originlab-python

April 30, 2014

Francisco José Navarro-Brull
TÉCNICAS DE CÁLCULO NUMÉRICO APLICADAS A LA FÍSICA Y A LA QUÍMICA - Máster en Ciencia de Materiales de la Universidad de Alicante

1 OriginLab vs Python (comparación de gráficas)

El software OriginLab® es uno de lo más utilizados en el mundo académico gracias a su versatilidad y potencia de cálculo. En concreto, OriginLab® tiene una serie de ventajas frente a sus *competidores* (Excel®, MATLAB®...) ya que logra unir ambos mundos permitiendo a alguien acostumbrado a la interfaz del primero realizar tareas de cálculo numérico más comunes del segundo. Pero si hay algo que verdaderamente es útil para un investigador que utiliza OriginLab®, son sus gráficas "listas para publicar".

En este contexto, Python cuenta con magníficas librerías capaces de llevar a cabo tareas de cálculo numérico (NumPy, SciPy) así para obtener gráficas (matplotlib) de calidad equivalente o superior a OriginLab.

Si has leído con detenimiento hasta aquí y te suena el software citado hasta ahora, la pregunta que te vendrá a la mente es ¿cómo un lenguaje de programación como Python va a sustituir a OrginLab® y su interfaz tipo Excel®?

Veamos, Python junto a sus librerías permite a día de hoy resolver el ciclo de trabajo propio tal y como permite OriginLab®. Éstos son: 1. Importar datos (xlrd, NumPy, csv, pandas) 2. Procesado de datos y cálculo (SciPy, NumPy) 3. Visualización (matplotib) 4. Iteración pasos 2 y 3 (IPython Notebook, Spyder) 5. Publicación de resultados (matplotib)

Entonces, **¿por qué no todo el mundo está usando Python? ¿cuál es el problema?**

Bien, en todo este planteamiento Python *falla* en un concepto. OriginLab® no requiere conocimientos de programación para su uso más básico, Python sí.

Por norma general:

- Muchos científicos no tienen conocimientos de programación (o éstos son muy reducidos)
- No tienen tiempo y quieren realizar estas tareas de la forma más rápida posible
- Representar gráficas por comandos de texto puede ser frustrante

Un fanático de Python (u otro lenguaje) te intentará convencer hablándote de las ventajas de aprender a programar, que ciertamente son muchas, pero esto te llevará tiempo (que *no* tienes) y más problemas (cuando tú lo que buscabas era una solución).

1.1 "Use the right tool"

Si te sientes cómodo utilizando OriginLab® y no le encuentras limitaciones, sigue utilizándolo y amortiza los 850 o 1800 dólares que vale su licencia para uso académico o profesional.

Si por el contrario, quieres (y dispones de tiempo para ello): * Automatizar el importado y procesado de datos * Tener a tu alcance algoritmos avanzados (y gratuitos) de estadística, machine learning, inteligencia artificial, computer vision... * Ahorrar en licencias y poder usar este software en la empresa a la que vayas * Aprender a programar en uno de los lenguajes más versátiles que existen * Hacer tu investigación

reproducible añadiendo a tu paper no sólo la información, si no además las herramientas poder reutilizar tu trabajo y ganar más impacto * Obtener gráficas para tus artículos de calidad igual o superior a Origin

¡Eres bienvenido al mundo de Python!

1.2 ¿De qué va este Notebook?

Como prueba de concepto, se utilizará IPython Notebook para llevar a cabo un proceso de reproducción de muchas de las gráficas que OriginLab® publicita. Puedes dar un vistazo a la Galería de OriginLab® y Galería de matplotib para ver lo similares que pueden llegar a ser.

IPython Notebook un formato y solución elegante que ademas de código puede contener texto, fórmulas mediante LaTeX, vídeos, imágenes y gráficas.

Nota: La mejor manera de trabajar con matplotlib es buscar en su galería una gráfica similar al resultado que queramos conseguir y utilizar dicho código como guía o plantilla de trabajo

1.3 ¿De qué *NO* va este Notebook?

Pese a que se comentarán brevemente ciertas líneas de código, este notebook no pretende ser un tutorial de matplotlib y/o Python. Para una introducción a los mismos te recomendamos este Curso online (gratuito) de introducción a Python para científicos e ingenieros por [@Pybonacci](http://pybonacci.wordpress.com) y organizado por CAChemE.org.

Por cierto, también puedes echarle un vistazo a Avoplot, un proyecto muy interesante que pretende simiplificar la vida a muchos científicos pero que de momento se encuentra en fase de desarrollo.

1.4 Librerías gráficas en Python:

Los siguientes ejemplos harán uso de matplotlib y ajustarán el estilo de las gráficas para hacerlas similares las de OriginLab®. Matplotlib puede parecer viejo, estático y tener una configuración por defecto cutre. Sin embargo, su uso es muy, muy extenso y ha demostrado ser una librería a prueba de todo. Tal y como mencionaba [@Pybonacci](http://twitter.com/ptbonacci), es válido para el 99% de las personas. El 1% restante tiene varias alternativas que elegir. Muchas de ellas se basan matplotlib, lo que demuestra su robustez, como seaborn, vincent, ggplot-py, prettyplot, plot.ly, bearcart o una de las más recientes e interesantes mpld3. Por otro lado existen alternativas que han empezado un motor gráfico de Python desde cero cómo por ejemplo Bokeh.

En cualquier caso matplotlib otorga las siguientes funcionalidades.

- API: Tipo MATLAB (por estados, tersa, menos poderosa) u orientada a objetos (sin estados, verbosa, más poderosa)
- Abstracción: Básicamente un potente modelo interno de objetos SVG (vectoriales)
- Gráficos de salida:
  - Gráficos estáticos finales (backends) de los cuales puedas necesitar: pdf, png, svg, eps, ps, pgf, jpeg...
  - Así como interfaces gráficas (GUI backends): Tk, Agg, OSX, GTK, Qt4, WebAgg...

Para aprender más sobre matplotib es recomendable leer Python in the browser age por Jake VanderPlas (fuente original de lo descrito en este último párrafo).

2 Comparación de capacidades gráficas (Origin vs Python)

En primer lugar, queremos que las gráficas aparezcan en este mismo notebook por lo que damos la siguiente instrucción:

```
In [1]: %matplotlib inline
```

Acto seguido importamos las librerías necesarias para la generación y representación los resultados:

