



Curso Python Científico: **Scipy**

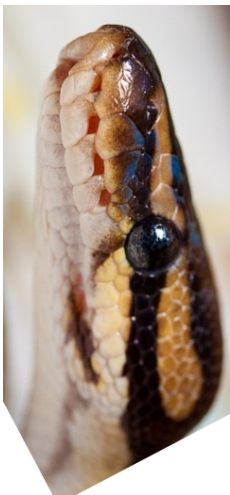
Módulo 3

Cesar Husillos Rodríguez

IAA-CSIC

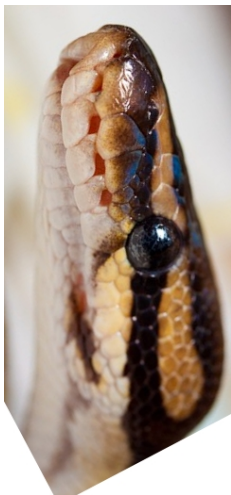
Septiembre de 2014

Índice



- 1 Introducción
- 2 Búsqueda de mínimos
- 3 Búsqueda de raíces
- 4 Ajuste de curvas
- 5 Interpolación
- 6 Integración
- 7 Estadística y Distribuciones
- 8 Álgebra lineal
- 9 Procesamiento de señales
- 10 Agrupamiento
- 11 Magnitudes Físicas

Índice



- 1 **Introducción**
- 2 Búsqueda de mínimos
- 3 Búsqueda de raíces
- 4 Ajuste de curvas
- 5 Interpolación
- 6 Integración
- 7 Estadística y Distribuciones
- 8 Álgebra lineal
- 9 Procesamiento de señales
- 10 Agrupamiento
- 11 Magnitudes Físicas

¿Qué es SCIPY?

- **Paquete** que extiende la funcionalidad de NUMPY.
- SCIPY implementa módulos que realizan tareas de optimización, integración, agrupamientos, álgebra lineal... y que dan acceso a funciones especiales, distribuciones de probabilidad y constantes científicas, entre otros.

NOTA: Los ejemplos y ejercicios que se proponen han sido probados en versiones iguales o superiores a la 0.12 (*Mayo 2013*).

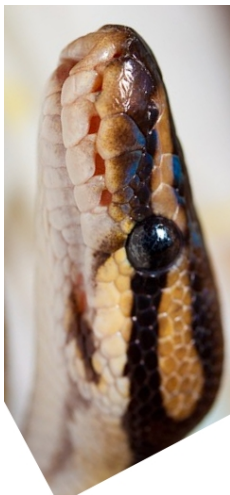
subpaquetes SCIPY

Subpaquete	Descripción
cluster	Algoritmos de clustering
constants	Constantes físicas y matemáticas
fftpack	Rutinas de Transformadas Rápidas de Fourier
integrate	Para integrar y resolver ecuaciones diferenciales ordinarias
interpolate	Rutinas de interpolación.
io	Entrada y salida (Input/Output)
linalg	Álgebra Lineal
maxentropy	Métodos de Máxima Entropía
ndimage	Procesado de imágenes n-dimensionales
odr	Ajustes a distancias ortogonales (Orthogonal Distance Regression)
optimize	Procedimientos de optimización y búsqueda de ceros en funciones
signal	Procesado de señal
sparse	Matrices dispersas y rutinas asociadas
spatial	Datos con estructura espacial y algoritmos
special	Funciones especiales
stats	Distribuciones estadísticas y funciones
weave	Integración con C/C++

¿Qué vamos a aprender?

- 1 Determinar los valores mínimos de funciones.
- 2 Localizar las raíces de una función.
- 3 Interpolación n-dimensional.
- 4 Ajuste de curvas.
- 5 Integración de funciones multivariables.
- 6 Estadística descriptiva.
- 7 Manejar distribuciones estadísticas. Verificar qué parámetros de una distribución se ajustan a una muestra experimental.
- 8 Realizar operaciones sobre matrices.
- 9 Resolver sistemas de ecuaciones lineales.
- 10 Clasificación de datos.
- 11 Acceder a información de constantes físicas.

Índice



- 1 Introducción
- 2 **Búsqueda de mínimos**
- 3 Búsqueda de raíces
- 4 Ajuste de curvas
- 5 Interpolación
- 6 Integración
- 7 Estadística y Distribuciones
- 8 Álgebra lineal
- 9 Procesamiento de señales
- 10 Agrupamiento
- 11 Magnitudes Físicas

Búsqueda de mínimos

`scipy.optimize`

Dentro del subpaquete `scipy.optimize`, la función

```
minimize(fun, x0[, args=(), method='BFGS',  
jac=None, hess=None, hessp=None, bounds=None,  
constraints=(), tol=None, callback=None,  
options=None])
```

Proporciona una **interfaz común** para todos los **algoritmos de minimización** que se apliquen sobre **funciones escalares multivariables**.

Parámetros

minimize

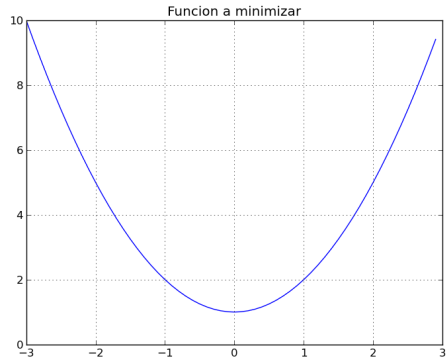
Parámetro	Tipo	Descripción
<code>fun</code>	Función	El nombre de una función.
<code>x0</code>	ndarray	Punto inicial al partir del se busca el mínimo.
<code>args</code>	tupla	(Opcional) Parámetros extra si los necesitan la <code>fun</code> y sus derivadas.
<code>method</code>	string	(Opcional) Método de búsqueda de mínimo. Posibles: 'Nelder-Mead', 'Powell', 'CG', 'BFGS', 'Newton-CG', 'Anneal', 'L-BFGS-B', 'TNC', 'COBYLA' y 'SLSQP'.
<code>jac</code>	Booleano o función	(Opcional) Si es <code>True</code> se supone que <code>fun</code> devuelve el valor del Jacobiano junto al de la función. Si es <code>False</code> , <code>jac</code> se estima numéricamente. Puede ser una función, en cuyo caso recibe los mismos parámetros que <code>fun</code> .
<code>hess</code>	Función	(Opcional) Hessiano de <code>fun</code> . (Sólo para Newton-CG.)
<code>bounds</code>	secuencia	(Opcional) (Sólo para L-BFGS-B, TNC, COBYLA y SLSQP) Límites para las variables. Un par (<code>min</code> "min" None, <code>max</code> "max" None) por cada variable de <code>fun</code> .
<code>tol</code>	float	(Opcional) Tolerancia por debajo de la cual se detienen las iteraciones.
<code>options</code>	dict	(Opcional) Las opciones aceptadas por todos los métodos son: <code>maxiter</code> (int); <code>disp</code> (bool) Si <code>True</code> , info de convergencia.

Ejemplo 1

Minimización

Ejemplo

```
# Funcion a minimizar  
def parabola(x):  
    return x ** 2 + 1  
  
# jacobiano de la funcion  
def parabola_derivada(x):  
    return 2 * x
```



Método SIMPLEX (Nelder-Mead)

Ejemplo 1

Es la forma más simple de determinar el mínimo de una función. Sólo necesita de una función a evaluar. Es lento.

```
>>> res = minimize(parabola, 202.34, method='nelder-mead', \
    options={'xtol': 1e-8, 'disp': True})
Optimization terminated successfully.
    Current function value: 1.000000
    Iterations: 38
    Function evaluations: 76

>>> res
    status: 0
    nfev: 76
    success: True
    fun: 1.0
     x: array([ -9.09494702e-13])
message: 'Optimization terminated successfully.'
    nit: 38
```

Método

Broyden-Fletcher-Goldfarb-Shanno

Ejemplo 1

Converge más rápidamente porque usa el gradiente de la función. Si no es dado, se estima.

Algoritmo de BFGS

```
>>> res = minimize(parabola, 202.34, method='BFGS', \
    options={'disp': True})
Optimization terminated successfully.
    Current function value: 1.000000
    Iterations: 4
    Function evaluations: 15
    Gradient evaluations: 5
```

Método Newton del gradiente conjugado

Ejemplo 1

Algoritmo de Newton-CG

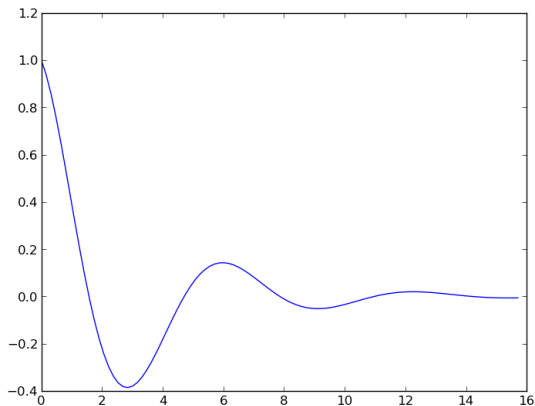
```
>>> res = minimize(parabola, 202.34, method='Newton-CG', \
    jac=parabola_derivada, options={'xtol': 1e-8, 'disp': True})
Optimization terminated successfully.
    Current function value: 1.000000
    Iterations: 3
    Function evaluations: 4
    Gradient evaluations: 7
    Hessian evaluations: 0
```

Más rápido aún. Posibilidad de insertar el Hessiano de la función.

Minimización

`scipy.optimize`

¿Qué sucede si hay mínimos locales?



