Vectores y matrices

Temario

- Aproximación de funciones en un intervalo
- numpy: vectores. arrays. numpy (vs listas)
- Generaciones automáticas: linspace
- Multiplicaciones de arrays: matmul, dot,

Fecha Primer Parcial: 25 de Setiembre! (la semana que viene)

Series de Taylor

Supongamos el desarrollo alrededor de x_0 de la función $f(x)$

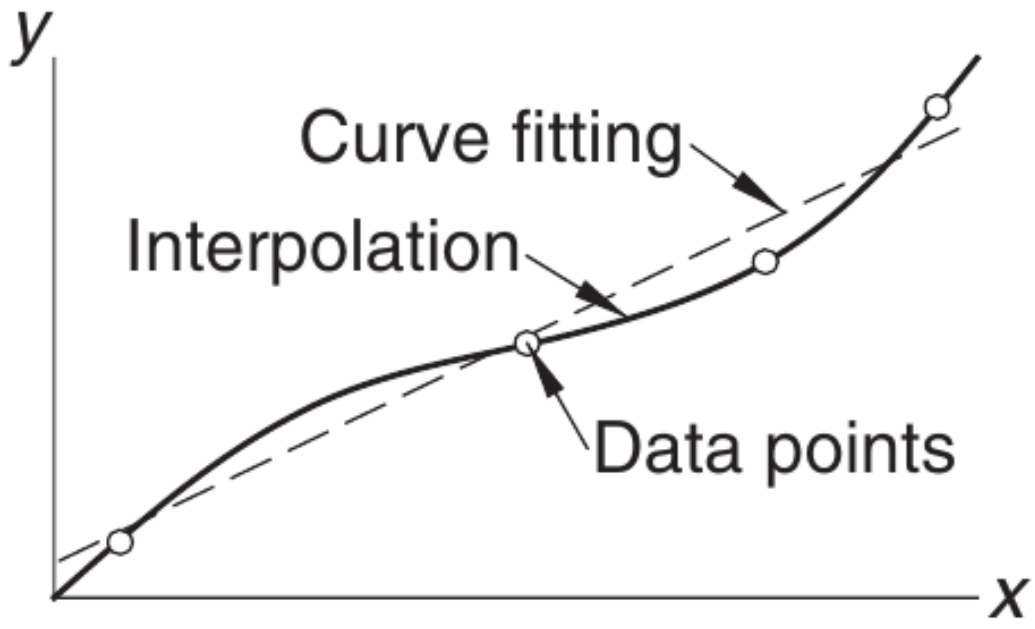
$$f(x) \approx f(x_0) + \frac{df}{dx}(x_0)(x - x_0) + \frac{1}{2}\frac{d^2f}{dx^2}(x_0)(x - x_0)^2 + \cdots + \frac{1}{n!}\frac{d^nf}{dx^n}(x_0)(x - x_0)^n$$

$$f(x) \approx \sum_{i=0}^{n} \frac{1}{i!}\frac{d^if}{dx^i}(x_0)(x - x_0)^i$$

- La idea es que si $x - x_0 \ll 1$, entonces los términos de la serie disminuyen con la potencia.
- Esto permite que el error de la aproximación este acotado (los primeros términos de la serie son los dominantes).
- El error de aproximacón disminuye a medida que aumentamos la cantidad de términos.

Aproximación de funciones en un intervalo

Dados un conjunto de n puntos x_i, y_i, cual es la función que mejor representa a estos puntos?



Interpolación polinómica de funciones

El polinomio que pasa por los puntos se puede obtener por el método de Lagrange:

$$P_n(x) = \sum_{i=0}^{n} y_i l_i(x)$$

donde las $l_i(x)$ son funciones cardinales que tienen la propiedad que son polinomios que se hacen 0 en todos los puntos excepto el que es de interés

