Guías para Métodos Numéricos

Héctor F. Villafuerte

Agosto, 2010

Contenido, parte II

Guía	Breve Descripción
Guía 9	Polinomios, Interpolación de Lagrange, Trazadores.
Guía 10	Ajuste de curvas, mínimos cuadrados ordinarios.
Guía 11	Ajuste de curvas, descomposición (factorización) de valor singular.
Guía 12	Derivación numérica, diferencias finitas.
Guía 13	Integración numérica, métodos del trapecio y Simpson.
Guía 14	Métodos de Monte Carlo.
Guía 15	Ecuaciones diferenciales ordinarias.

Métodos Numéricos **Guía 9**
UVG-MM2010, MM2022

Instrucciones: En sus cursos de matemática se le ha recomendado que ataque los problemas desde varios enfoques: **analítico**, **gráfico**, y **numérico**. En este curso nos interesan particularmente los últimos dos enfoques. Responda las siguientes preguntas dejando constancia clara de su procedimiento; recuerde incluir **gráficas**, **código en Python**, y explicaciones que considere pertinentes.

1. Introducción

En Python existen varios comandos para tratar el problema de *interpolación y ajuste de curvas*, e.g. `lstsq`, `polyfit`, `linregress`, `polyfit`, `lagrange`, `splrep`, `splev`, etc. Ejemplificaremos su uso con algunos ejercicios resueltos, pero antes veamos el tratamiento básico de polinomios.

1.1. Polinomios

Los comandos `poly1d` y `poly` nos proporcionan formas básicas de construcción de polinomios en una variable. Por ejemplo,

$$p(x) = x^2 - 1$$
$$q(x) = x^3 + 1$$
$$r(x) = (x-1)(x+1/2)(x-4)$$

En Python, noten el uso del comando `roots` para encontrar las raíces de polinomios:

```
In [34]: p = poly1d([1, 0, -1]); p
Out[34]: poly1d([ 1,  0, -1])
In [35]: q = poly1d([1, 0, 0, 1]); q
Out[35]: poly1d([1, 0, 0, 1])
In [36]: r = poly([1, -0.5, 4]); r
Out[36]: array([ 1. , -4.5,  1.5,  2. ])
In [37]: roots(p)
Out[37]: array([-1.+0.j,  1.+0.j])
In [38]: roots(q)
Out[38]: array([-1.0+0.j, 0.5+0.8660254j, 0.5-0.8660254j])
In [39]: roots(r)
Out[39]: array([ 4.0+0.j, 1.0+0.j, -0.5+0.j])
```

Ahora para graficarlos necesitamos *evaluar* el polinomio con `polyval`; aunque una forma más natural de evaluar polinomios en Python es usar notación funcional $f(x)$ como se ejemplifica para el polinomio r:

```
In [44]: x = linspace(-1,1)
In [45]: plot(x, polyval(p,x))
In [46]: plot(x, polyval(q,x))
In [47]: plot(x, r(x))  # notacion funcional
```

```
In [48]: grid()
```

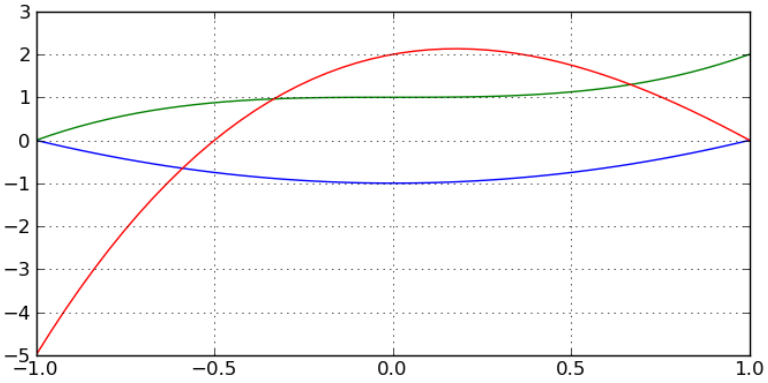


Figura 1: $p(x), q(x), r(x)$

Para más información: `http://docs.scipy.org/doc/numpy/reference/routines.poly.html`

1.2. Problemas Resueltos

1. Encuentre el polinomio de grado no mayor a tres que interpola los siguientes puntos:

x	y
-3	4
-1	3
1	3
5	-2

Emplee interpolación de Lagrange.