Minimización

`scipy.optimize`

Coseno amortiguado

$$f(x) = \cos(x) \cdot e^{\frac{-x}{\pi}}$$

Derivada primera (jacobiano)

$$f'(x) = -e^{\frac{-x}{\pi}} \left[\cos(x) - \frac{1}{\pi} \sin(x) \right]$$

Derivada segunda (hessiano)

$$f''(x) = e^{\frac{-x}{\pi}} \left[\frac{2\sin(x)}{\pi} + \left(\frac{1}{\pi^2} - 1 \right) \cos(x) \right]$$

Minimización

scipy.optimize. Minimos locales

Coseno amortiguado y dos primeras derivadas

```
def coseno_amortiguado(x):  
    return scipy.cos(x) * scipy.exp(- x / scipy.pi)  
  
def cos_amort_derivada(x):  
    return -1 * scipy.exp(- x / scipy.pi) * \  
    (scipy.sin(x) + scipy.cos(x) / scipy.pi)  
  
def cos_amort_derivada2(x):  
    return scipy.exp(- x / scipy.pi) * \  
    (2 * scipy.sin(x) / scipy.pi + \  
    ((scipy.pi ** -2 - 1) * scipy.cos(x)))
```


Minimización

Mínimos locales

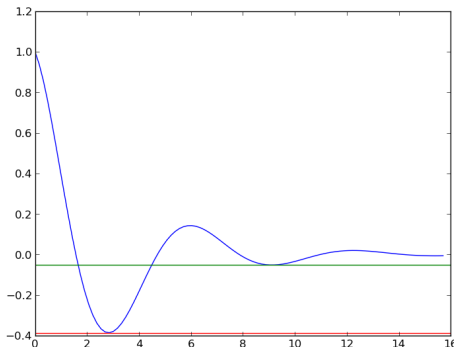
```
>>> res = scipy.optimize.minimize(coseno_amortiguado, \
... scipy.pi / 2, method='Newton-CG', \
... jac=cos_amort_derivada, hess=cos_amort_derivada2, \
... options={'xtol': 1e-8, 'disp': True})
>>> print res['x'], res['fun']
[ 2.83342358] [-0.38667837]
>>>
>>> res1 = scipy.optimize.minimize(coseno_amortiguado, \
... 3.22 * scipy.pi, method='Newton-CG', \
... jac=cos_amort_derivada, hess=cos_amort_derivada2, \
... options={'xtol': 1e-8, 'disp': True})
>>> print res1['x'], res1['fun']
[ 9.11660888] [-0.05233123]
```

Minimización

Mínimos locales

¿Qué sucede si hay mínimos locales?

El mínimo obtenido depende del punto inicial para la evaluación de la función.



Minimización

Rango restringido de búsqueda

Cuando interesa obtener el valor mínimo de una función en un rango concreto, lo hacemos explícito a través del parámetro `bounds` de la función `minimize`.

- Válido para métodos: L-BFGS-B, TNC, COBYLA y SLSQP.
- `bounds` es una tupla (de tuplas).
- La longitud de la tupla es la misma que el número de variables de la función.
- Para cada variable independiente, los límites se dan como una tupla de valores (`min`, `max`).

Como ejemplo, para la función

$$f(x, y, z) = x^2 + xy + 3z$$

`bounds` sería algo así como

$$((1, 5), (None, 4), (None, None))$$

donde $x \in [1, 5]$, $y \in (-\infty, 4)$ y z puede tomar cualquier valor.

Minimización

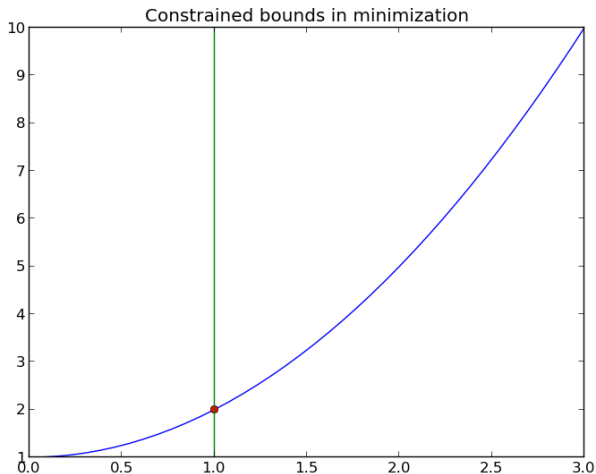
Rango restringido para búsqueda de mínimos. Ejemplo

```
>>> import scipy
>>> import scipy.optimize
>>> from matplotlib import pyplot
>>> res = scipy.optimize.minimize(lambda x: x ** 2 + 1, 2, \
... method='L-BFGS-B', bounds=((1, None),))
>>> print res
status: 0
success: True
nfev: 2
fun: array([ 2.])
x: array([ 1.])
message: 'CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_ <= _PGTOL'
jac: array([ 1.99999999])
nit: 1
>>> x = scipy.linspace(0, 3, 40)
>>> y = x ** 2 + 1
>>> pyplot.plot(x, y)
>>> pyplot.plot(res['x'], res['fun'], 'ro')
>>> pyplot.axvline(1, color='g')
>>> pyplot.title('Constrained bounds in minimization')
```



Minimización

Rango restringido para búsqueda de mínimos. Ejemplo

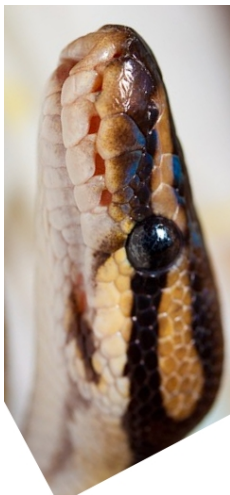


Ejercicios

Ejercicios 1 y 2

<http://www.iaa.es/python/cientifico/ejercicios/scipy>

Índice



- 1 Introducción
- 2 Búsqueda de mínimos
- 3 **Búsqueda de raíces**
- 4 Ajuste de curvas
- 5 Interpolación
- 6 Integración
- 7 Estadística y Distribuciones
- 8 Álgebra lineal
- 9 Procesamiento de señales
- 10 Agrupamiento
- 11 Magnitudes Físicas

Búsqueda de raíces

(`scipy.optimize`)

1 Para una función escalar.

Se trata de buscar las coordenadas para las cuales la función toma valores nulos.

2 Para un conjunto de ecuaciones.

En este caso, se trata de encontrar los puntos de corte de las funciones definidas por las ecuaciones.

Búsqueda de raíces

(`scipy.optimize`)

Dentro del subpaquete `scipy.optimize`, la función

```
scipy.optimize.root(fun, x0, args=())
```

Permite resolver ecuaciones escalares y sistemas de ecuaciones (lineales o no).

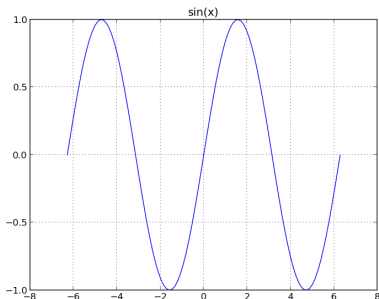
Parámetro	Tipo	Descripción
<code>fun</code>	Función	El nombre de una función.
<code>x0</code>	<code>ndarray</code>	Punto inicial al partir del se busca la raíz.
<code>args</code>	tupla	Parámetros extra si los necesitan la <code>fun</code> y sus derivadas.

Búsqueda de raíces

Ecuación escalar

Trabajamos con una función sobradamente conocida: $\sin(x)$. Se trata de resolver la ecuación

$$\sin(x) = 0$$



Búsqueda de raíces

Ecuación escalar

- Siempre que hay un punto de evaluación inicial, el resultado dependerá de esa elección.
- Podemos obviar esa dificultad evaluando en múltiples puntos del dominio de la función considerado en la forma:

```
>>> import scipy
>>> import scipy.optimize
>>> res = scipy.optimize.root(scipy.sin, [-1, 0, 1])
```

donde, como hemos visto, `res` es un diccionario. Sus claves son:

```
>>> res.keys()
... ['status', 'success', 'qtf', 'nfev', 'r', 'fun', 'x',
... 'message', 'fjac']
```

Búsqueda de raíces

Ecuación escalar

De las claves del diccionario `res` interesan

- `fun`, que da el resultado de la función evaluada en las raíces.
(Los valores obtenidos serán 0 o muy cercanos a 0.)
- `x`, que da las coordenadas de las raíces de la función.

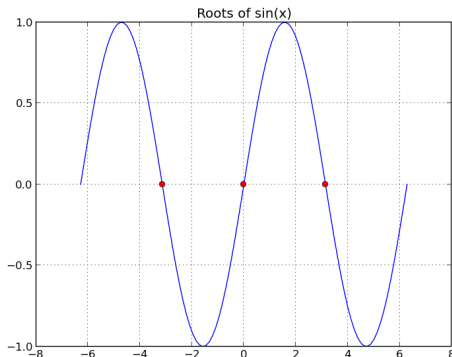
En nuestro caso...

```
>>> res['x']  
... array([-3.14159265,  0.          ,  3.14159265])  
>>> res['fun']  
... array([ 2.42794333e-13,  0.00000000e+00,  2.43483352e-13])
```

Búsqueda de raíces

Ecuación escalar

Grafico las raíces obtenidas



Búsqueda de raíces

Sistema de ecuaciones

Resolver un sistema de ecuaciones implica obtener los valores de x e y que cumplen de forma simultánea todas las ecuaciones del conjunto.

Tomemos como ejemplo, el siguiente sistema

$$\begin{aligned}y &= \sin(x) \\ y &= 0,1 \cdot x + 0,3\end{aligned}$$

en el intervalo de $x \in [-5, 5]$.