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
```

En caso de trabajar con Python 2.7:

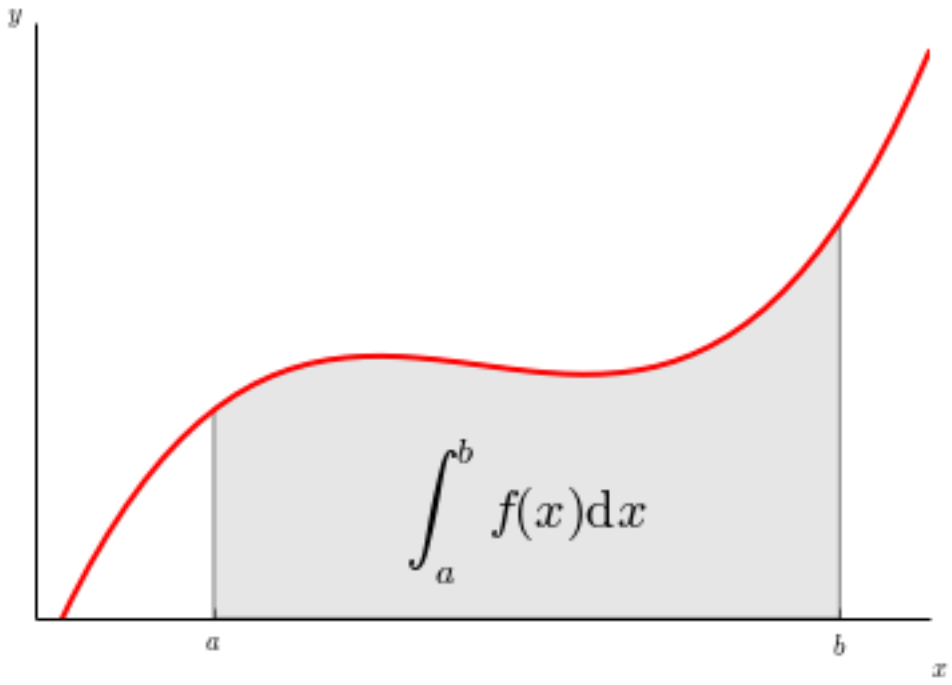
```
In [3]: from __future__ import division
```

2.0.1 Potencia de matplotlib: Resultados combinados con LaTeX

Antes de empezar veamos un ejemplo de la capacidad de matplotlib para generar figuras "listas para publicar" y su magnífica inegración con LaTeX. Además de representar valores numéricos, matplotlib permite la añadir notas y texto en este formato. Un ejemplo de esta funcionalidad es el siguiente:

```
In [4]: # Los comentarios en Python se escriben empezando con una almohadilla
        # estas líneas serán ignoradas en la ejecución


        # Creamos la función que queremos representar (e integrar)
        def func(x):
            return (x - 3) * (x - 5) * (x - 7) + 85

        # Especificamos los límtes de integración
        a, b = 2, 9

        # Crea un vector con valores de 0 a 10
        x = np.linspace(0, 10)

        # Obtenemos los valores de y correspondientes
        y = func(x)

        # Utilizamos matplotlib (este formato es orientado a clases)
        # matplotlib también puede usarse tipo MATLAB(R)
        fig, ax = plt.subplots()

        # Dibuja con una línea roja de espesor determinado los resultados
        plt.plot(x, y, 'r', linewidth=2)

        # Establece el límite inferior del eje y
        plt.ylim(ymin=0)

        # Crea la región sombreada que representará la integral
        ix = np.linspace(a, b)
        iy = func(ix)

        # Coordenadas de los puntos del polígono de la integral a representar
        verts = [(a, 0)] + list(zip(ix, iy)) + [(b, 0)]

        # Carga la función Polygon de la librería
        from matplotlib.patches import Polygon

        # Representa dicho polígono cierto 90 y 50% de transparencia para el relleno y borde.
        poly = Polygon(verts, facecolor='0.9', edgecolor='0.5')
        ax.add_patch(poly)

        # Añade texto en latex con la siguiente instruccion
```

```
# plt.text(coordenada_x, coordenada_y, texto, opciones)
plt.text(0.5 * (a + b), 30, r"$\int_a^b f(x)\mathrm{d}x$",
         horizontalalignment='center', fontsize=20)

# Añade texto con posición relativa para indicar los ejes
plt.figtext(0.9, 0.05, '$x$')
plt.figtext(0.1, 0.9, '$y$')

# Ocutlamos los bordes superior y derecho del cuadro de la gráfica
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)

# Añade marcas en el eje x con los límites de integración
# Coordenadas
ax.set_xticks((a, b))
# Texto
ax.set_xticklabels(('$a$', '$b$'))

# Establece marcas sólo en el eje x inferior
ax.xaxis.set_ticks_position('bottom')

# Elimina marcas del eje y
ax.set_yticks([])

# Muestra figura
plt.show()

# Guarda figura en png o pdf
# Más formatosy y/o opciones mirar en la documentación
# http://matplotlib.org/faq/howto_faq.html#save-multiple-plots-to-one-pdf-file

# Dado que llamamos a la figura "fig" cuando la creamos,
# llamamos al método .savefig

#fig.savefig(r'figuras/mi_figura.pdf')
#fig.savefig(r'figuras/mi_figura.png')
```

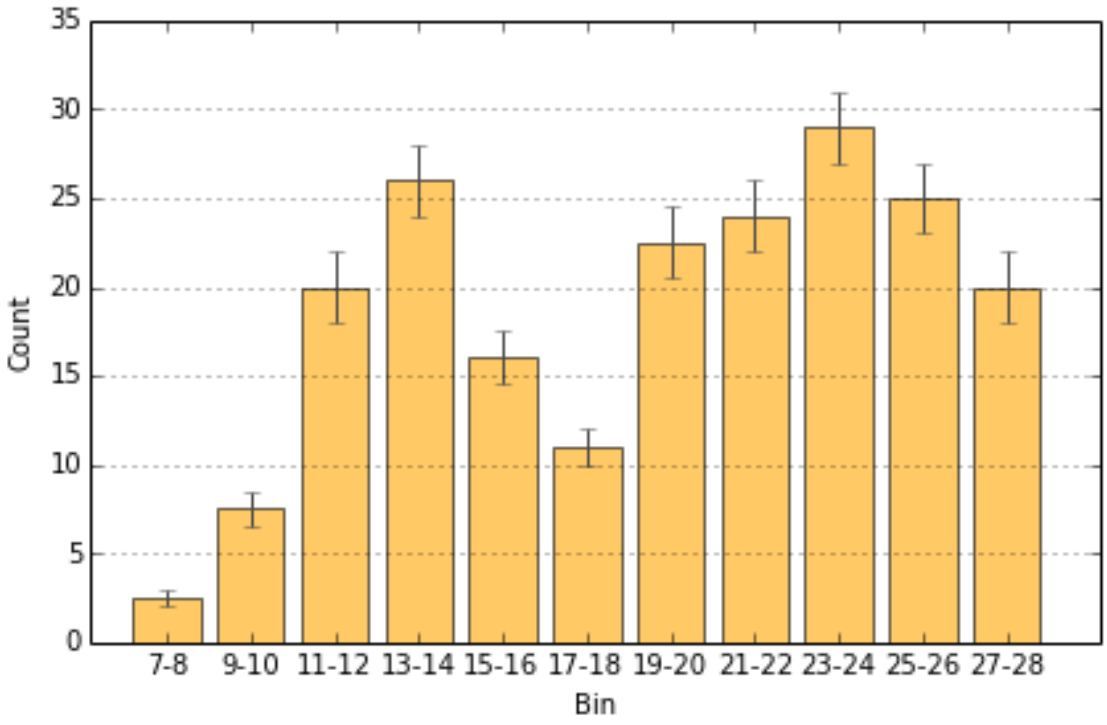


2.0.2 Representación de errores

A pesar de que Excel® permite representar errores en los gráficos, esta es una funcionalidad importante. Veamos como podemos hacer esto con matplotib.