Solución: Acá podemos emplear los comandos `polyfit`, o `lagrange`:

```
from pylab import *
import scipy.interpolate as si

x = array([-3., -1, 1, 5])
y = array([4., 3, 3, -2])

P1 = polyfit(x,y,3)
P2 = si.lagrange(x, y)

allclose(polyval(P1,x), y)
allclose(P2(x), y)
```

Ambos comandos devuelven el mismo polinomio interpolante $P(x) = -0.04167x^3 + 0.04167x + 3$ (noten que el coeficiente de x^2 es cero).

2. Considere la *función de Runge* $f(x) = 1/(1+25x^2)$. El *fenómeno de Runge* consiste en tratar de interpolar esta función con un solo polinomio de Lagrange.

   (a) Encuentre el polinomio de Lagrange para la función de Runge en `x=linspace(-1,1,10)`

Solución:

```
x = linspace(-1,1,100)
y = 1/(1 + 25*x**2)
xi = linspace(-1,1,10)
yi = 1/(1 + 25*xi**2)
P = si.lagrange(xi, yi)
plot(x,y)
plot(xi, yi, 'o')
plot(x, P(x))
grid()
```

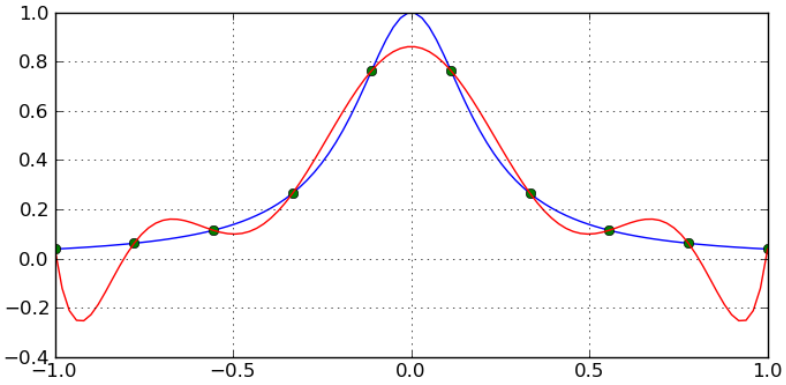


Figura 2: Función de Runge y un polinomio de Lagrange

Importante: note lo malo que resulta usar sólo un polinomio interpolante con esta función. La solución es tomar polinomios interpolantes por trozos, ésto es lo que llamaremos *trazadores* ("spline", en inglés).

   (b) Interpole esta función empleando un trazador cúbico natural tomando $n = 8, 12$ muestras de la función de Runge en $x \in [-1, 1]$. Reporte sus observaciones.

Solución: Con el siguiente código encontramos dos polinomios cúbicos. Pongan especial atención en los comandos `splrep` (encuentra la representación del trazador), `splev` (evalúa el trazador).

```
from pylab import *
import scipy.interpolate as si

x = linspace(-1,1,100)
y = 1./(1 + 25*x**2)
```

```
6
7  def trazador_cub(n):
8      xi = linspace(-1,1,n)
9      yi = 1./(1 + 25*xi**2)
10     tck = si.splrep(xi,yi,s=0)
11     return tck
12
13 tck = trazador_cub(8)
14 ys8 = si.splev(x, tck)
15
16 tck = trazador_cub(12)
17 ys12 = si.splev(x, tck)
18
19 plot(x,y, x,ys8,'og', x,ys12,'or')
20 grid()
```