Búsqueda de raíces

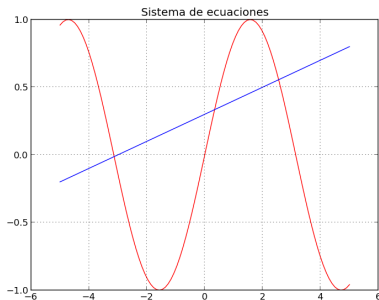
Sistema de ecuaciones

```
>>> import scipy
>>> from matplotlib import pyplot
>>> x = scipy.linspace(-5, 5, 100)
>>> y = scipy.sin(x)
>>> y2 = 0.1*x+0.3
>>> pyplot.plot(x, y, 'r-', x, y2, 'b-')
>>> pyplot.title('Sistema de ecuaciones')
>>> pyplot.grid(True)
>>> pyplot.show()
```

Búsqueda de raíces

Sistema de ecuaciones

Problema: obtener los puntos de corte



Búsqueda de raíces

Sistema de ecuaciones

Dado el sistema

$$\begin{aligned}y &= \sin(x) \\ y &= 0,1 \cdot x + 0,3\end{aligned}$$

tengo que localizar las raíces de la ecuación

$$\sin(x) - 0,1 \cdot x - 0,3 = 0$$

y eso, sabemos hacerlo :-)

Búsqueda de raíces

Sistema de ecuaciones

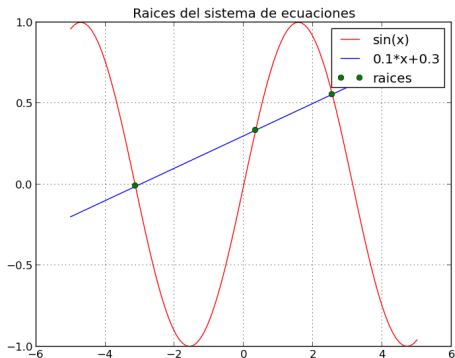
```
>>> import scipy
>>> import scipy.optimize
>>> def func(x):
...     return scipy.sin(x) - 0.1 * x - 0.3
...
>>> x0s = scipy.linspace(-5, 5, 5)
>>> sol = scipy.optimize.root(func, x0s)
>>> print sol['x']
... [-14.0099837 , -3.13130629,  0.34024025,  2.55290884,
14.03909441]
>>> abool = (-5 <= sol['x']) & (sol['x'] <= 5) # filtro en intervalo
>>> nx = scipy.compress(abool, sol['x'])
>>> ny = scipy.sin(nx)
>>> # Grafico
>>> pyplot.plot(x, y, 'r-')
>>> pyplot.plot(x, y2, 'b-')
>>> pyplot.plot(nx, ny, 'go')
>>> pyplot.grid(True)
>>> pyplot.legend(['sin(x)', '0.1*x+0.3', 'raices'])
>>> pyplot.show()
```



Búsqueda de raíces

Ecuación escalar

Solución al sistema de ecuaciones

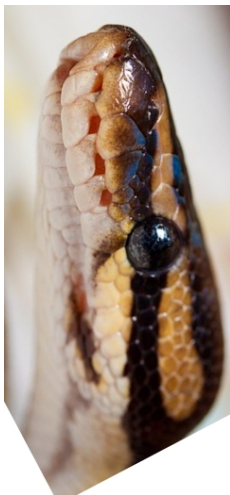


Ejercicios

Ejercicio 3

<http://www.iaa.es/python/cientifico/ejercicios/scipy>

Índice



- 1 Introducción
- 2 Búsqueda de mínimos
- 3 Búsqueda de raíces
- 4 **Ajuste de curvas**
- 5 Interpolación
- 6 Integración
- 7 Estadística y Distribuciones
- 8 Álgebra lineal
- 9 Procesamiento de señales
- 10 Agrupamiento
- 11 Magnitudes Físicas

Ajuste de curvas

`scipy.optimize.curve_fit`

De nuevo dentro del subpaquete `scipy.optimize`, la función

```
scipy.optimize.curve_fit(func, xdata, ydata, p0=None)
```

emplea un ajuste por mínimos cuadrados no lineales para establecer los parámetros de la función `func` que mejor se ajustan a los datos (`xdata`, `ydata`).

El parámetro `p0` es una tupla que representa los valores iniciales para los parámetros de `func`.

Devuelve una tupla de 2 elementos:

- el primer elemento es un array con los parámetros de la función que minimizan la suma cuadrática de los errores.
- la matriz de covarianza de la solución. La diagonal corresponde a las varianzas de cada uno de los parámetros.

Ajuste de curvas

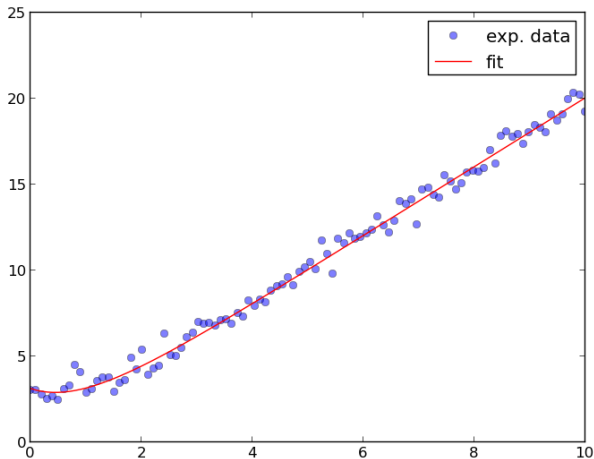
`scipy.optimize.curve_fit`

Ajuste la función con ruido normal gaussiano escalado por 0.5 a la función $y = 3 \cdot e^{-x} + 2 \cdot x$.

```
>>> def func_fit(x, a, b):  
...     return a * scipy.exp(-x) + b * x  
...  
>>> x = scipy.linspace(0, 10, 100)  
>>> y = 3 * scipy.exp(-x) + 2 * x  
>>> noise = scipy.randn(100) * 0.5  
>>> y += noise  
>>> res = scipy.optimize.curve_fit(func_fit, x, y)  
>>> print res  
(array([ 2.97253275,  1.99997092]), array([[ 3.61334133e-02,  
-1.06635306e-04],  
      [ -1.06635306e-04,   5.89627071e-05]]))
```

Ajuste de curvas

`scipy.optimize.curve_fit`

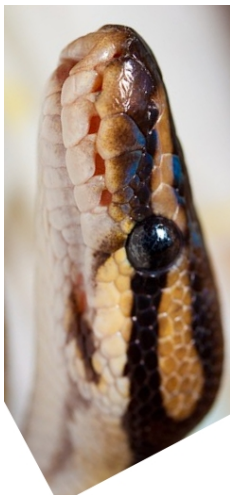


Ejercicios

Ejercicio 4

<http://www.iaa.es/python/cientifico/ejercicios/scipy>

Índice



- 1 Introducción
- 2 Búsqueda de mínimos
- 3 Búsqueda de raíces
- 4 Ajuste de curvas
- 5 Interpolación**
- 6 Integración
- 7 Estadística y Distribuciones
- 8 Álgebra lineal
- 9 Procesamiento de señales
- 10 Agrupamiento
- 11 Magnitudes Físicas

Interpolación

`scipy.interpolate`

Definiciones

- “En el subcampo matemático del **análisis numérico**, se denomina interpolación a la obtención de nuevos puntos partiendo del conocimiento de un conjunto discreto de puntos.”
- “En **ingeniería** y **algunas ciencias** es frecuente disponer de un cierto número de puntos obtenidos por muestreo o a partir de un experimento y pretender construir una función que los ajuste.”

Obtenidas de la Wikipedia (<http://es.wikipedia.org/wiki/Interpolación>)

Interpolación

`scipy.interpolate`

Hay dos formas básicas de interpolación:

- 1 Ajustar todo el conjunto de datos a una función.
- 2 Ajustar diferentes subconjuntos a distintas funciones.

El tipo 2 se denomina *interpolación con splines*. Se emplea sobre todo cuando la forma funcional de los datos es compleja.

Veremos ejemplos de todos estos tipos.

Interpolación

Splines

Una función spline está formada por varios polinomios, cada uno definido sobre un subintervalo, que se unen entre sí obedeciendo a ciertas condiciones de continuidad.

Supongamos que disponemos de $n + 1$ puntos, a los que denominaremos **nudos**, tales que $t_0 < t_1 < \dots < t_n$. Supongamos además que se ha fijado un entero $k > 0$. Decimos entonces que una función spline de **grado k** con nudos en t_0, t_1, \dots, t_n es una función S que satisface las condiciones:

- 1 en cada intervalo (t_{i-1}, t_i) , S es un polinomio de grado menor o igual a k .
- 2 S tiene una derivada de orden $(k - 1)$ continua en $[t_0, t_n]$.

Interpolación-1D: sobre muestra completa

`scipy.interpolate.interp1d`

Para interpolar todos los datos de una función unidimensional usaremos

```
scipy.interpolate.interp1d(x, y, kind='linear')
```

Parámetro	Tipo	Descripción
x	array	Array de números ordenados de forma monótona creciente .
y	array	Array o matriz de valores reales. El tamaño de y a lo largo del eje de interpolación debe ser el mismo que el de x.
kind	string o entero	Si es string puede tomar valores entre: 'linear', 'nearest', 'zero', 'slinear', 'quadratic' o 'cubic'. Si es entero especifica el orden del <i>spline</i> usado para interpolar.
axis	entero	Especifica el eje del array N-dimensional y a usar.

Devuelve una **función**. Cuando se la llama (pasándole nuevas coordenadas), devuelve los valores interpolados.