```
In [5]: n_grupos = 11

        valores = (2.5, 7.5, 20, 26, 16, 11, 22.5, 24, 29, 25, 20)
        errores = (0.5, 1, 2, 2, 1.5, 1, 2, 2, 2, 2, 2)

        fig, ax = plt.subplots()

        ax.yaxis.grid()

        index = np.arange(n_grupos)
        ancho_banda = 0.8

        opacity = 0.6
        error_config = {'ecolor': '0.3'}

        distancia_inicio = 0.5

        coordenadas_x_barras = index+distancia_inicio

        rects1 = plt.bar(coordenadas_x_barras, valores, ancho_banda,
                         alpha=opacity,
                         color='orange',
                         yerr=errores,
```

```
                    error_kw=error_config,
                    )

plt.xlabel('Bin')
plt.ylabel('Count')

etiquetas_eje_x = ('7-8', '9-10', '11-12', '13-14', '15-16',
                   '17-18', '19-20', '21-22', '23-24', '25-26', '27-28')

coordenadas_etiquetas_x = index + ancho_banda/2 + distancia_inicio

plt.xticks(coordenadas_etiquetas_x, etiquetas_eje_x)
plt.tight_layout()

# Opcional si queremos salvar las figuras como archivos
#fig.savefig(r'figuras/barras_errores.pdf')
#fig.savefig(r'figuras/barras_errores.png',dpi=150)

plt.show()
```



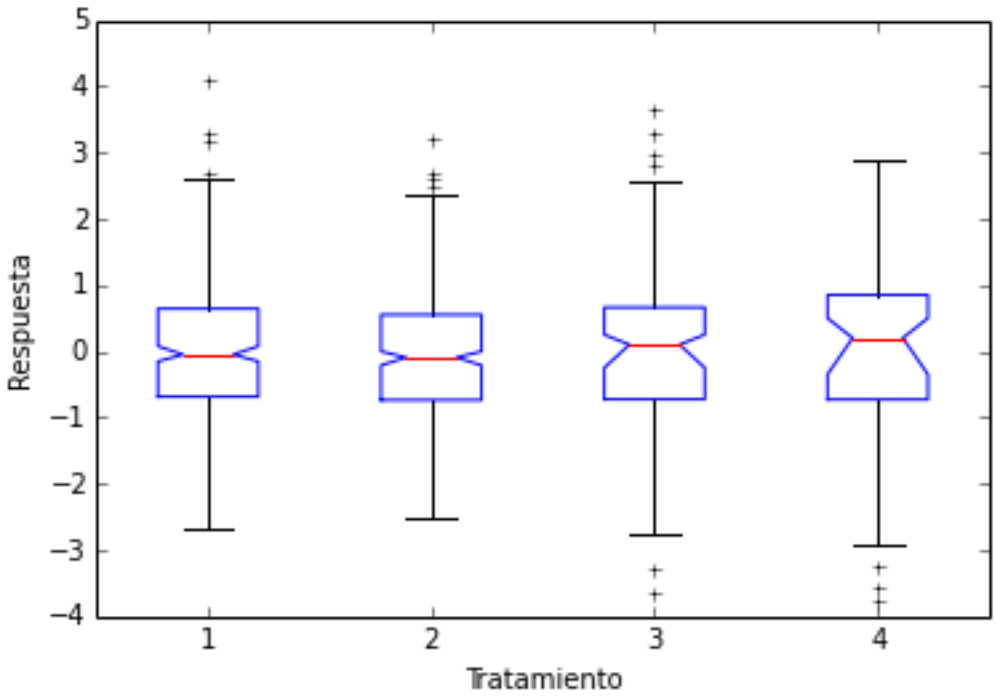
Resultado reproducido con Origin (ver online).

2.0.3 Diagramas de cajas (Whiskey)

Veamos algo más interesante, realizar diagramas de cajas y bigotes con Excel® ya no es tan sencillo. Sin embargo en matplotlib:

```
In [6]: def fakeBootStrapper(n):
            '''
```

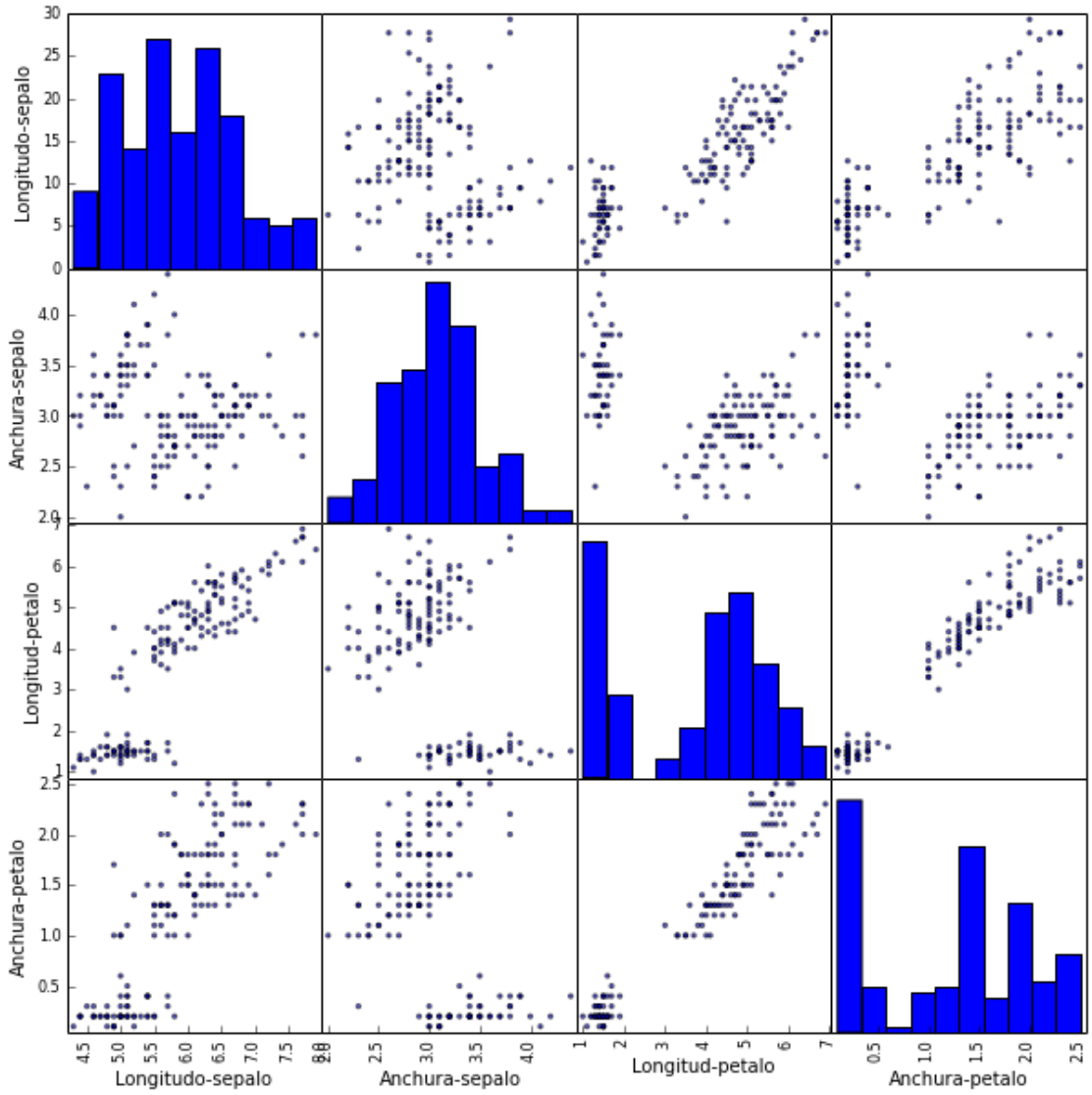
```
    Devuelve una mediana arbitraria e intervalos de confianza
    en una tupla
    '''
    if n == 1:
        med = 0.1
        CI = (-0.25, 0.25)
    else:
        med = 0.2
        CI = (-0.35, 0.50)

    return med, CI


# Fija una semilla para el random
np.random.seed(2)
inc = 0.1
e1 = np.random.normal(0, 1, size=(500,))
e2 = np.random.normal(0, 1, size=(500,))
e3 = np.random.normal(0, 1 + inc, size=(500,))
e4 = np.random.normal(0, 1 + 2*inc, size=(500,))

tratamientos = [e1,e2,e3,e4]
med1, CI1 = fakeBootStrapper(1)
med2, CI2 = fakeBootStrapper(2)
medianas = [None, None, med1, med2]
conf_intervalos = [None, None, CI1, CI2]

fig, ax = plt.subplots()