Con lo que obtenemos la siguiente gráfica:

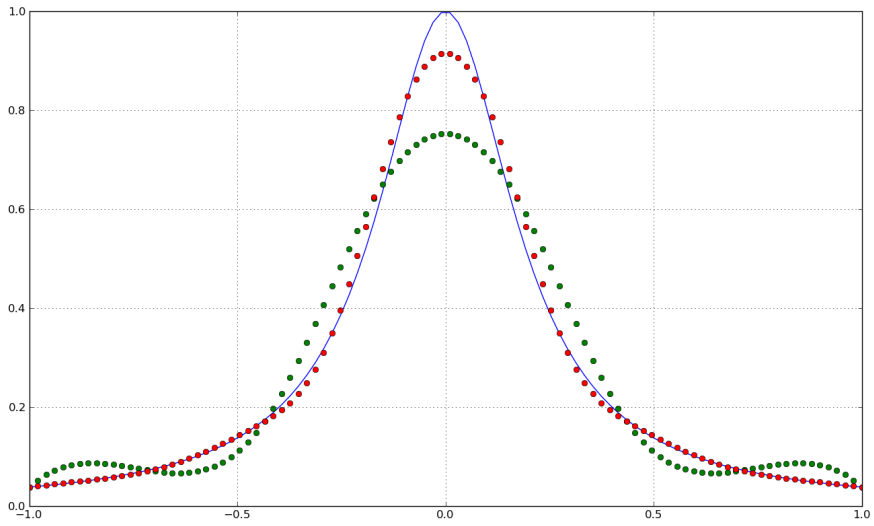


Figura 3: Función de Runge y un par de trazadores cúbicos

Noten cómo mejora la interpolación al usar trazadores en lugar de un solo polinomio interpolante.

2. Preguntas

3. Sea $P(x)$ el polinomio interpolante para la función $y = \cos(x)$ en n puntos equidistantes $(x_1, y_1), \ldots, (x_n, y_n)$ en el intervalo $[0, 2]$. Considere el error de interpolación como $e(x) = P(x) - \cos(x)$. Grafique $e(x)$ para

$n = 6, 10, 12$. Reporte sus observaciones.

Solución: Primero encontramos los distintos errores de interpolación:

```
import scipy.interpolate as si

def pol_int_cos(n):
    xi = linspace(0,2,n)
    yi = cos(xi)
    P = si.lagrange(xi, yi)
    return P

P6 = pol_int_cos(6)
P10 = pol_int_cos(10)
P12 = pol_int_cos(12)

x = linspace(0,2,100)
e6 = P6(x) - cos(x)
e10 = P10(x) - cos(x)
e12 = P12(x) - cos(x)
```

Con lo que obtenemos las siguientes gráficas (note el orden de magnitud en el eje y):