Ejemplo 1

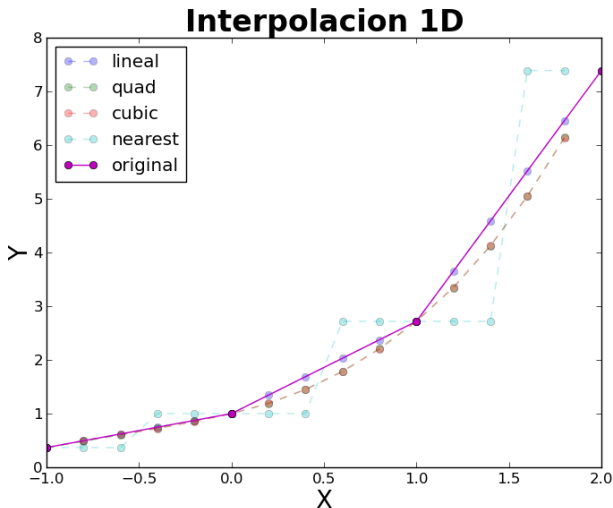
Interpolación-1D: sobre muestra completa

Código del ejemplo

```
>>> from scipy.interpolate import interp1d
>>> x = scipy.array([-1, 0, 1, 2])
>>> y = scipy.exp(x)
>>> flin = interp1d(x, y)
>>> fqua = interp1d(x, y, kind='quadratic')
>>> fcub = interp1d(x, y, kind='cubic')
>>> fnea = interp1d(x, y, kind='nearest')
>>> newx = scipy.arange(-1, 2, 0.2)
>>> newylin = flin(newx)
>>> newyqua = fqua(newx)
>>> newycub = fcub(newx)
>>> newynea = fnea(newx)
```

Ejemplo 1

Interpolación-1D: sobre muestra completa






Interpolación-1D: uso de splines

scipy.interpolate.UnivariateSpline

```
scipy.interpolate.UnivariateSpline(x, y, w=None,  
    bbox=[None, None], k=3, s=None)
```

Parámetro	Tipo	Descripción
x	array	Array 1D, ordenados de forma monótona creciente .
y	array	Array 1D de valores de la variable dependiente. Mismo número de
w	array	Array 1D de pesos para el ajuste con splines.
bbox	(tupla, lista o array)	Si toma valor None (por defecto), todos los pesos son iguales. (Opcional) Especifica los límites del intervalo a
k	entero	interpoliar. Si vale None (por defecto), $\text{bbox}=[x[0], x[-1]]$
s	float	(Opcional) Grado de suavizado del spline (≤ 5). (Opcional) Factor de suavizado empleado para elegir el número de nodos. Dicho número aumenta hasta que la condición de suavizado se alcanza: $\text{sum}((w[i]*(y[i]-s(x[i])))**2, \text{axis}=0) \leq s$ Si vale None (por defecto) $s = \text{len}(w)$

Devuelve una **función**. Cuando se la llama (pasándole nuevas coordenadas), devuelve los valores interpolados. ▶  ▶  ▶ 

Ejemplo 2

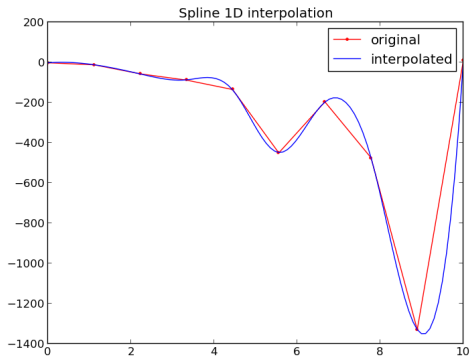
Interpolación-1D: uso de splines

Código del ejemplo

```
>>> import scipy
>>> import scipy.interpolate
>>> from matplotlib import pyplot
>>> x = scipy.linspace(0, 10, 10) # muestra inicial
>>> y = x ** 3 * scipy.sin(2 * x) - 9 * x ** 2 - 2.
>>> # interpolate
>>> sp1d = scipy.interpolate.UnivariateSpline(x, y)
>>> nx = scipy.linspace(0, 10, 100) # nueva muestra
>>> ny = sp1d(nx)
>>> pyplot.plot(x, y, 'r.-')
>>> pyplot.plot(nx, ny, 'b-')
>>> pyplot.title('Spline 1D interpolation')
>>> pyplot.legend(['original', 'interpolated'], numpoints=1)
>>> pyplot.show()
```

Ejemplo2

Interpolación-1D: uso de splines



Interpolación

Splines con datos ruidosos

¿Podemos interpolar datos con ruido usando splines?

Sí, haciendo uso del *factor de suavizado* (parámetro `s`) de la función `UnivariateSpline`.

```
scipy.interpolate.UnivariateSpline(x, y, w=None,  
                                   bbox=[None, None], k=3, s=None)
```

Ejemplo 3

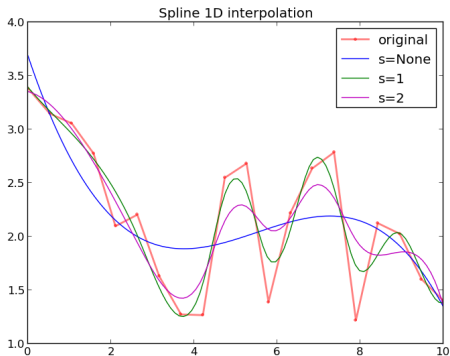
Interpolación-1D: uso de splines con datos ruidosos

Código del ejemplo

```
>>> x = scipy.linspace(0, 10, 20)
>>> y = scipy.cos(x) * scipy.exp(-x) + 1
>>> y += scipy.rand(20) * 2    # noise data
>>> sp1d = scipy.interpolate.UnivariateSpline(x, y)
>>> sp1d_s1 = scipy.interpolate.UnivariateSpline(x, y, s=1)
>>> sp1d_s2 = scipy.interpolate.UnivariateSpline(x, y, s=2)
>>> # nuevas muestras
>>> nx = scipy.linspace(0, 10, 100)
>>> ny = sp1d(nx)
>>> ny_s1 = sp1d_s1(nx)
>>> ny_s2 = sp1d_s2(nx)
>>> pyplot.plot(x, y, 'r.-', lw=2, alpha=0.5)
>>> pyplot.plot(nx, ny, 'b-')
>>> pyplot.plot(nx, ny_s1, 'g-')
>>> pyplot.plot(nx, ny_s2, 'm-')
>>> pyplot.title('Spline 1D interpolation')
>>> pyplot.legend(['original', 's=None', 's=1', 's=2'], numpoints=1)
>>> pyplot.show()
```

Ejemplo3

Interpolación-1D: uso de splines con datos ruidosos



Interpolación-2D

`scipy.interpolate.griddata`

```
scipy.interpolate.griddata(points, values,  
                           xi, method='linear', fill_value=nan)
```

Parámetro	Tipo	Descripción
points	array	Coordenadas. Array de forma (n, D) , o una tupla de $ndim$ arrays.
values	array	Valores asignados a las coordenadas dadas por points. Dimensión $(n,)$.
xi	array	Forma (M, D) . Puntos en los que interpolar.
method	string	(Opcional). Método de interpolación. Valores posibles: <i>nearest</i> , <i>linear</i> o <i>cubic</i> .
fill_value	float	(Opcional) Valor usado para la región fuera de la zona de interpolación definida por points. Si no se da, se toma nan. No tiene efecto para el método <i>nearest</i> .

Devuelve el conjunto de datos interpolados en `xi`.

Ejemplo4

Interpolación-2D

```
import scipy
from scipy.interpolate import griddata

def func(x, y): # funcion a interpolar
    return (scipy.sin(x) + scipy.cos(y)) * 10

intvx = scipy.linspace(0, 10, 100)
intvy = scipy.linspace(0, 10, 200)
grid_x, grid_y = scipy.meshgrid(intvx, intvy)
points = scipy.random.rand(1000, 2) * 10
values = func(points[:, 0], points[:, 1])
grid_z0 = griddata(points, values, (grid_x, grid_y), \
    method='nearest')
grid_z1 = griddata(points, values, (grid_x, grid_y), \
    method='linear')
grid_z2 = griddata(points, values, (grid_x, grid_y), \
    method='cubic')
```

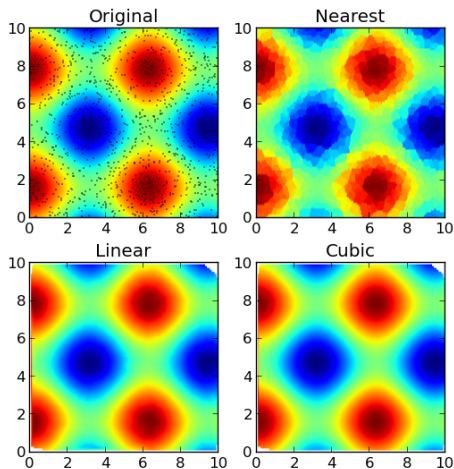

Ejemplo4

Interpolación-2D (Continuación)

```
# plotting
pyplot.subplot(221)
pyplot.imshow(func(grid_x, grid_y).T, extent=(0, 10, 0, 10), \
origin='lower')
pyplot.plot(points[:, 0], points[:, 1], 'k.', ms=1)
pyplot.title('Original')
pyplot.subplot(222)
pyplot.imshow(grid_z0.T, extent=(0, 10, 0, 10), origin='lower')
pyplot.title('Nearest')
pyplot.subplot(223)
pyplot.imshow(grid_z1.T, extent=(0, 10, 0, 10), origin='lower')
pyplot.title('Linear')
pyplot.subplot(224)
pyplot.imshow(grid_z2.T, extent=(0, 10, 0, 10), origin='lower')
pyplot.title('Cubic')
pyplot.gcf().set_size_inches(6, 6)
pyplot.show()
```

Ejemplo4

Interpolación-2D (Continuación)



Interpolación-2D

`scipy.interpolate.SmoothBivariateSpline`

```
scipy.interpolate.SmoothBivariateSpline(x, y,
    z, w=None, bbox=[None, None, None, None],
    kx=3, ky=3, s=None)
```

Parámetro	Tipo	Descripción
x	array	Coordenadas en x.
y	array	Coordenadas en y.
z	array	Coordenadas en z.
w	array	(Opcional). Secuencia de pesos. Misma dimensión que x, y y z.
bbox	float	(Opcional) Secuencia de tamaño 4, con los valores que limitan el dominio de x e y. Valores por defecto: <code>bbox=[min(x),max(x), min(y),max(y)]</code>
kx	entero	(Opcional) Grado del spline en x. Por defecto 3.
ky	entero	(Opcional) Grado del spline en y. Por defecto 3.
s	float	(Opcional) Valor positivo para el <i>factor de suavizado</i> .

Devuelve la función de interpolación.