$$l_i(x_j) = \delta_{ij}$$

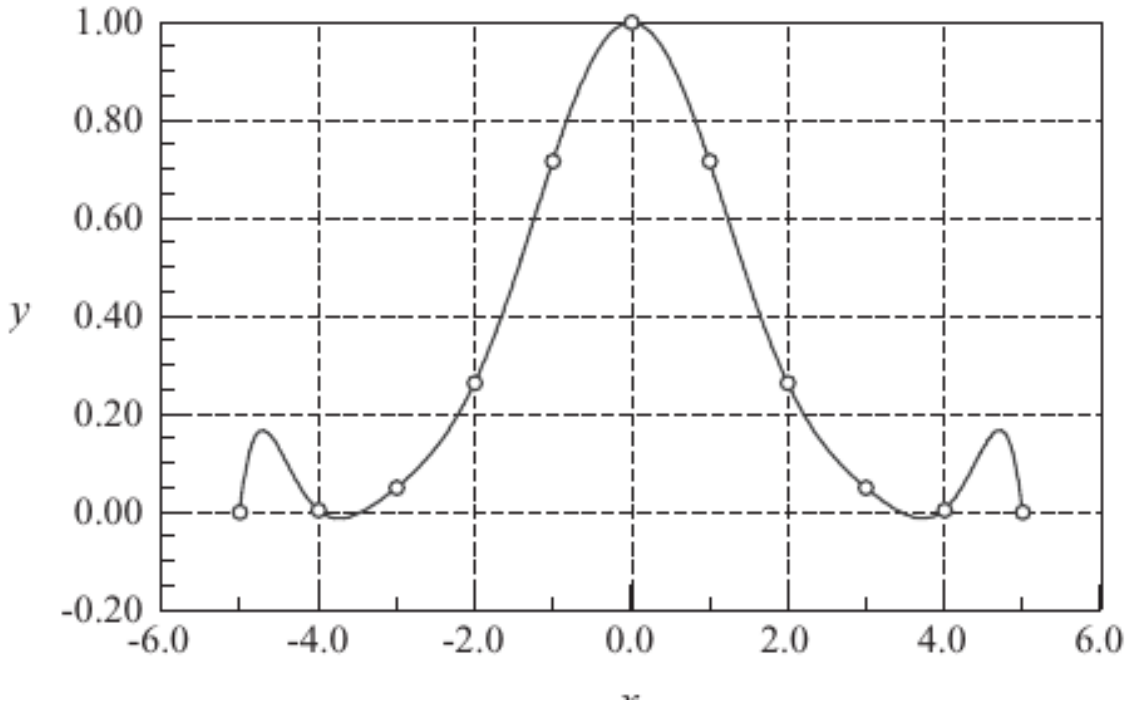
donde δ_{ij} es la delta de Kronecker se hace 0 si $i \neq j$ mientras es 1 si $i = j$. Se definen por

$$l_i(x) = \frac{x - x_0}{x_i - x_0} \frac{x - x_1}{x_i - x_1} \cdots \frac{x - x_{i-1}}{x_i - x_{i-1}} \frac{x - x_{i+1}}{x_i - x_{i+1}} \cdots \frac{x - x_n}{x_i - x_n}$$

$$l_i(x) = \prod_{j=0, j\neq i}^{n} \frac{x - x_i}{x_i - x_j}$$

Problemas de la interpolación

Si el orden de la interpolación polinómica es alto se producen irregularidades con oscilaciones importantes entre los puntos.



Lo mejor entonces es tomar una diferente perspectiva, polinomios lineales (rectas) que directamente vayan entre los puntos.

Requerimos que $P_{1i}(x_i) = y_i$, $P_{1i}(x_{i+1}) = y_{i+1}$.