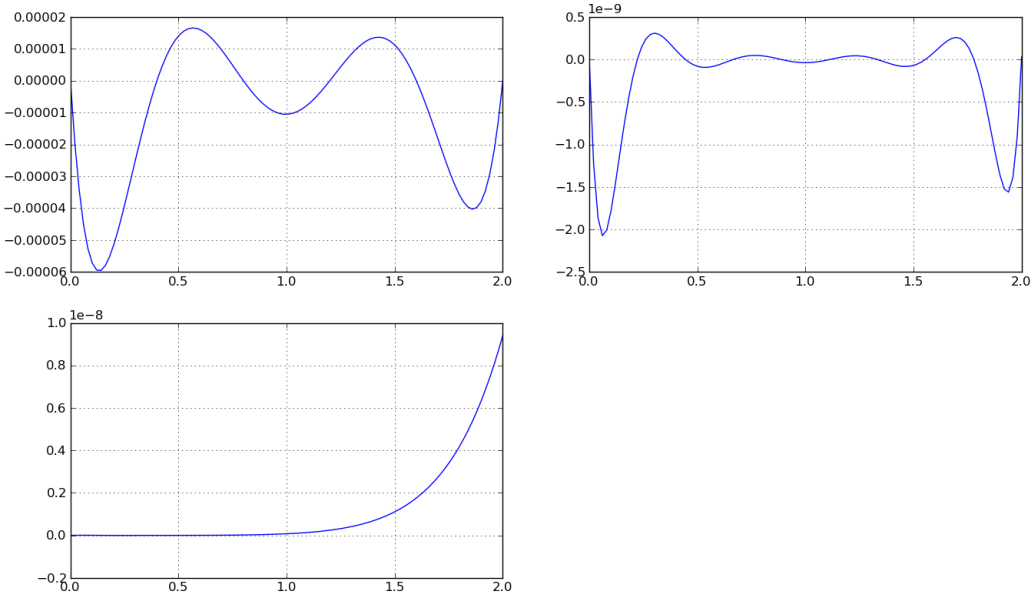


Figura 4: $e(x)$ para $n = 6, 10, 12$

4. En el campo de "Computer graphics" es necesario trazar curvas empleando interpolación.

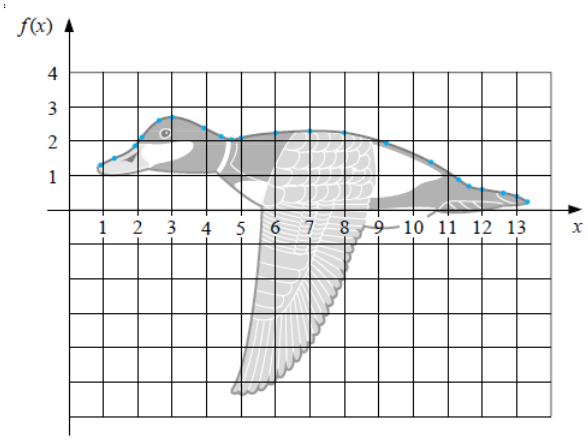


Figura 5: Modele el contorno del pato.

(a) En la figura anterior, tome 20 puntos del contorno del pato (sólo en el primer cuadrante).

Solución: Aproximadamente los puntos son:

```
P = [(0.9, 1.3), (1.3, 1.5), (1.9, 1.8), (2.1,2.1), (2.6, 2.6), (3.0, 2.7),
     (3.9, 2.3), (4.4, 2.1), (4.8, 2.0), (5.0, 2.1), (6, 2.2), (7, 2.3),
     (8, 2.2), (9.1, 1.9), (10.5, 1.4), (11.2, 0.9), (11.6, 0.8), (12, 0.6),
     (12.6, 0.5), (13, 0.4), (13.2, 0.2)]
```

(b) Con los puntos anteriores, encuentre el polinomio interpolante (usando `polyfit`, `lagrange`). Grafique. ¿Considera que este polinomio interpolante es un buen modelo para describir el contorno del pato?

Solución:

```
xi = array([k[0] for k in P])
yi = array([k[1] for k in P])
x = linspace( min(xi), max(xi), 1000)

L = si.lagrange(xi,yi)
plot(xi,yi,'o')
plot(x,L(x))
axis([0,14,0,4])
grid()
```

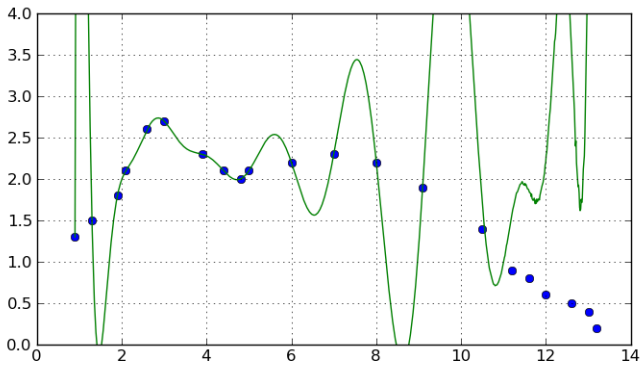


Figura 6: Contorno del pato con un solo polinomio de Lagrange.