Ejemplo5

Interpolación-2D con splines

```
import scipy
from scipy.interpolate import SmoothBivariateSpline
def func(x, y):
    return (scipy.sin(x) + scipy.cos(y)) * 10
intvx = scipy.linspace(0, 10, 100) # dominio de la grafica
intvy = scipy.linspace(0, 10, 200)
points = scipy.random.rand(1000, 2) * 10 # puntos para interpolar
z = func(points[:, 0], points[:, 1])
spline2d = SmoothBivariateSpline(points[:, 0], points[:, 1], z, \
s=0.01, kx=4, ky=4)
```

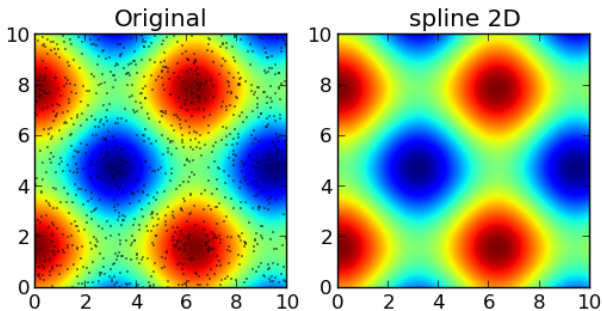
Ejemplo5

Interpolación-2D con splines (Continuación)

```
# plotting
pyplot.subplot(121)
pyplot.imshow(func(grid_x, grid_y).T, extent=(0, 10, 0, 10), \
origin='lower')
pyplot.plot(points[:, 0], points[:, 1], 'k.', ms=1)
pyplot.title('Original')
pyplot.subplot(122)
pyplot.imshow(spline2d(intvx, intvy), extent=(0, 10, 0, 10), \
origin='lower')
pyplot.title('spline_2D')
pyplot.gcf().set_size_inches(6, 3)
pyplot.show()
```

Ejemplo5

Interpolación-2D con splines (Continuación)

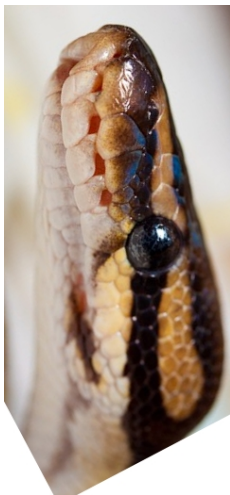


Ejercicios

Ejercicios 5 y 6

<http://www.iaa.es/python/cientifico/ejercicios/scipy>

Índice



- 1 Introducción
- 2 Búsqueda de mínimos
- 3 Búsqueda de raíces
- 4 Ajuste de curvas
- 5 Interpolación
- 6 Integración**
- 7 Estadística y Distribuciones
- 8 Álgebra lineal
- 9 Procesamiento de señales
- 10 Agrupamiento
- 11 Magnitudes Físicas

Integración

`scipy.integrate`

- El subpaquete `scipy.integrate` proporciona rutinas para integrar funciones de una o varias variables.
- También para integrar ecuaciones diferenciales ordinarias.

Integración en una variable

`scipy.integrate.quad`

La función

```
quad(func, a, b, args=())
```

sirve para integrar la función `func` de una variable entre dos puntos (`a` y `b`). `args` es una tupla con parámetros adicionales para `func`.

Los límites de integración pueden ser `scipy.inf` para indicar los límites infinitos.

Retorna una tupla:

```
(valor de la integral, estimación del error absoluto  
en el resultado)
```

Integración en una variable

`scipy.integrate.quad`

```
>>> def const(x):  
...     return 1  
...  
>>> import scipy.integrate  
>>> # Funcion constante, de valor 1, en [0, 1]  
>>> # Es un CUADRADO  
>>> res = scipy.integrate.quad(const, 0, 1)  
>>> print res  
(1.0, 1.1102230246251565e-14)
```

Integración

`scipy.integrate`

Hemos integrado una función analítica. Lo normal en investigación científica es obtener medidas en función de uno o varios parámetros. Para obtener la integral en estos casos, se emplea la función

```
scipy.integrate.trapz(y, x=None, dx=1.0)
```

Parámetro	Tipo	Descripción
y	array	Conjunto de puntos a integrar.
x	array	(Opcional) Si es None es espacio entre puntos es dx.
dx	escalar	Si x=None, dx es el espacio entre puntos. Por defecto se asume 1.0.

Devuelve el valor de la integral obtenida por el método de la regla trapezoidal.

Integración analítica vs. numérica

`scipy.integrate.quad` vs. `scipy.integrate.trapz`

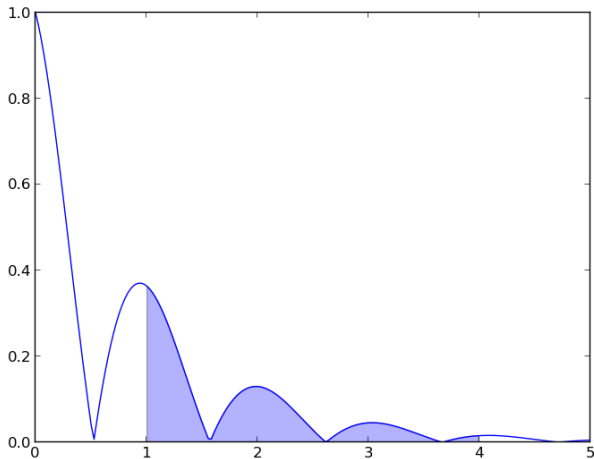
```
>>> import scipy
>>> from scipy.integrate import quad, trapz
>>> from matplotlib import pyplot
x = scipy.sort(scipy.randn(1000) * 5).clip(1, 4)
>>> def func2(x):
...     return scipy.absolute(scipy.exp(-x) * scipy.cos(x * 3))
>>> an_sol = quad(func2, 1, 4)[0]
>>> nu_sol = trapz(func2(x), x=x)
>>> print 'Soluciones (analitica, numerica) = (%7.6f, %7.6f)' % \
(an_sol, nu_sol)
>>> nx = scipy.linspace(0, 5, 200)
>>> pyplot.plot(nx, func2(nx))
>>> pyplot.fill_between(nx, 0, func2(nx), where=(nx>=1)&(nx<=4))
>>> pyplot.show()
```

Salida por pantalla:

Soluciones (analitica, numerica) = (0.231357, 0.231502)

Ejemplo5

Integración analítica vs. numérica (Continuación)



Integración doble

`scipy.integrate.dblquad`

La función

```
dblquad(func, a, b, gfun, hfun, args=())
```

sirve para integrar una función `func` de dos variables:

- `func`, es la función a integrar (dependiente de 2 variables)
- `a` y `b`, son los límites de integración en x ($a < b$)
- `gfun`, es una **función** de y , que depende sólo de una variable (x), que funciona como límite inferior de integración para y .
- `hfun`, es la **función** que determina el límite superior de integración para y (de nuevo, como función sólo de x).
- `args`, es una tupla con los parámetros adicionales de la función `func`.

Integración doble

`scipy.integrate.dblquad`

Volumen un cubo de lado parametrizable)

```
>>> import scipy
>>> import scipy.integrate
>>> def cubo(y, x, lado):
...     return lado
...
>>> for lado in range(1, 4):
...     res = scipy.integrate.dblquad(cubo, 0, lado, \
...     lambda x: 0, lambda x: lado, args=(lado,))
...     print 'Cubo de lado = %d --> Volumen = ' % lado, res
```

Salida:

```
Cubo de lado = 1 --> Volumen = (1.0, 1.1102230246251565e-14)
Cubo de lado = 2 --> Volumen = (8.0, 8.881784197001252e-14)
Cubo de lado = 3 --> Volumen = (27.0, 2.9976021664879227e-13)
```


Integración doble

`scipy.integrate.dblquad`

Reto

¿Se atreve a obtener el volumen de una esfera?

Integración triple

`scipy.integrate.tplquad`

La función

```
scipy.integrate.tplquad(func, a, b, gfun,  
                        hfun, qfun, rfun, args=())
```

sirve para realizar integrales triples en funciones `func` de tres variables (x, y, z) :

- `func`, es la función a integrar (dependiente de 3 variables).
- `a` y `b`, son los límites de integración en x e y ($a < b$).
- `gfun`, es una función de y , dependiente de x , que funciona como límite inferior de integración para y .
- `hfun`, límite superior de integración para y .
- `qfun`, superficie(x, y) que funciona como límite inferior para la variable z .
- `rfun`, mismas características que `qfun`, pero con funciones de límite superior para la variable z .

Retorna una tupla: (integral, estimación del error absoluto del resultado)

Integración triple

`scipy.integrate.tplquad`

$f(x, y, z) = y \cdot \sin(x) + z \cdot \cos(x)$ en la región $0 \leq x \leq \pi$,
 $0 \leq y \leq 1$, $-1 \leq z \leq 1$

```
from scipy.integrate import tplquad
import numpy as np

def integrand(z, y, x):
    return y * np.sin(x) + z * np.cos(x)

ans, err = tplquad(integrand, \
    ...              0, np.pi, # x limits
    ...              lambda x: 0, lambda x: 1, # y limits
    ...              lambda x,y: -1, lambda x,y: 1) # z limits
print ans
```

Salida

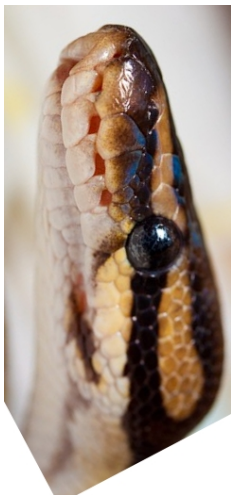
2.0

Ejercicios

Ejercicios 7, 8 y 9

<http://www.iaa.es/python/cientifico/ejercicios/scipy>

Índice



- 1 Introducción
- 2 Búsqueda de mínimos
- 3 Búsqueda de raíces
- 4 Ajuste de curvas
- 5 Interpolación
- 6 Integración
- 7 Estadística y Distribuciones**
- 8 Álgebra lineal
- 9 Procesamiento de señales
- 10 Agrupamiento
- 11 Magnitudes Físicas

Estadística descriptiva

Valores centrales y medidas de dispersión

SCIPY se basa en NUMPY adoptando las funciones de las que éste dispone para caracterizar medidas de valores centrales y dispersión de un array cualquiera.

Función	Descripción
<code>scipy.average(a, axis=None, weights=None)</code>	Devuelve un escalar o array con la media “pesada” del array <code>a</code> por los valores “weights” en el eje “axis” seleccionado.
<code>scipy.mean(a, axis=None)</code>	Devuelve un escalar o array con la media aritmética del array sobre el “axis” dado.
<code>scipy.median(a, axis=None)</code>	Devuelve un escalar o array con la mediana del array para el eje seleccionado.
<code>scipy.std(a, axis=None)</code>	Devuelve un escalar o array con la desviación estándar en el “axis”.
<code>scipy.var(a, axis=None)</code>	Devuelve un escalar o array con la varianza de los elementos en “axis” eje seleccionado.