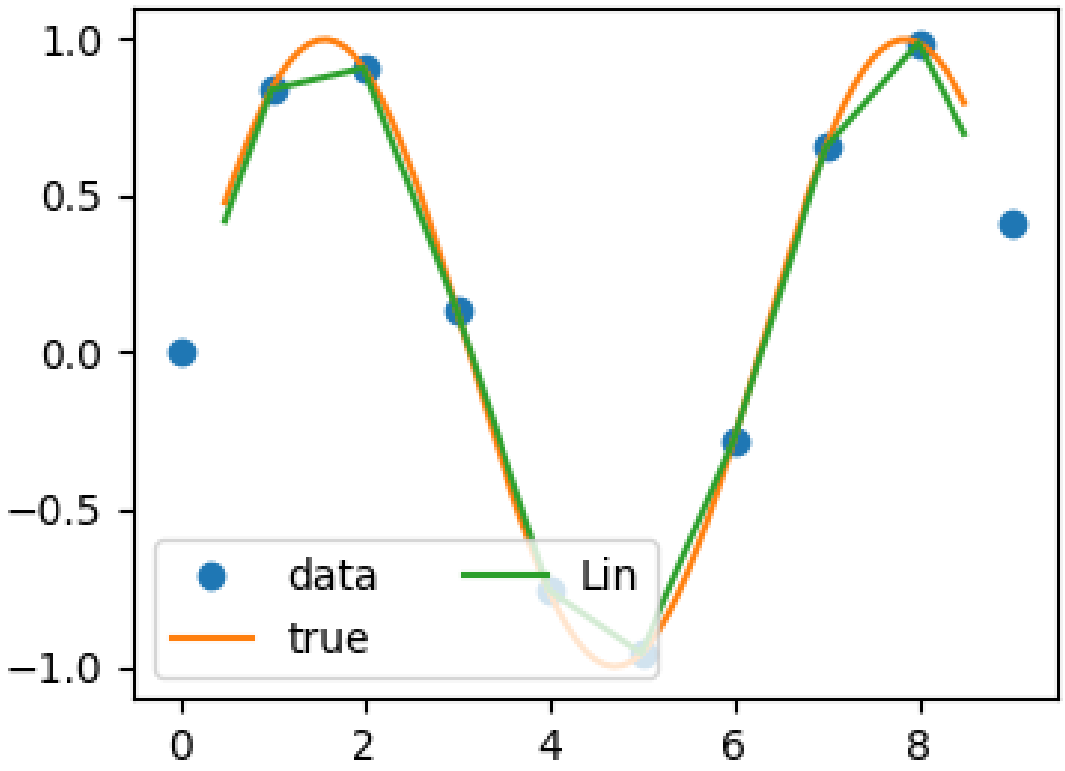
Lineal por pedazos

Usando el método de Lagrange:

$$P_1(x) = y_0 l_0(x) + y_1 l_1(x)$$

donde $l_0(x) = \frac{x-x_1}{x_0-x_1}$, $l_1(x) = \frac{x-x_0}{x_1-x_0}$.

Esto sería en forma recursiva para cada par de puntos, x_i, x_{i+1}.

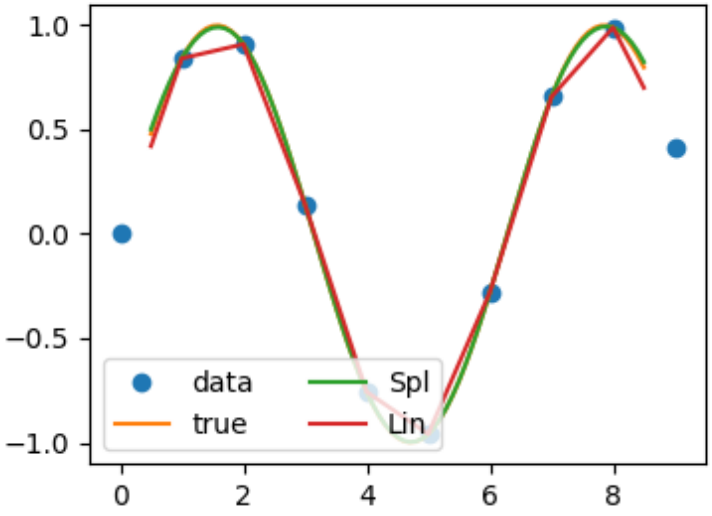


Interpolación con splines cúbicos

En general es un buen aproximador, pero va a estar lleno de quiebres debido a que son lineas rectas entre los puntos.
Cuando estamos en un pico es cuando el efecto es mas notable.
Para que lo que tengamos sea una función suave, vamos a proponer que entre punto y punto haya polinomios cúbicos. De esta manera además de exigir que la función pase por los puntos vamos a exigir otras condiciones.
Condiciones:

- Los polinomios cúbicos pasen por los puntos.
- Las derivadas primeras de dos polinomios sea la misma en los puntos (continuidad de la funcion y de la derivada primera.
- Las derivadas segundas sean iguales.
- Las derivadas 2das sean 0 en los puntos extremos.

