la proyección ortogonal de x sobre u^\perp . Los proyectores ortogonales elementales **no** son matrices ortogonales, son singulares, son simétricas y $P^2 = P$. El proyector ortogonal elemental de x sobre u^\perp tienen rank igual a $m - 1$ y el proyector ortogonal de x sobre $\text{span}\{u\}$ definido por $I_m - P = uu^T$ tienen rank igual a 1.

x se escribe como la suma de 2 componentes:
 $x = (I - P)x + Px$
Observa que $u^T P x = 0$ y
 $\|u^T x\| = \|u^T x\|$

Sólo para mostrar que Rx construyendo R es equivalente a lo anterior, se construye a continuación R :

```
R = np.eye(3)-beta*np.outer(v,np.transpose(v))
```

```
R
```

```
R@x
```

Comentario:

- Otra opción para definir al vector de Householder es $v = x + \|\mathbf{x}\|_2 e_1$ con $e_1 = (1, 0, 0)^T$:

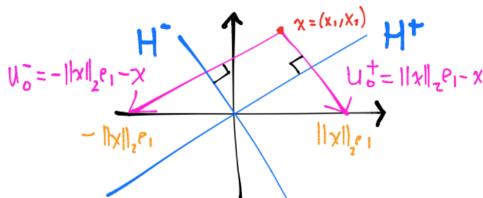
```
v = x+np.linalg.norm(x)*np.array([1,0,0]) #uno en la primer entrada pues se desea
#hacer ceros en las entradas restantes
```

```
beta = 2/v.dot(v)
```

```
x-beta*v*(v.dot(x))
```

¿Cuál definición del vector de Householder usar?

La respuesta a la pregunta tiene que ver con que en cualquiera de las dos definiciones del vector de Householder $v = x \pm \|\mathbf{x}\|_2 e_1$, la multiplicación Rx refleja x en el primer eje coordenado:



El vector $v^+ = -u_0^+ = x - \|\mathbf{x}\|_2 e_1$ refleja x respecto al subespacio H^+ . El vector $v^- = -u_0^- = x + \|\mathbf{x}\|_2 e_1$ refleja x respecto al subespacio H^- . Para disminuir los errores por redondeo y evitar el problema de cancelación en la aritmética de punto flotante (ver [Sistema de punto flotante](#)) se utiliza $v = x + signo(x_1)\|\mathbf{x}\|_2 e_1$ donde $signo(x_1) = \begin{cases} 1 & \text{si } x_1 \geq 0, \\ -1 & \text{si } x_1 < 0 \end{cases}$. La idea de la definición anterior con la función $signo(\cdot)$ es que la reflexión (en el dibujo anterior $-\|\mathbf{x}\|_2 e_1$ o $\|\mathbf{x}\|_2 e_1$) sea lo más alejada posible de x . En el dibujo anterior como $x_1, x_2 > 0$ entonces se refleja respecto al subespacio H^- quedando su reflexión igual a $-\|\mathbf{x}\|_2 e_1$.

Comentarios:

- Otra forma de lidiar con el problema de cancelación es definiendo a la primera componente del vector de Householder v_1 como $v_1 = x_1 - \|\mathbf{x}\|_2$ y haciendo una manipulación algebraica como sigue:

$$v_1 = x_1 - \|\mathbf{x}\|_2 = \frac{x_1^2 - \|\mathbf{x}\|_2^2}{x_1 + \|\mathbf{x}\|_2} = -\frac{x_2^2 + x_3^2 + \dots + x_m^2}{x_1 + \|\mathbf{x}\|_2}.$$

- En la implementación del cálculo del vector de Householder, es útil que $v_1 = 1$ y así únicamente se almacenará $v(2 : m)$. Al vector $v(2 : m)$ se le nombra **parte esencial del vector de Householder**.

Transformaciones de rotación

Referencias:

- G. H. Golub, C. F. Van Loan, Matrix Computations, John Hopkins University Press, 2013.

By Erick Palacios Moreno

© Copyright 2020.