Distribuciones de probabilidad

`scipy.stats`

SCIPY dispone de un número enorme de distribuciones de probabilidad, tanto discretas (12 distribuciones) como continuas (unas 84).

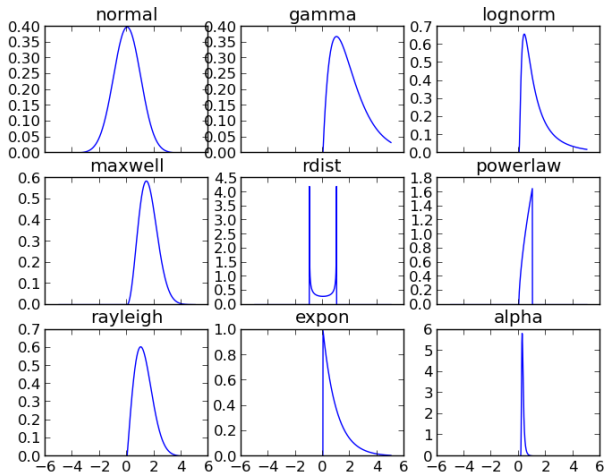
(Consultar documentación)

Entre las posibilidades más atractivas están

- 1 Capacidad de generar muestras aleatorias de cualquiera de esas casi 100 disponibles.
- 2 Dada una muestra aleatoria obtenida experimentalmente, determinar los parámetros de una determinada distribución que mejor ajustan.

Distribuciones de probabilidad

`scipy.stats`



Distribuciones de probabilidad

`scipy.stats`

De los métodos disponibles de las distribuciones, los que más nos interesan son:

Método	Descripción
<code>rvs(loc=0, scale=1, size=1)</code>	Genera una muestra aleatoria de esta distribución. El tamaño viene dado <code>size</code> .
<code>pdf(x, loc=0, scale=1)</code>	Función Densidad de Probabilidad.
<code>cdf(x, loc=0, scale=1)</code>	Función de distribución acumulativa.
<code>ppf(q, loc=0, scale=1)</code>	Percentil.
<code>stats(loc=0, scale=1, moments='mv')</code>	Devuelve la media, varianza, sesgo (de Fisher) o kurtosis (de Fisher), según el valor de <code>moments</code> : media('m'), varianza('v'), sesgo('s'), y kurtosis('k')
<code>fit(data, loc=0, scale=1)</code>	Estima parámetros de la distribución a partir de <code>data</code> .

Distribuciones de probabilidad

scipy.stats

```
>>> import scipy.stats
>>> print scipy.stats.norm.cdf(scipy.array([-2, 0, 2]))
[ 0.02275013  0.5          0.97724987]
>>> print scipy.stats.norm.stats(moments="mvsk")
(array(0.0), array(1.0), array(0.0), array(0.0))
>>> print scipy.stats.norm.ppf(0.5)  # mediana = percentil 50
0.0
>>> print scipy.stats.norm.rvs(size=3)  # muestra de 3 elementos
[ 1.37629104  0.24356188 -1.7217918 ]
>>> sample = scipy.stats.norm.rvs(loc=3, scale=2, size=3)
>>> print sample
[ 4.79340559  3.46683487  4.74428621]
>>> print sample.mean(), sample.std(), sample.var()
4.33484222524 0.614101380386 0.377120505392
>>> a = scipy.stats.norm.rvs(loc=4, scale=0.5, size=1000)
>>> print scipy.stats.norm.fit(a)
(4.0141137335638515, 0.49461520101014156)
```

Distribuciones de probabilidad

`scipy.stats`. Funciones Útiles

Obtener la media, varianza y desviación estándar con errores de estimación.

```
scipy.stats.bayes_mvs(data, alpha=0.9)
```

Parámetro	Tipo	Descripción
data	array	Muestra. Si es multidimensional, lo pasa a 1D. Necesita de 2 puntos como mínimo.
alpha	float	(Opcional) Probabilidad de que el intervalo de confianza contenga el valor verdadero.

Retorna una tupla: (mean_cntr, var_cntr, std_cntr)
donde cada elemento es una tupla con el siguiente formato:
(valor_dentral, (mínimo, máximo) de intervalo de
confianza)

Distribuciones de probabilidad

scipy.stats. Funciones Útiles

Se puede eliminar *outliers* de la muestra mediante *sigmaclipping*.

```
scipy.stats.sigmaclip(a, low=4.0, high=4.0)
```

El array de salida contiene los elementos que verifican

$$\text{mean}(a) - \text{std}(a) * \text{low} < a < \text{mean}(a) + \text{std}(a) * \text{high}$$

Parámetro	Tipo	Descripción
a	array	Muestra.
low	float	(Opcional) Factor de escala inferior para el <i>clipping</i> .
high	float	(Opcional) Factor de escala superior para el <i>clipping</i> .

Retorna una tupla:

(array filtrado, límite inferior de filtrado, límite superior de filtrado)

Distribuciones de probabilidad

`scipy.stats`. Funciones Útiles (continuación)

Se puede truncar la muestra a un límite inferior o superior (o ambos).

```
scipy.stats.threshold(a, threshmin=None,  
                      threshmax=None, newval=0)
```

Parámetro	Tipo	Descripción
<code>a</code>	array	Muestra.
<code>threshmin</code>	float, int, None	(Opcional) Valor mínimo de la muestra.
<code>threshmax</code>	float, int, None	(Opcional) Valor máximo de la muestra
<code>newval</code>	float, int	(Opcional) Reemplazo para valores fuera de los limites.

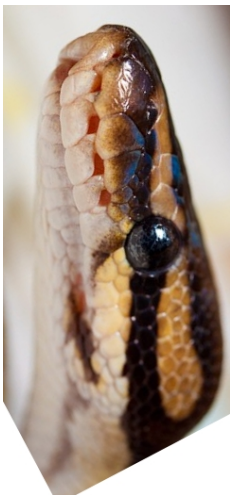
Retorna el array de la misma longitud que el de entrada.
Los valores fuera de los límites son sustituidos por `newval`.

Ejercicios

Ejercicios 10 y 11

<http://www.iaa.es/python/cientifico/ejercicios/scipy>

Índice



- 1 Introducción
- 2 Búsqueda de mínimos
- 3 Búsqueda de raíces
- 4 Ajuste de curvas
- 5 Interpolación
- 6 Integración
- 7 Estadística y Distribuciones
- 8 **Álgebra lineal**
- 9 Procesamiento de señales
- 10 Agrupamiento
- 11 Magnitudes Físicas

Álgebra lineal

`scipy.linalg`

- `scipy.linalg` contiene y amplía las funcionalidades de `numpy.linalg`.
- las rutinas están construidas sobre las librerías optimizadas ATLAS, LAPACK y BLAS, lo cual garantiza el resultado y un rendimiento en operaciones espectacular.

Producto escalar de matrices

scipy.dot

```
scipy.dot(array1, array2)
```

Las dimensiones han de ser compatibles.

```
>>> from scipy import linalg
>>> A = scipy.arange(1, 7).reshape((2,3))
>>> A.shape
(2, 3)
>>> b = scipy.ones(3)
>>> b.shape
(3,)
>>> b = scipy.array([b])
>>> b.shape
(1, 3)
>>> prod = scipy.dot(A, b.transpose())
>>> print prod.T
[[ 6. 15.]]
>>> print prod.shape
(2, 1)
```

Inversa de una matriz

`scipy.linalg.inv`

`scipy.linalg.inv(ndarray)`

```
>>> from scipy import linalg
>>> A = scipy.arange(1, 5).reshape((2,2))
>>> A
array([[1, 2],
       [3, 4]])
>>> B = linalg.inv(A)
>>> B
array([[ -2. ,  1. ],
       [ 1.5, -0.5]])
>>> A.dot(B) # verificacion
array([[ 1.,  0.],
       [ 0.,  1.]])
```

Determinante de una matriz

`scipy.linalg.det`

`scipy.linalg.det(ndarray)`

```
>>> from scipy import linalg
>>> A = scipy.arange(1, 5).reshape((2,2))
>>> A
array([[1, 2],
       [3, 4]])
>>> linalg.det(A)
-2.0
```

Resolución de sistemas lineales

`scipy.linalg.solve`

`scipy.linalg.solve(a, b)`

Parámetro	Tipo	Descripción
a	array	Dimensiones (M, M) .
b	array	Dimensiones $(M,)$ o (M, N) Corresponde a la matriz de la ecuación matricial $a \cdot x = b$.

Devuelve un array de dimensiones $(M,)$ o (M, N) que resuelve el sistema matricial

$$a \cdot x = b$$

Tiene, por tanto, la misma forma (*shape*) que b.