Un solo polinomio de Lagrange es un muy mal modelo para el contorno del pato.

(c) Con los puntos anteriores, encuentre los trazadores (usando `splrep`, `splev`). Grafique. ¿Considera que estos trazadores son un buen modelo para describir el contorno del pato?

Solución:

```
tck = si.splrep(xi, yi, s=0)
ysp = si.splev(x, tck)
plot(xi,yi,'o')
plot(x,ysp)
axis([0,14,0,4])
axis('equal')
grid()
```

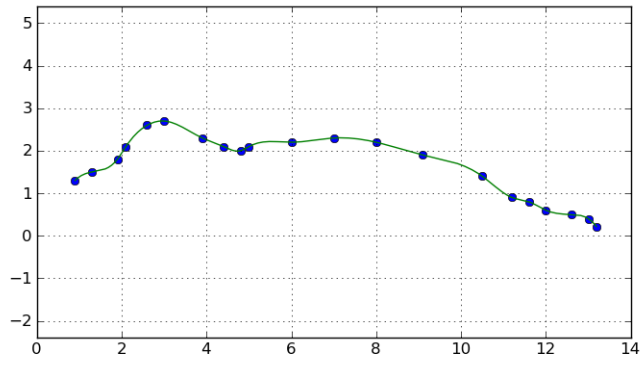


Figura 7: Contorno del pato con trazadores.

Es claro que con trazadores obtenemos mucho mejores resultados para modelar el contorno del pato.

Métodos Numéricos
UVG-MM2010, MM2022

Guía 10

Instrucciones: En sus cursos de matemática se le ha recomendado que ataque los problemas desde varios enfoques: **analítico**, **gráfico**, y **numérico**. En este curso nos interesan particularmente los últimos dos enfoques. Responda las siguientes preguntas dejando constancia clara de su procedimiento; recuerde incluir **gráficas**, **código en Python**, y explicaciones que considere pertinentes.

1. Introducción

En el laboratorio anterior empezamos a tratar el problema de *interpolación y ajuste de curvas*. En este lab nos concentraremos en el ajuste de curvas, en particular, en la técnica de *mínimos cuadrados ordinarios*. En Python existen varios comandos para tratar este problema, e.g. `lstsq`, `polyfit`, `linregress`.

1.1. Mínimos cuadrados ordinarios

En ciencia y en ingeniería frecuentemente recopilamos observaciones a las cuales deseamos asignarles un modelo matemático. Uno de los modelos más útiles es el *modelo lineal*. Asuma que se han tomado los siguientes datos y se desea encontrar un modelo lineal $y = mx + b$.

x	y
0.0	0.000
0.1	0.078
0.2	0.138
0.3	0.192
0.4	0.244

Ingresamos estos puntos en Python:

```
In [158]: P = [(0.0, 0.000),(0.1, 0.078),(0.2, 0.138),(0.3, 0.192),(0.4, 0.244)]
In [159]: x = array([p[0] for p in P])
In [160]: y = array([p[1] for p in P])
In [161]: x
Out[161]: array([ 0. ,  0.1,  0.2,  0.3,  0.4])
In [162]: y
Out[162]: array([ 0.   ,  0.078,  0.138,  0.192,  0.244])
```

Básicamente el problema se reduce a encontrar los parámetros que describen la recta que mejor se ajusta a estos puntos, i.e. la pendiente m y el intercepto b. La deducción analítica consiste en minimizar el cuadrado del error entre el modelo $y(x_k)$ y los datos recopilados y_k:

$$\begin{aligned} e(m, b) &= (y(x_k) - y_k)^2 \\ &= (mx_k + b - y_k)^2 \end{aligned}$$