Interpolación con splines cúbicos



Listas para manejo de funciones matemáticas

Supongamos que queremos trabajar con los puntos de la función $f(x) = x * sin(x^2)$ en el intervalo $[0, \sqrt{2\pi}]$. Resolución de 100 puntos.

```
import math as m
n=100
dx=m.sqrt(2*m.pi)/(n-1.0)
x=[] #crea una lista vacia
y=[]
for i in range(n):
    x1=i*dx
    x.append(x1)
    y1=funcion(x1)
    y.append(y1)
```

Listas para manejo de funciones matemáticas

La función viene dada por:

```
def funcion(x):
    f=x*m.sin(x**2)
    return f
```

Para generar listas en forma pythonica es:

```
x=[i*dx for i in range(n)]
y=[funcion(x1) for x1 in x]
```

NumPy: Vectores

Para realizar cálculos científicos y trabajar con vectores, matrices, tensores, etc existe una librería específica: Numerical Python “numpy”.

```
import numpy  as np
```

En general a los vectores, matrices o tensores de cualquier rango, se les llama “arrays”.
Para convertir una lista en un array, array:

```
ar=np.array([5.0,2.3,7.2])
```

```
ar=np.array(lista)
```

Para generar un array, zeros, es decir lo llenamos de 0s para generarlo.

```
ar=np.zeros(n)
```

Esto genera un vector de n componentes.

Generación de matrices

Para generar una matriz de n filas por m columnas hacemos

```
mtx=np.zeros([n,m])
```

No olvidar los corchetes.

Para acceder a los elementos de un array se hace de la misma manera que en las listas. Si es un vector

```
a[5]; a[0:2]; a[3:]; a[2,-3]
```

Si es una matriz:

```
a[0,5];a[3,5:7]
```

Las matrices también pueden definirse a través de listas:

```
  a=[[1,2],[3,4]]
```

Para acceder a un elemento de la lista de listas se tiene que hacer a[0][1]

```
  anp=np.array(a)
```

Matriz identidad

Si queremos generar la matriz identidad:

```
identidad=eye(N); identidad2=eye(N,M)
```

Si queremos generar un vector o una matriz de unos:

```
unos=ones(N); unos2=ones(N,M)
```

Si queremos extraer la diagonal de una matriz

```
A.diagonal()
```

Operaciones con matrices

Multiplicación por un escalar:

```
mtx1=alpha*mtx2; mtx3=alpha*ones(N)
```

Multiplicación entre vectores o matrices:

```
mtx1=np.matmul(mtx2,mtx3); alpha=np.dot(v2,v3)
```

Una nueva forma de escribir el producto matricial en python 3 es:

```
mtx1=mtx2 @ mtx3
```

Funciones matemáticas

Evaluación de funciones matemáticas que dependen de arrays:

```
v2=np.exp(-v1); v3=np.log(v1); v4=np.sin(v1)
```

```
import numpy as np
import time
n=300
A=np.random.normal(0,1,[n,n,n])
B=np.zeros([n,n,n])

t0=time.time()
for i in range(n):
    for j in range(n):
        for k in range(n):
            B[i,j,k]=np.exp(A[i,j,k])
print('Termino el for en: ',time.time()-t0)

t0=time.time()
B=np.exp(A)
print('Termino la eval matricial en: ',time.time()-t0)
```

Esto es mucho mas eficiente que realizar evaluaciones componente a componente con for Siempre se debe intentar trabajar con vectores y matrices.

Generación de vectores uniformes

Cuando hacemos una tabla para graficación en general necesitamos generar puntos equiespaciados entre un valor mínimo y un máximo.
La funcion np.linspace nos genera el vector automaticamente:

```
xvec= np.linspace(xmin,xmax,1000)
```

Esto nos genera un vector que comienza con el valor xmin y determina con el valor xmax y tiene 1000 puntos.
Cual es la resolución que tienen los puntos?