Resolución de sistemas lineales

Dado el sistema lineal

$$\begin{aligned}3x + 2y + z &= 1 \\5x + 3y + 4z &= 2 \\x + y - z &= 1\end{aligned}$$

Se toma por matriz A los coeficientes de x , y y z , y por vector b el de los términos independientes.

```
>>> import scipy
>>> import scipy.linalg
>>> A = scipy.array([[3, 2, 1], [5, 3, 4], [1, 1, -1]])
>>> b = scipy.array([1, 2, 1])
>>> solucion = scipy.linalg.solve(A, scipy.transpose(b))
>>> print solucion.T
[-4.  6.  1.]
```

Autovalores y autovectores

`scipy.linalg.`

La función

```
scipy.linalg.eig(A)
```

realiza la descomposición del array (N x N) dimensional A.
Devuelve una tupla con dos elementos:

```
(valores_propios, vectores_propios)
```

Autovalores y autovectores

`scipy.linalg.`

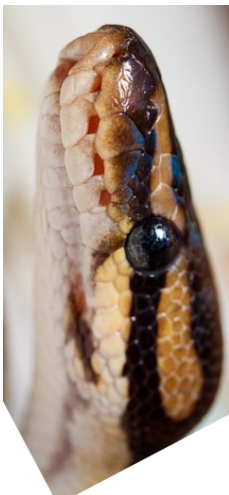
```
>>> import scipy.linalg
>>> A = scipy.array([[2, 4], [1, 3]])
>>> eigval, eigvect = scipy.linalg.eig(A)
>>> eigval[0], eigval[1]
((0.43844718719116971+0j), (4.5615528128088307+0j))
>>> eigvect[:,0] # eigenvectores en columnas
array([-0.93153209,  0.36365914])
>>> eigvect[:,1]
array([-0.84212294, -0.5392856 ])
>>> mod_eigvect1 = scipy.dot(eigvect[:,0], eigvect[:,0].T)
>>> mod_eigvect1 # modulo del vector propio 1
1.0
>>> mod_eigvect2 = scipy.dot(eigvect[:,1], eigvect[:,1].T)
>>> mod_eigvect2 # modulo del vector propio 2
0.99999999999999978
```

Ejercicios

Ejercicio 12

<http://www.iaa.es/python/cientifico/ejercicios/scipy>

Índice

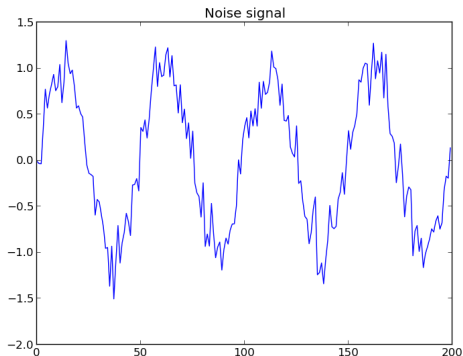


- 1 Introducción
- 2 Búsqueda de mínimos
- 3 Búsqueda de raíces
- 4 Ajuste de curvas
- 5 Interpolación
- 6 Integración
- 7 Estadística y Distribuciones
- 8 Álgebra lineal
- 9 Procesamiento de señales**
- 10 Agrupamiento
- 11 Magnitudes Físicas

Suavizado de señal ruidosa

scipy.signal

Un caso muy común en ciencia



Suavizado de señal ruidosa

scipy.signal

Para suavizar, podemos emplear

- 1 filtros.
- 2 ventanas.

Dependiendo de si se trabaja sobre imágenes o señales unidimensionales, existen multitud de filtros y ventanas predefinidos en los paquetes `scipy.ndimage` y `scipy.signal`.

Filtros

scipy.signal.wiener

```
scipy.signal.wiener(im, mysize=None,  
                    noise=None)
```

Aplica el filtro de wiener a una señal N-dimensional.

Parámetro	Tipo	Descripción
im	array	Array N-dimensional.
mysize	entero o array	(Opcional) Escalar o array N-dimensional que indica el tamaño de la ventana del filtro en cada dimensión. El número de elementos de mysize debe ser impar.
noise	float	(Opcional) La potencia del ruido a emplear (<i>nois-power</i>). Si no se da (None), se estima como el promedio de la varianza local de la señal de entrada (im).

Retorna un array de la misma que el de entrada (im) sobre el que se ha realizado el filtrado.

Filtros

scipy.signal.medfilt

```
scipy.signal.medfilt(volume,  
                      kernel_size=None)
```

Aplica el filtro de mediana a una señal N-dimensional (dada por un array).

Parámetro	Tipo	Descripción
volume	array	Array N-dimensional.
kernel_size	entero o array	(Opcional) Escalar o array N-dimensional que indica el tamaño de la ventana del filtro en cada dimensión. El número de elementos de kernel_size debe ser impar. Por defecto vale 3.

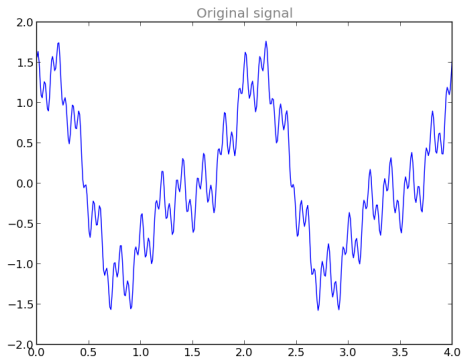
Retorna un array de la misma que el de entrada (`im`) sobre el que se ha realizado el filtrado.

Filtros

scipy.signal

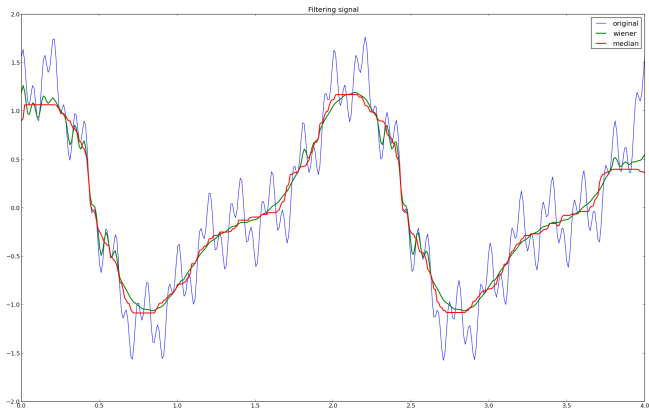
Comparamos el trabajo de ambos filtros sobre una señal que es la superposición de varios armónicos

$$f(t) = \cos(\pi \cdot t) + 0,5 \cdot \sin(2 \cdot \pi \cdot t + 0,3) + 0,3 \cdot \cos(10 \cdot \pi \cdot t + 0,2) + 0,2 \cdot \sin(30 \cdot \pi \cdot t + 0,7)$$



Filtros

`scipy.signal`. Comparando filtros



Filtros

scipy.signal. Comparando filtros:código

```
>>> import scipy
>>> import scipy.signal
>>> def signal(t):
...     return scipy.cos(scipy.pi * t) + \
...     0.5 * scipy.sin(2 * scipy.pi * t + 0.3) + \
...     0.3 * scipy.cos(10 * scipy.pi * t + 0.2) + \
...     0.2 * scipy.sin(30 * scipy.pi * t + 0.7)
...
>>> t = scipy.linspace(0, 4, 400)
>>> pyplot.plot(t, signal(t))
>>> pyplot.title('Filtering signal')
>>> pyplot.plot(t, scipy.signal.wiener(signal(t), mysize=41), lw=2)
>>> pyplot.plot(t, scipy.signal.medfilt(signal(t), \
... kernel_size=41), lw=2)
>>> pyplot.legend(['original', 'wiener', 'median'])
>>> pyplot.show()
```


Ventanas

scipy.signal

Otra forma de suavizado de datos consiste en la *convolución* de una señal de entrada con una ventana. Una ventana es una función matemática que tiene valores nulos fuera de un dominio dado.

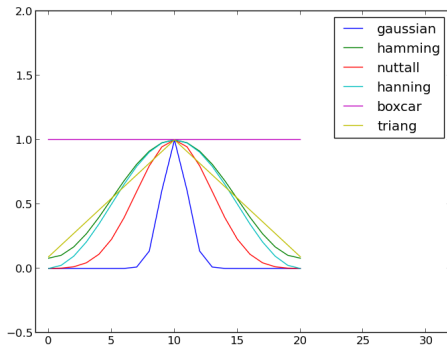
En el proceso de convolución

- 1 se relaciona el valor de cada medida con las que tiene a su alrededor.
- 2 Se tienen en cuenta más vecinos cuando mayor sea la anchura de la ventana.
- 3 La "*forma*" de dicha ventana influye críticamente en el resultado final.

Ventanas

`scipy.signal`. Ventanas de convolución para suavizado.

SCIPY en su paquete `signal` (`scipy.signal`) permite usar ventanas estándar en el procesamiento de señal.



Ventanas

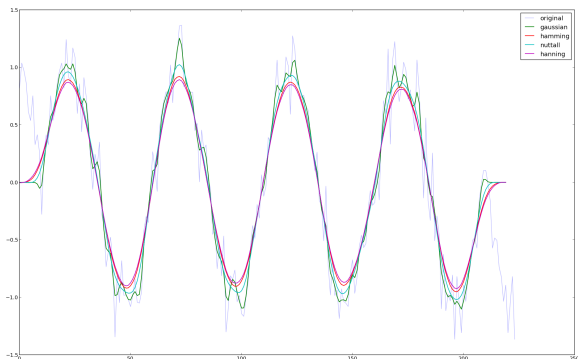
scipy.signal. Ventanas de convolución para suavizado: código

```
>>> import scipy.signal
>>> windows = ['gaussian', 'hamming', 'nuttall', 'hanning', 'boxcar',
>>> # windows width = 10
>>> for w in windows:
>>>     if w == 'gaussian':
...         eval('pyplot.plot(scipy.signal.%s(21,1))' % w)
...     else:
...         eval('pyplot.plot(scipy.signal.%s(21))' % w)
...     pyplot.ylim([-0.5, 2])
...     pyplot.xlim([-1, 32])
...     pyplot.legend(windows)
>>> pyplot.show()
```

Ventanas

scipy.signal

El resultado de aplicar la operación de convolución con algunas de estas ventanas sobre la señal ruidosa es



Ventanas

scipy.signal. Ejemplo de código de suavizado

```
>>> import scipy.signal
>>> x = scipy.linspace(-4 * scipy.pi, 4 * scipy.pi, 200)
>>> signal = scipy.sin(x)
>>> noise = scipy.random.randn(signal.size) * 0.2
>>> noise_signal = signal + noise # generacion de sennal y ruido
>>> # sennal extendida para su correcto muestreo
>>> e_noise_signal = scipy.concatenate([noise_signal[10:0:-1], \
... noise_signal, noise_signal[-1:-15:-1]])
>>> # Smoothing
>>> pyplot.plot(e_noise_signal, alpha=0.3)
>>> windows2 = ['gaussian', 'hamming', 'nuttall', 'triang']
>>> for w in windows2:
...     if w == 'gaussian':
...         window = eval('scipy.signal.%s(21,1)' % w)
...     else:
...         window = eval('scipy.signal.%s(21)' % w)
>>> output = scipy.convolve(window / window.sum(), noise_signal)
>>> pyplot.plot(output, lw=1.5)
>>> pyplot.legend(['original'] + windows)
>>> pyplot.show()
```