Dado que $e(m, b)$ es una función de varias variables, se puede minimizar encontrando m y b que satisfagan $\nabla e = \vec{0}$. Así encontramos las fórmulas que ha visto en otros cursos (como Modelos Matemáticos 1, Estadística). Sin embargo, acá trabajaremos directamente con algunos comandos en Python; retomando:

```
In [163]: polyfit(x,y,1)
Out[163]: array([ 0.602,  0.01 ])
In [164]: from scipy import stats
In [166]: stats.linregress(x,y)
Out[166]:
(0.60199999999999998,
 0.010000000000000009,
 0.99633922564860877,
 0.00026573862022051591,
 0.031413806967119796)
In [174]: stats.linregress(x,y)[0:2]
Out[174]: (0.60199999999999998, 0.010000000000000009)
```

Nota: vea la documentación de `linregress` para poder interpretar el resto de los valores que devuelve. Tenemos entonces que ambos comando, `polyfit` y `linregress`, han determinado que la pendiente es $m = 0.602$ y que el intercepto es $b = 0.0.1$. Veamos nuestro modelo ahora con una gráfica:

```
In [176]: x
Out[176]: array([ 0. ,  0.1,  0.2,  0.3,  0.4])
In [177]: y
Out[177]: array([ 0.   ,  0.078,  0.138,  0.192,  0.244])
In [178]: plot(x,y,'o')
In [179]: plot(x, 0.602*x + 0.01)
In [180]: grid()
In [181]: axis([-0.05, 0.45, -0.05, 0.3])
```

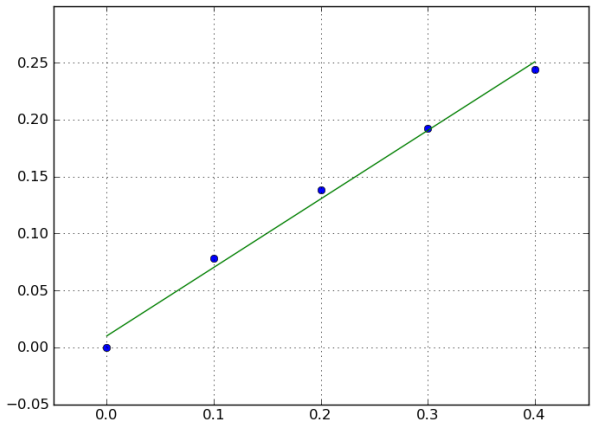


Figura 1: Recta de mejor ajuste

Consideremos ahora otro problema, uno en donde a priori sabemos cuál es la relación entre las variables, e.g. $y = -0.3x + 1.4$. Veamos qué devuelven los comandos que implementan el ajuste de mínimos cuadrados:

```
In [185]: x = linspace(0,5,10)
In [186]: x
Out[186]:
array([ 0.        ,  0.55555556,  1.11111111,  1.66666667,  2.22222222,
        2.77777778,  3.33333333,  3.88888889,  4.44444444,  5.        ])
In [187]: y = -0.3*x + 1.4
In [189]: polyfit(x,y,1)
Out[189]: array([-0.3,  1.4])
In [190]: stats.linregress(x,y)[0:2]
Out[190]: (-0.30000000000000004, 1.3999999999999999)
In [191]: plot(x,y,'o')
In [192]: plot(x, -0.3*x + 1.4)
In [193]: grid()
```

Como era de esperarse, los métodos devuelven exactamente el modelo que teníamos al principio.