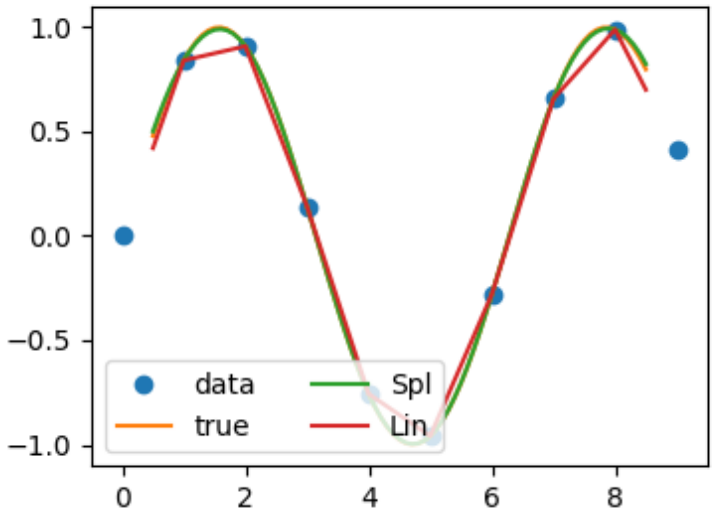
$$dx = (xmax - xmin)/(nptos - 1)$$

Entonces si hacemos:

```
xvec=[xmin+i*dx for i in range(nptos)]
```

Cual es la diferencia?
En numpy tenemos la instrucción: np.arange(1.5,10,2.3) similar a range pero se puede utilizar con flotantes.

Interpolación con splines cúbicos




```
import numpy as np
from scipy import interpolate as interp
x = np.arange(10)
y = np.sin(x)
cs = interp.CubicSpline(x, y)
y_cs=cs(xs)
```

Operaciones con las componentes

Algunas operaciones que podemos realizar con las componentes de un array:

```
a = np.array([2, 4, 3], float)
a.sum()
a.prod()
```

Alternativa:

```
np.sum(a)
np.prod(a)
```

Si quiero ver las numerosas funciones/metodos que tienen los arrays hacer:

```
dir(a)
```

Operaciones estadísticas

```
a = np.array([2, 4, 3], float)
a.mean()
a.std()
a.var()
```

Ejercicio: Tenemos un array pesos donde se tienen todos los pesos medidos (E.g. `pesos=np.randon.normal(15,3,100)`) y se quiere sacar la media y el error de las mediciones.

Máximo y mínimo

Para obtener el valor máximo o mínimo de los elementos de un array se debe hacer:

```
a.min()
a.max()
```

Si en cambio queremos determinar los índices del elemento donde se encuentra el máximo o mínimo se debe hacer:

```
a.argmin()
a.argmax()
```

Operaciones lógicas con arrays

Comparación de dos arrays. Esto lo realiza elemento por elemento:

```
>>> a = np.array([1, 3, 0], float)
>>> b = np.array([0, 3, 2], float)
>>> c= a > b
>>> print c
array([ True, False, False], dtype=bool)
```

Igualdad:

```
>>> a == b
array([False, True, False], dtype=bool)
```

Comparación con un valor:

```
>>> c= a > 2
>>> print c
array([ False, True, False], dtype=bool)
```

Operaciones lógicas con arrays

Si quiero combinar dos arrays logicos con los operadores logicos hay que usar: $||$,$\&$

```
>>> c = a == b | a > 2
array([False, True, False], dtype=bool)
```

También puedo devolver los índices donde se cumple un condicional:

```
>>> a = np.array([1, 3, 0,-2])
>>> ind=np.where(a > 0)
>>> sqrta = np.sqrt(a[ind])
```

Luego lo puedo usar para hacer evaluaciones de ese u otro array en esos índices.

Para todo el array

Si queremos saber si se satisface para cualquier elemento del array o para todo elemento del array. Se realiza primero la operación lógica y luego

```
  >>> c = np.array([ True, False, False], bool)
>>> any(c)
True
>>> all(c)
False
```

Selección de elementos

Supongamos que queremos realizar la raiz cuadrada a los elementos positivos de un array.

```
  >>> a = np.array([[4, -1, 9]], float)
>>> a >= 0
array([[ True, False, True], dtype=bool)
>>> np.sqrt(a[a >= 0])
array([ 2., 3.])
```

De lo contrario se puede guardar en un array lógico:

```
>> sel = (a >= 0)
a = np.array([[4, -1, 9]], float)
>>> np.sqrt(a[sel])
array([ 2., 3.])
```

Guardado de arrays

En formato ascii:

```
>>> a = np.array([1, 2, 3, 4])
>>> np.savetxt('test1.txt', a, fmt='%d')
>>> b = np.loadtxt('test1.txt', dtype=int)
```

En formato binario (recomendado):

```
>>> np.save('test3.npy', a)
>>> d = np.load('test3.npy')
```

La extensión *npy* es la que se utiliza para datos binarios en python. Si uds no la agregan python la tomará por default.