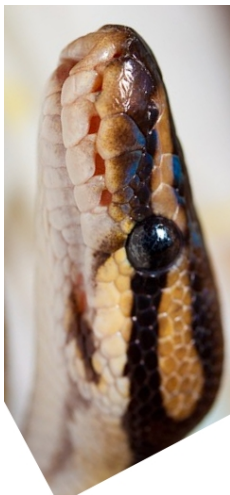


Ejercicios

Ejercicios 13 y 14

<http://www.iaa.es/python/cientifico/ejercicios/scipy>

Índice

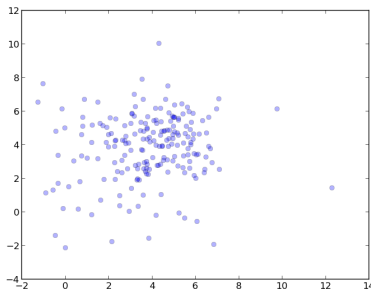


- 1 Introducción
- 2 Búsqueda de mínimos
- 3 Búsqueda de raíces
- 4 Ajuste de curvas
- 5 Interpolación
- 6 Integración
- 7 Estadística y Distribuciones
- 8 Álgebra lineal
- 9 Procesamiento de señales
- 10 Agrupamiento**
- 11 Magnitudes Físicas

Agrupamiento

Caso práctico

Tengo el siguiente conjunto conjunto de datos



¿Cómo agrupo las observaciones en 2 o más grupos?

Agrupamiento

`scipy.clustering`

Dos paquetes realizan este tipo de tareas:

- 1 `scipy.clustering.vq`
Soporta técnicas de *vector quantization* y *algoritmo k-means*.
- 2 `scipy.clustering.hierarchy`
Agrupamiento por aglomeración y jerárquico.

Agrupamiento k-means

`scipy.clustering.vq`

En el algoritmo *k-means* se necesitan dos parámetros de entrada:

- 1 el conjunto de **datos** a agrupar.
- 2 el **número de grupos** a generar.

Los datos son normalmente arrays-2D de dimensión $M \times N$, donde M es el número de objetos y N es el número de datos del que se dispone para cada objetos

Dado el número de grupos, el algoritmo trabaja asignando unas coordenadas al centro de cada uno de los grupos y asignando el identificador a cada objeto en función del grupo cuyo centro esté más próximo. Esto es lo que se denomina *cuantificar vectores* (*vector quantization*)

k-means

`scipy.cluster.vq.kmeans`

```
scipy.cluster.vq.kmeans(obs, k_or_guess, iter=20,
                        thresh=1e-05)
```

Parámetro	Tipo	Descripción
obs	array	de dimensión $M \times N$ (M observaciones con N datos cada una).
k_or_guess	entero o array	Si es entero, es el número de grupos. Los centros se inicializan aleatoriamente entre las observaciones. Si es un array, se inicializan los centros a esas k posiciones ($k \times N$).
iter	entero	Número de iteracciones
thresh	entero	Termina las iteraciones si la distorsión (suma de las distancias al cuadrado entre cada observación y su <i>centroide dominante</i>) cambia menos que este valor.

Retorna

Parámetro	Tipo	Descripción
codebook	array	Array $k \times N$, con las coordenadas de los centroides que minimizan la distorsión
distorsion	float	Suma de las distancias al cuadrado entre cada observación y su <i>centroide dominante</i>

Vector quantization

`scipy.cluster.vq.vq`

`scipy.cluster.vq.vq(obs, code_book)`

Parámetro	Tipo	Descripción
<code>obs</code>	array	de dimensión $M \times N$ (M observaciones con N datos cada una).
<code>code_book</code>	array	Array $k \times N$, con las coordenadas de los centroides que minimizan la distorsión

Retorna

Parámetro	Tipo	Descripción
<code>code</code>	array	Array de longitud N , con el código del centroide dominante para cada observación
<code>dist</code>	array	Array de longitud N , con la distancia de cada observación al centroide dominante.

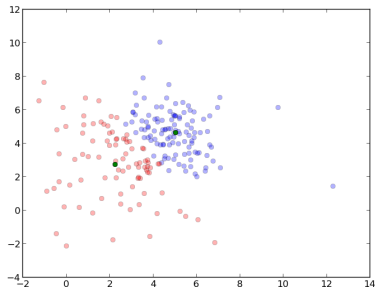
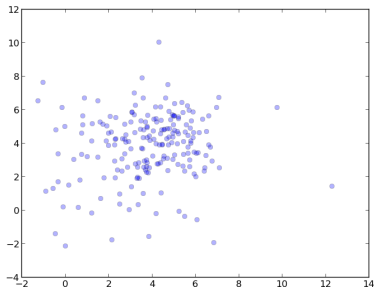
Ejemplo 1

Agrupamiento por k-means

```
>>> # genero la muestra
>>> x = scipy.concatenate((random.normal(2, 3, 100), \
random.normal(5, 1, 100)))
>>> y = scipy.concatenate((random.normal(4, 1, 100), \
random.normal(3, 2, 100)))
>>> data = scipy.concatenate((x, y))
>>> data = data.reshape((200,2))
>>> Agrupo
>>> codebook, distorsion = cluster.vq.kmeans(data, 2)
>>> code, dist = cluster.vq.vq(data, codebook)
>>> # puntos de un grupo
>>> pyplot.plot(data[:,0].compress(code==0), \
data[:,1].compress(code==0), 'bo', alpha=0.3)
>>> # puntos del otro grupo
>>> pyplot.plot(data[:,0].compress(code==1), \
data[:,1].compress(code==1), 'ro', alpha=0.3)
>>> # centroides
>>> pyplot.plot(codebook[:,0], codebook[:,1], 'go')
```

Ejemplo 1

Agrupamiento por k-means

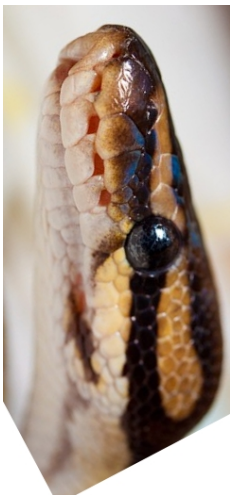


Ejercicios

Ejercicio 15

<http://www.iaa.es/python/cientifico/ejercicios/scipy>

Índice



- 1 Introducción
- 2 Búsqueda de mínimos
- 3 Búsqueda de raíces
- 4 Ajuste de curvas
- 5 Interpolación
- 6 Integración
- 7 Estadística y Distribuciones
- 8 Álgebra lineal
- 9 Procesamiento de señales
- 10 Agrupamiento
- 11 **Magnitudes Físicas**

Constantes *SCIPY*

`scipy.constants`

Este módulo contiene sobre constantes matemáticas y físicas.

Incluye un diccionario

“`scipy.constants.physical_constants`” en el que

- cada clave es el nombre de una constante física, y
- el valor es una tupla que incluye (valor de la constante, unidades, precisión).

Se dispone de 399 constantes físicas.

Constantes *SCIPY*

`scipy.constants`

El siguiente fragmento de código muestra diez de ellas:

```
>>> import scipy.constants
>>> keys = scipy.constants.physical_constants.keys()
>>> print len(keys)
399
>>> for j in range(10):
...     print keys[j]
...
joule-electron volt relationship
conductance quantum
standard acceleration of gravity
electron volt-hertz relationship
shielded helion to proton mag. mom. ratio
lattice parameter of silicon
neutron mag. mom. to nuclear magneton ratio
deuteron mass in u
kelvin-joule relationship
shielded helion magn. moment
```

Constantes *SCIPY*

`scipy.constants`

Cada constante lleva asociados tres valores, almacenados en forma de tupla en el diccionario `scipy.constants.physical_constants`

```
>>> scipy.constants.physical_constants['joule-electron_volt_relations']  
(6.24150965e+18, 'eV', 1600000000000.0)  
# (VALOR, UNIDADES, PRECISION)
```

Constantes *SCIPY*

`scipy.constants`

Existen 3 funciones que permiten obtener esos valores por separado para una constante dada:

- `scipy.constants.value(nombre_constante),`
- `scipy.constants.unit(nombre_constante),`
- `scipy.constants.precision(nombre_constante).`

```
>>> scipy.constants.value('electron_mass')
9.10938291e-31
>>> scipy.constants.unit('electron_mass')
'kg'
>>> scipy.constants.precision('electron_mass')
4.3910768045648e-08
```

Constantes *SCIPY*

`scipy.constants`

¿Tengo que leer todas las claves para saber si una constante existe y cuál es su nombre?

No es necesario. Podemos hacer una búsqueda aproximada mediante el uso de la función

```
scipy.constants.find(regex)
```

```
>>> scipy.constants.find('light')  
['speed_of_light_in_vacuum']
```

Prefijos de unidades

`scipy.constants`

Disponemos de órdenes para prefijos de unidades físicas.

```
>>> scipy.constants.yotta
1e+24
>>> scipy.constants.exa
1e+18
>>> scipy.constants.mega
1000000.0
>>> scipy.constants.kilo
1000.0
>>> scipy.constants.hecto
100.0
>>> scipy.constants.micro
1e-06
>>> scipy.constants.femto
1e-15
```

Unidades de información binaria

`scipy.constants`

También para unidades de información en sistema binario.

```
>>> scipy.constants.kibi
1024
>>> scipy.constants.gibi
1073741824
>>> scipy.constants.tebi
1099511627776
```

Funciones de conversión

`scipy.constants`

Existen funciones de conversión específicas para determinadas magnitudes.

```
>>> scipy.constants.lambda2nu(3000)
99930.819333333333
>>> # Equivalente a ...
>>> scipy.constants.find('light')
['speed_of_light_in_vacuum']
>>> scipy.constants.value('speed_of_light_in_vacuum')/3000
99930.81933333333
```


Ejercicios

Ejercicio 16

<http://www.iaa.es/python/cientifico/ejercicios/scipy>

FIN