# Posiciones de las cajas
pos = np.array(range(len(tratamientos)))+1

# Representa el diagrama
bp = ax.boxplot(tratamientos, sym='k+', positions=pos,
                notch=1, bootstrap=5000,
                usermedians=medianas,
                conf_intervals=conf_intervalos)

# Etiquetas para los ejes
ax.set_xlabel('Tratamiento')
ax.set_ylabel('Respuesta')

# Tipo de diagrama de cajas (Whisker)
plt.setp(bp['whiskers'], color='k',  linestyle='-' )
plt.setp(bp['fliers'], markersize=5.0)

# Opcional si queremos salvar las figuras como archivos
#fig.savefig(r'figuras/diagra-cajas.pdf')
#fig.savefig(r'figuras/diagra-cajas.png',dpi=150)

plt.show()
```



Resultado reproducido con Origin (ver online).

2.0.4 Matriz de dispersión (Scatter Matrix)

In [30]:
```
# La forma más sencilla de realizar esta visualización es
# mediante la librería pandas que permite trabajar de forma similar a R

import pandas as pd

# Lee los datos (deben de estar en la carpeta de este Notebook)
# http://en.wikipedia.org/wiki/Iris_flower_data_set

iris = pd.read_csv("data/iris.csv")

# Establece el dataframe
df = pd.DataFrame(iris, columns=['Longitudo-sepalo', 'Anchura-sepalo',
                                 'Longitud-petalo', 'Anchura-petalo'])

# Calcula y representa resultados
pd.scatter_matrix(df, alpha=0.6, figsize=(10,10))

# Opcional si queremos salvar las figuras como archivos
#plt.savefig(r'figuras/matriz-dispersion.pdf')
#plt.savefig(r'figuras/matriz-dispersion.png',dpi=150)

plt.show()
```

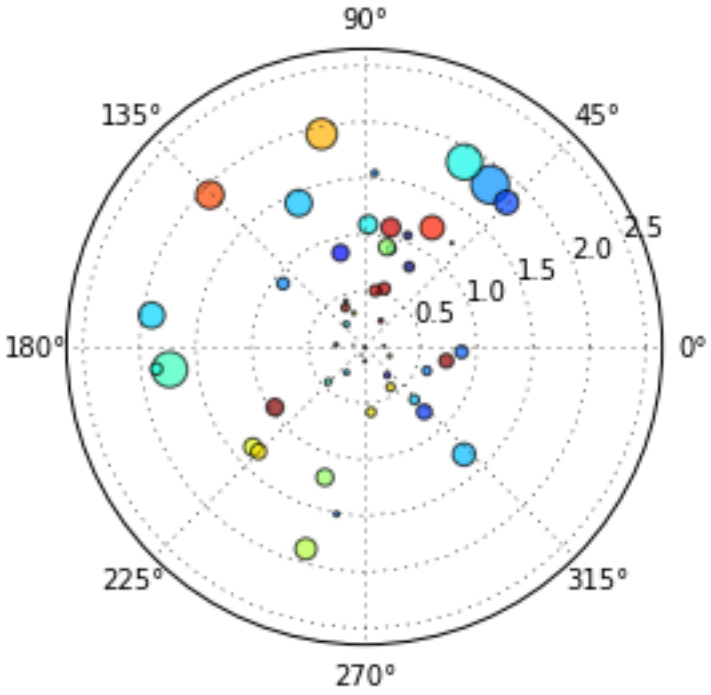


Resultado reproducido con Origin (ver online).

2.0.5 Diagramas con coordenadas polares:

```
In [28]: N = 50
         radio = 2 * np.random.rand(N)
         theta = 2 * np.pi * np.random.rand(N)
         area = 75 * radio**2 * np.random.rand(N)
         colores = np.random.rand(N)

         # Crea una figura con coordenadas polares
         ax = plt.subplot(111, polar=True)

         # Representa una diagrama de dispersión
         c = plt.scatter(theta, radio, c=colores,
                         s=area, alpha=0.70)
```

```
# Opcional si queremos salvar las figuras como archivos
#plt.savefig(r'figuras/coordenadas-polares.pdf')
#plt.savefig(r'figuras/coordenadas-polares.png',dpi=150)

plt.show()
```



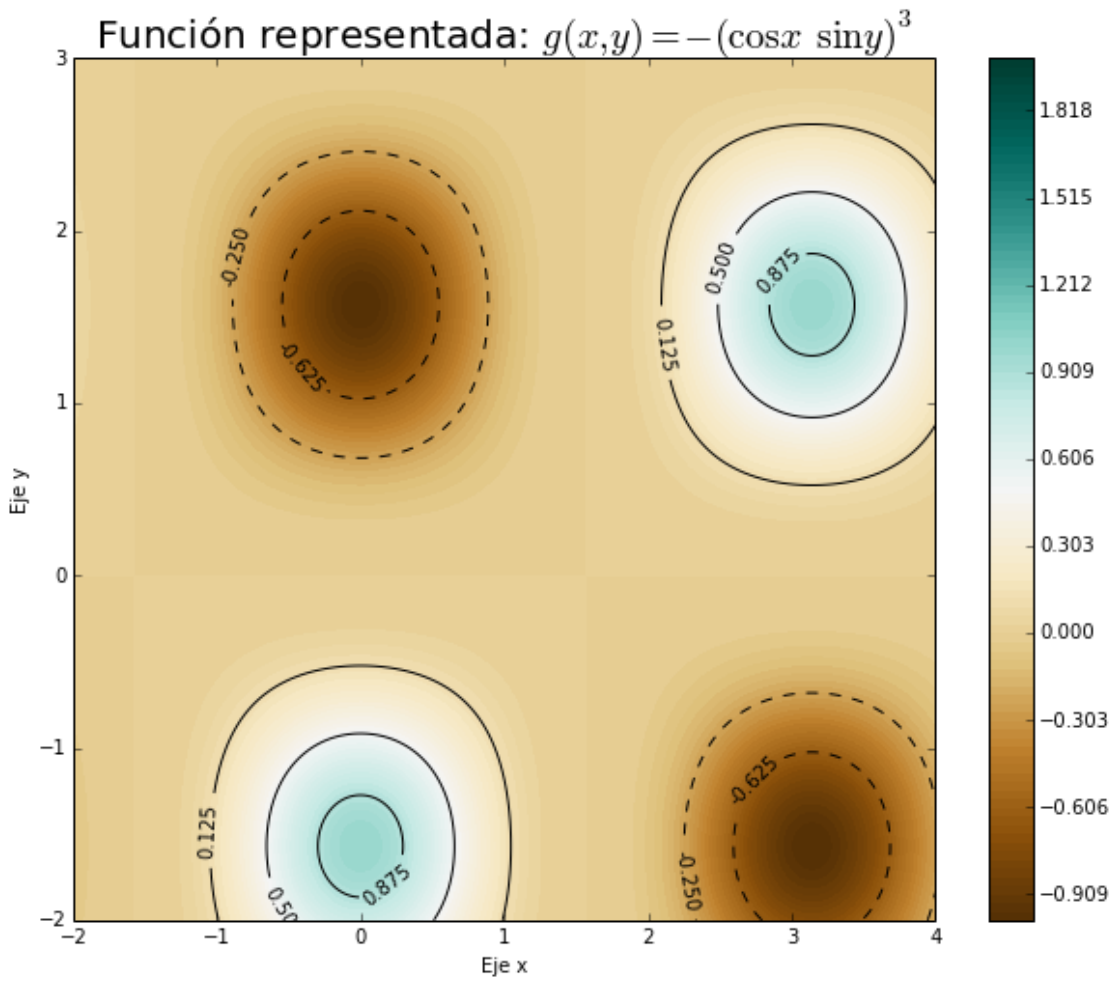
Resultado reproducido con Origin:

2.0.6 Diagramas de contorno:

Otro tipo de gráfico que Excel® presenta limtaciones son los diagramas de contorno. Veamos cómo hacerlo con matplotlib:

In [9]:
```
# Función que queremos representar
def g(x, y):
    return -(np.cos(x) * np.sin(y))**3

# Genera dos vectores para la malla
# valor numérico alto para ver resultados de la curva suaves
x = np.linspace(-2, 4, 1000)
y = np.linspace(-2, 3, 1000)

# Creamos la malla
xx, yy = np.meshgrid(x, y)

# Calculamos los valores de la función para cada punto de la malla
zz = g(xx, yy)
```

```
# Ajusta el tamaño de la figura con figsize
fig, axes = plt.subplots(figsize=(10, 8))

# Asigna la salida a la variable cs para luego crear el colorbar
cs = axes.contourf(xx, yy, zz, np.linspace(-1, 2, 100), cmap=plt.cm.BrBG)

# Con ‘colors='k'‘ dibujamos todas las líneas negras
# Asigna la salida a la variable cs2 para crear las etiquetas
cs2 = axes.contour(xx, yy, zz, np.linspace(-1, 2, 9), colors='k')

# Crea las etiquetas sobre las líneas
axes.clabel(cs2)

# Crea la barra de colores (nótese que pertenece a fig)
fig.colorbar(cs)

# Ponemos las etiquetas de los ejes
axes.set_xlabel("Eje x")
axes.set_ylabel("Eje y")
axes.set_title(u"Función representada: $g(x, y) = - (\cos{x} \, \sin{y})^3$",fontsize=20)


# Opcional si queremos salvar las figuras en archivos
#fig.savefig(r'figuras/diagrama-contorno.pdf')
#fig.savefig(r'figuras/diagrama-contorno.png',dpi=150)

# Muestra figura
plt.show()
```



Resultado reproducido con Origin (ver online).

2.1 Gráficas en 3D

Una de las cosas más llamativas que publicita OriginLab® es el uso de su herramienta para graficar resultados en 3D. Ciertamente, Excel® dispone de pocas herramientas en ese aspecto y MATLAB® tiene el mismo problema que Python (hay que aprender a programar) con el añadido del coste de licencia extra. Por ello la solución más cómoda es usar OriginLab®. No te ahorras la licencia pero te evitas tener que aprender programar.

Existe una regla no escrita que dice algo así: ***"si la única forma de representar tus resultados es con una gráfica 3D, es que algo estás haciendo mal"***

Sin ánimo de ser tan radicales, vivimos en un mundo 2D. El papel o pantalla donde se verán tus resultados son en 2D y por ello es mejor evitar representar resultados mediante gráficas 3D. No obstante, para exploración y visualización de resultados interactiva, son una herramienta muy potente. Cabe destacar que matplotlib empezó a dar soporte a figuras en 3D recientemente. Si se requieren visualizaciones más complejas es recomendable usar Mayavi

Veamos como hacer esto con matplotlib:

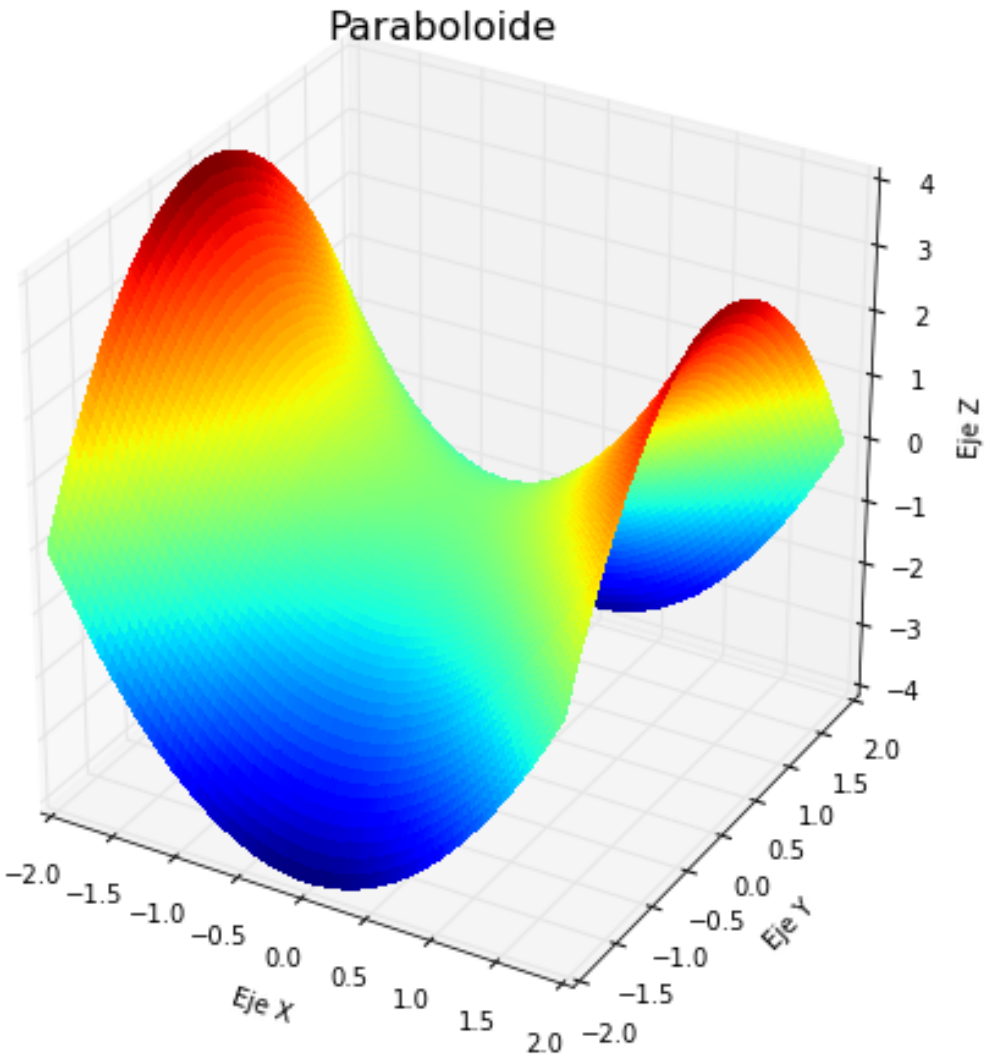
```
In [10]: # Cargamos la librería 3D de matplotlib
         from mpl_toolkits.mplot3d import axes3d
```

```
        # Acceso rápido a los mapas de colores (colormap)
        from matplotlib import cm
```

2.2 Superficie en 3D

Empecemos con algo sencillo. Vamos a representar la superficie de un parboloide.

```
In [11]: # Crea los vectores x e y
         x = np.arange(-2, 2, 0.05)
         y = np.arange(-2, 2, 0.05)

         # Genera el mallado 2D
         X, Y = np.meshgrid(x,y)

         # Calcula Z: Paraboloide
         Z = (X)**2-(Y)**2

         # Tamaño de figura
         fig = plt.figure(figsize=(8,8))

         # Proyección de la figura en 3D
         ax = fig.gca(projection='3d')

         # Dibuja los resultados
         ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=cm.jet,
                 linewidth=0, antialiased=False)

         # Texto de las etiquetas
         ax.set_xlabel("Eje X")
         ax.set_ylabel("Eje Y")
         ax.set_zlabel("Eje Z")
         ax.set_title("Paraboloide", fontsize=16)

         # Opcional si queremos salvar las figuras en archivos
         #fig.savefig(r'figuras/superficie-3D.pdf')
         #fig.savefig(r'figuras/superficie-3D.png',dpi=150)

         plt.show()
```

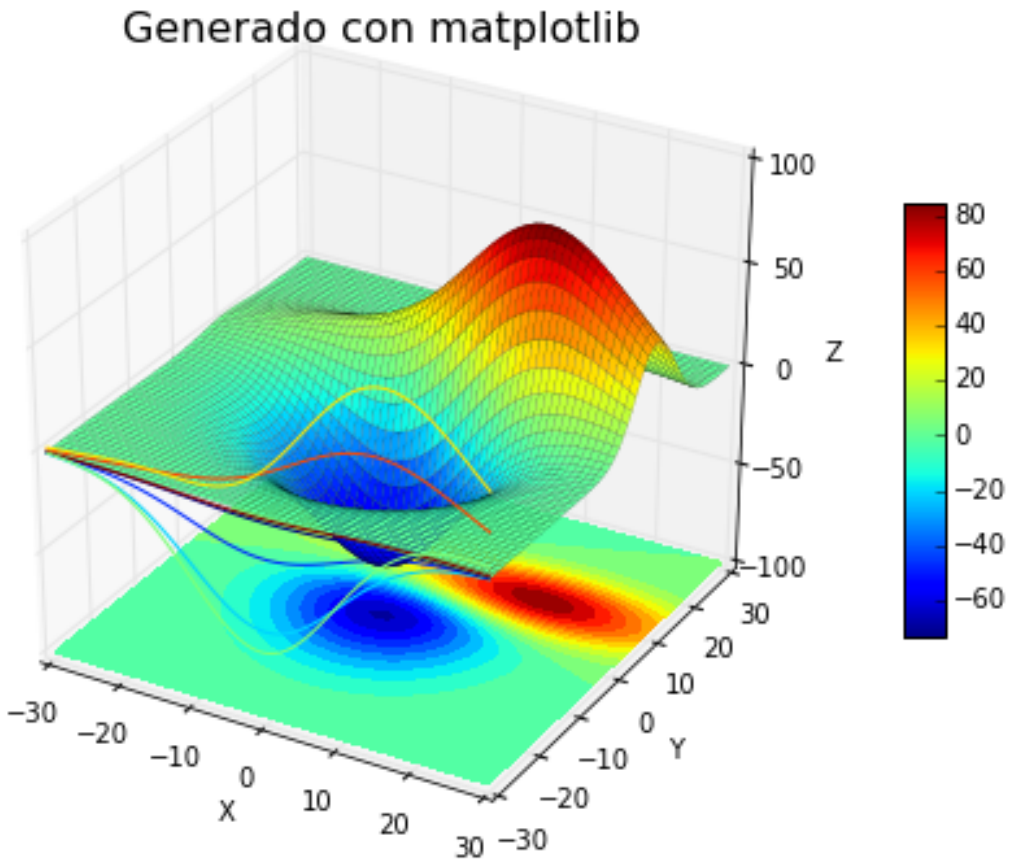


Resultado reproducido con Origin (ver online).

2.2.1 Superficies 3D combinadas con diagramas de contorno

```
In [12]: fig = plt.figure(figsize=(8,6))

         ax = fig.gca(projection='3d')

         # Carga datos de para test
         X, Y, Z = axes3d.get_test_data(0.01)

         surf = ax.plot_surface(X, Y, Z, rstride=10, cstride=10, alpha=1,
                        cmap=cm.jet, linewidth=0.1)

         # Proyección en la base
         cset = ax.contourf(X, Y, Z, 25, zdir='z', offset=-100, cmap=cm.jet, )
```

```
# Proyección de líneas en el lateral
cset = ax.contour(X, Y, Z, zdir='y', offset=-30, cmap=cm.jet)

# Añade barra de color con valores de Z
fig.colorbar(surf, shrink=0.5, aspect=10)

# Configuración de ejes y etiquetas
ax.set_xlabel('X')
ax.set_xlim(-30, 30)
ax.set_ylabel('Y')
ax.set_ylim(-30, 30)
ax.set_zlabel('Z')
ax.set_zlim(-100, 100)

ax.set_title("Generado con matplotlib", fontsize=16)

# Opcional si queremos salvar las figuras en archivos
#fig.savefig(r'figuras/superficie-3D-combinado.pdf')
#fig.savefig(r'figuras/superficie-3D-combinado.png',dpi=150)

plt.show()
```

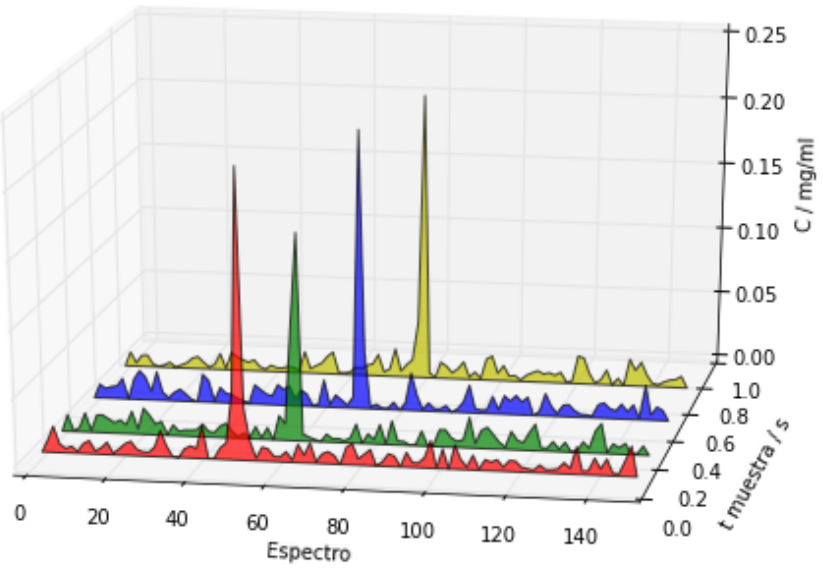


Resultado reproducido con Origin (ver online).

2.2.2 Cascada 3D

```
In [13]: from mpl_toolkits.mplot3d import Axes3D
         from matplotlib.collections import PolyCollection
         from matplotlib.colors import colorConverter
         import matplotlib.pyplot as plt
         import numpy as np

         # Para poder crear las señales usaremos una funcion de Gauss
         # y le añadiremos ruido blanco

         def gauss_ruido(x,posicion,desviacion):
             ''' Genera una curva de Gauss con ruido blanco
             con la posicion y desviacion estándar de entrada'''
             y = np.exp( -(x-posicion)**2) / (2*np.sqrt(desviacion))
             ruido = 0.01* np.random.randn(len(x))
             senyal = y + ruido
             return senyal


         fig = plt.figure(figsize=(10,6))
         ax = fig.gca(projection='3d')

         cc = lambda arg: colorConverter.to_rgba(arg, alpha=0.6)



         posicion= [50, 60, 70, 80]


         verts = []
         for pos in posicion:
             x = np.linspace(0,150,100)

             y = np.abs(gauss_ruido(x,pos,5))

             y[0], y[-1] = 0, 0
             verts.append(list(zip(x, y)))


         poly = PolyCollection(verts, facecolors = [cc('r'), cc('g'),
                                                    cc('b'), cc('y')])
         poly.set_alpha(0.7)

         # Posicion de z
         zs = [0.1, 0.25, 0.50, 0.75]

         # Dibuja los poligonos generados seleccionado
         # la señal y como eje vertical
         ax.add_collection3d(poly, zs=zs, zdir='y')

         ax.set_xlabel('Espectro')
```

```
ax.set_xlim3d(0, 150)
ax.set_ylabel('t muestra / s')
ax.set_ylim3d(0, 1)
ax.set_zlabel(u'C / mg/ml')
ax.set_zlim3d(0, 0.25)

# Ángulo
ax.view_init(20,280)

# Opcional si queremos salvar las figuras en archivos
#fig.savefig(r'figuras/cascada-3D.pdf')
#fig.savefig(r'figuras/cascada-3D.png',dpi=150)


plt.show()
```



Resultado reproducido con Origin (ver online).

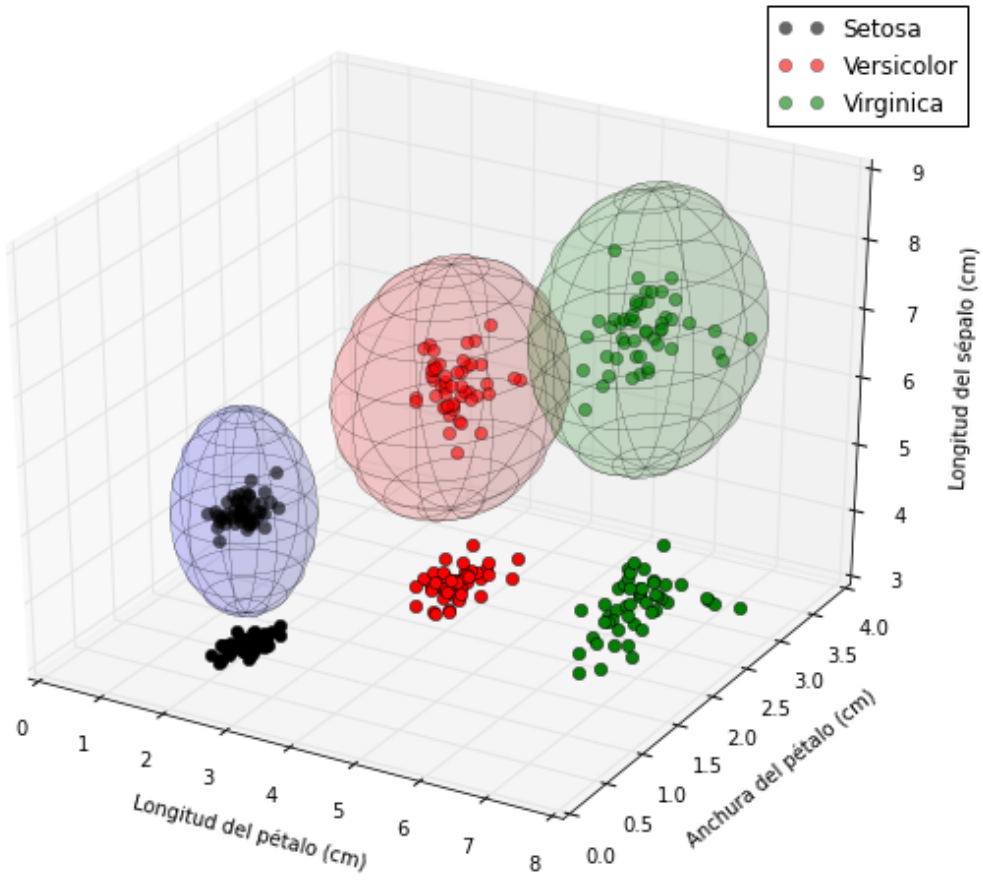
2.2.3 Diagramas de dispersión en 3D:

Por simplicidad, el siguiente ejemplo creará los datos con funciones de números aleatorios. No obstante, se podría hacer el agrupamiento de datos Iris con Python y scikit-learn.

```
In [14]: fig = plt.figure(figsize=(10,8))
         ax = fig.gca(projection='3d')

         # Genereamos los puntos con una distribución estándar normal de (n_puntos, dimensiones)
         # y su repreesntación 3d junto a la proyección en el plano x-y
         z_offset = 3
```

```
coordenadas_clase1 = np.array([2,1,5])
clase1 = 0.15 * np.random.standard_normal((50,3)) + coordenadas_clase1
ax.plot(clase1[:,0],clase1[:,1],clase1[:,2],'ko', alpha=0.6, label='Setosa')

ax.plot(clase1[:,0], clase1[:,1], np.zeros_like(clase1[:,2])+z_offset, 'ko')

coordenadas_clase2 = np.array([3.5,2.5,6])
clase2 = 0.3 * np.random.standard_normal((50,3)) + coordenadas_clase2
ax.plot(clase2[:,0],clase2[:,1],clase2[:,2],'ro', alpha=0.6, label='Versicolor')
ax.plot(clase2[:,0], clase2[:,1], np.zeros_like(clase2[:,2])+z_offset, 'ro')

coordenadas_clase3 = np.array([6,3,7])
clase3 = 0.4 * np.random.standard_normal((50,3)) + coordenadas_clase3
ax.plot(clase3[:,0],clase3[:,1],clase3[:,2],'go', alpha=0.6, label='Virginica')
ax.plot(clase3[:,0], clase3[:,1], np.zeros_like(clase3[:,2])+z_offset, 'go')

# Generamos las esferas

u1 = np.linspace(0, 2 * np.pi, 100)
v1 = np.linspace(0, np.pi, 100)

x_esfera_1 = 1   * np.outer(np.cos(u1), np.sin(v1)) + coordenadas_clase1[0]
y_esfera_1 = 0.5 * np.outer(np.sin(u1), np.sin(v1)) + coordenadas_clase1[1]
z_esfera_1 = 1.5 * np.outer(np.ones(np.size(u1)), np.cos(v1)) + coordenadas_clase1[2]
ax.plot_surface(x_esfera_1, y_esfera_1, z_esfera_1,
                rstride=10, cstride=10, linewidth=0.1, color='b', alpha=0.1)

u2 = np.linspace(0, 2 * np.pi, 100)
v2 = np.linspace(0, np.pi, 100)

x_esfera_2 = 1.5 * np.outer(np.cos(u2), np.sin(v2)) + coordenadas_clase2[0]
y_esfera_2 = 1   * np.outer(np.sin(u2), np.sin(v2)) + coordenadas_clase2[1]
z_esfera_2 = 1.8   * np.outer(np.ones(np.size(u2)), np.cos(v2)) + coordenadas_clase2[2]
ax.plot_surface(x_esfera_2, y_esfera_2, z_esfera_2,
                rstride=10, cstride=10, linewidth=0.1, color='r', alpha=0.1)

u3 = np.linspace(0, 2 * np.pi, 100)
v3 = np.linspace(0, np.pi, 100)

x_esfera_3 = 1.5   * np.outer(np.cos(u3), np.sin(v3)) + coordenadas_clase3[0]
y_esfera_3 = 1     * np.outer(np.sin(u3), np.sin(v3)) + coordenadas_clase3[1]
z_esfera_3 = 2   * np.outer(np.ones(np.size(u3)), np.cos(v3)) + coordenadas_clase3[2]
ax.plot_surface(x_esfera_3, y_esfera_3, z_esfera_3,
                rstride=10, cstride=10, linewidth=0.1, color='g', alpha=0.1)

# Establcemos límites en los ejes y los etiquetamos
ax.set_xlim3d(0, 8)
ax.set_ylim3d(0, 4)
ax.set_zlim3d(z_offset, 9)

ax.set_xlabel(u'Longitud del pétalo (cm)')
ax.set_ylabel(u'Anchura del pétalo (cm)')
ax.set_zlabel(u'Longitud del sépalo (cm)')
```

```
# Muestra leyenda
ax.legend()

# Opcional si queremos salvar las figuras en archivos
#fig.savefig(r'figuras/dispersion-3D.pdf')
#fig.savefig(r'figuras/dispersion-3D.png',dpi=150)

plt.show()
```



Resultado reproducido con Origin (ver online).

2.3 Un paso más allá

Como se ha comentado, Python dispone de diversas librerías. Una de las más interesante es SymPy que proporciona poderosas funciones sistemas de álgebra computacional (CAS, computer algebraic system). El siguiente código se puede ejecutar sin necesidad de las anteriores celdas ya que es independiente. El propósito es una demostración de la interactivdad con los nuevos widgets de IPython Notebook 2.0.

```
In [15]: # Cargamos widgets interactivos de IPython Notebook

         from IPython.html.widgets import interact
         from IPython.display import display
```

In [16]:
```
# Cargamos las partes de SymPy que vamos a utilizar

from sympy import Symbol, Eq, factor, init_printing
init_printing(use_latex='mathjax')
```

In [17]:
```
# Creamos la variable simbólica x
x = Symbol('x')
```

In [18]:
```
# Creamos la función de factorización para el ejemplo
def factorit(n):
    display(Eq(x**n-1, factor(x**n-1)))
```

Cuando ejectuemos la función, nos devolverá un resultado en LaTeX

In [19]:
```
factorit(12)
```

$$x^{12} - 1 = (x - 1)(x + 1)\left(x^2 + 1\right)\left(x^2 - x + 1\right)\left(x^2 + x + 1\right)\left(x^4 - x^2 + 1\right)$$

In [20]:
```
interact(factorit, n=(2,20));
```

$$x^{11} - 1 = (x - 1)\left(x^{10} + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1\right)$$

Es hora de hacer algo más interesante. Hagamos una expansión por serie de Taylor de $\frac{sinx}{x}$

In [21]:
```
from sympy import Symbol, sin, series, exp
x = Symbol('x')

ecuacion_ejemplo =  sin(x)/x
orden = 5

series(ecuacion_ejemplo, x, n=orden)
```

Out[21]:

$$1 - \frac{x^2}{6} + \frac{x^4}{120} + \mathcal{O}\left(x^5\right)$$

Vamos a representar los resultados de forma interactiva

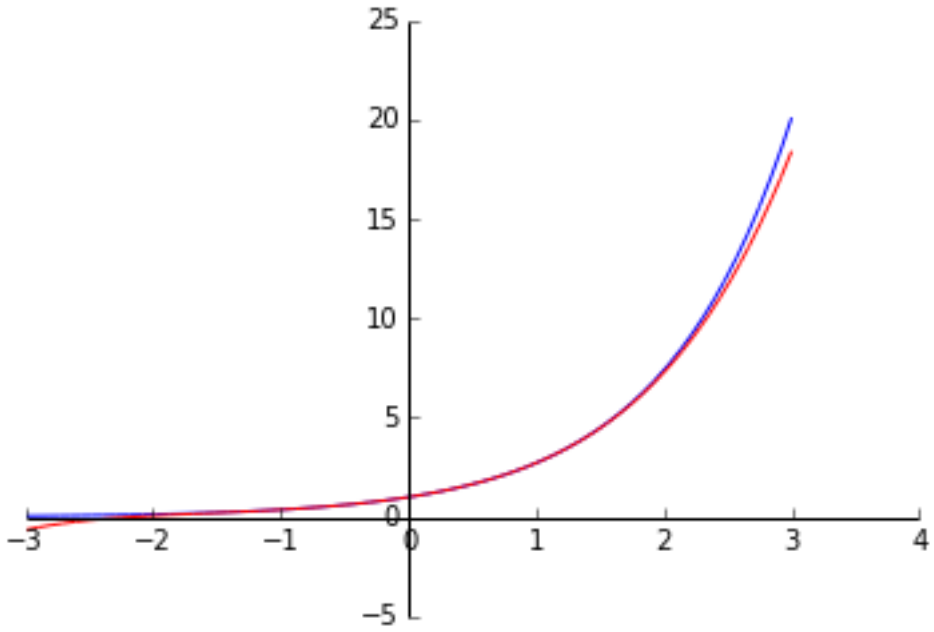
In [22]:
```
from sympy.plotting import plot

# Representa las figuras en línea con el documento (no en una ventana emergente)
%matplotlib inline
```

In [23]:
```
def taylor_graf(n):

    e = exp(1)
    ecuacion = e**x

    #calculamos la expansion elimnando el termino del error
    ecuacion_aprox = series(ecuacion, x, n=n+1).removeO()

    p1 = plot(ecuacion, (x, -3, 3), show=False, line_color='b', label='ecuacion')
    p2 = plot(ecuacion_aprox, (x, -3, 3), show=False, line_color='r', label='aprox')

    # Haz la segunda función parte de la primera
```

```
        p1.extend(p2)
        # Nota: Este código se debe a que representar dos funciones juntas con diferentes colores
        # aun no está implmentado
        # http://stackoverflow.com/questions/21429866/change-color-implicit-plot-sympy

        p1.show()
```

Función exponencial e^x (en azul) y la suma de los primeros n+1 términos de su serie de Taylor en torno a cero (en rojo).

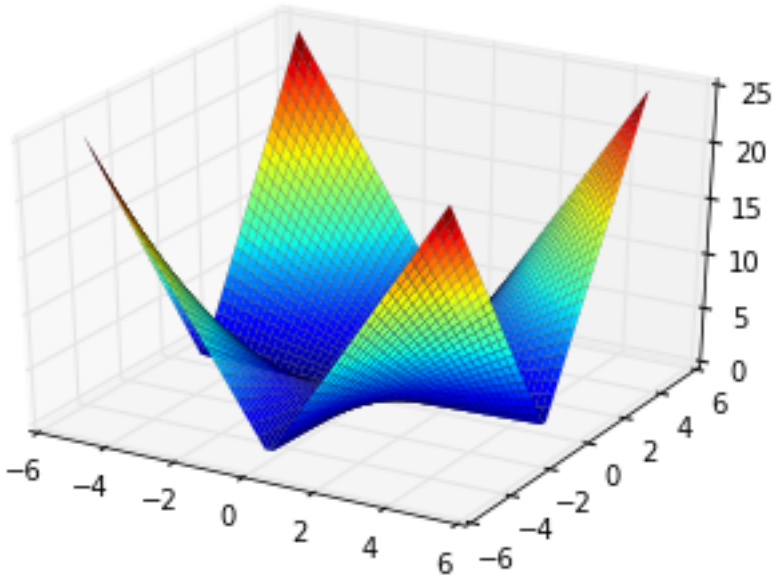
```
In [24]: interact(taylor_graf, n=(0,10));
```



Por último, SymPy también representa expresiones matemáticas en 3D:

```
In [25]: from sympy import symbols
         from sympy.plotting import plot3d

         x, y = symbols('x y')
```

```
In [29]: plot3d(abs(x*y), (x, -5, 5), (y, -5, 5))
```



Out[29]: <sympy.plotting.plot.Plot at 0x1370da58>

Este IPython Notebook puede imprimirse en PDF generando (previamente) un archivo ".tex" LaTeXmediante pandoc. Por ejemplo para obtener el archivo basta con ejecutar en la línea de comandos el siguiente código

```
ipython nbconvert --to latex originlab-python.ipynb
```

Si se desea obtener más información, consultar documentación.