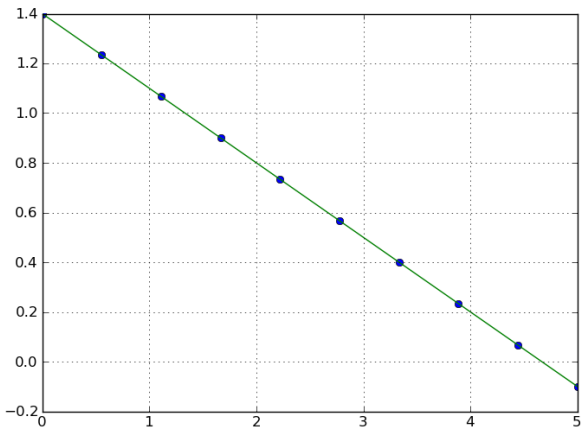


Figura 2: Recta de mejor ajuste con un modelo a priori

Consideremos un último ejemplo en donde nuevamente tenemos un modelo a priori pero *corrompemos* los datos con muestras de una distribución normal con $\mu = 0, \sigma = 1$ (empleando el comando `randn`).

```
In [200]: x = linspace(0,5)
In [201]: y = -0.3*x + 1.4 + 0.1*randn(len(x))
In [202]: plot(x,y,'o')
In [203]: grid()
In [204]: polyfit(x,y,1)
Out[204]: array([-0.31165352,  1.41384649])
In [205]: stats.linregress(x,y)[0:2]
In [206]: plot(x, -0.3116*x + 1.4138)
In [207]: plot(x, -0.3*x + 1.4)
```

Note que mínimos cuadrados hace un buen trabajo para recuperar los parámetros de la recta original. Claramente, mientras mayores sean las perturbaciones sufridas por los datos más difícil resulta modelarlos.

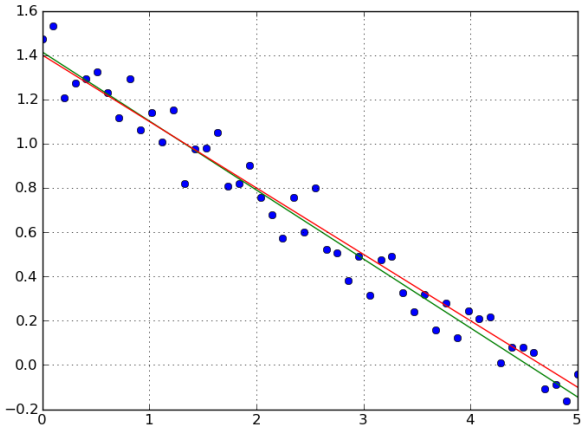


Figura 3: Recta de mejor ajuste con un modelo a priori con perturbaciones

2. Preguntas

1. Las enzimas actúan como catalizadores que aceleran las reacciones químicas. El *modelo de Michaelis-Menten* es comúnmente utilizado para describir dichas reacciones:

$$v = \frac{v_m[S]}{K_s + [S]}$$

Donde v es la velocidad de reacción, v_m es la velocidad máxima de reacción, $[S]$ es la concentración del sustrato y K_s es la costante de Michaelis-Menten. En el laboratorio se han recopilado los siguientes datos:

$[S]$	1.3	1.8	3	4.5	6	8	9
v	0.07	0.13	0.22	0.275	0.335	0.35	0.36

(a) Claramente el modelo de Michaelis-Menten describe una relación *no* lineal entre $[S]$ y v. Linealice dicho modelo. *Ayuda:* la clásica relación no lineal $y = \alpha e^{\beta x}$ puede linealizarse como $\ln y = \beta x + \ln \alpha$, donde puede apreciarse que existe una relación lineal entre x y $\ln y$, con pendiente β e intercepto $\ln \alpha$. ¿Cuál es entonces la técnica algebraica que linealiza el modelo de Michaelis-Menten?

Solución: En este caso la técnica algebraica es tomar el recíproco:

$$\begin{aligned} v &= \frac{v_m[S]}{K_s + [S]} \\ \frac{1}{v} &= \frac{K_s + [S]}{v_m[S]} \\ &= \frac{K_s}{v_m}\frac{1}{[S]} + \frac{1}{v_m} \end{aligned}$$

que exhibe una relación lineal entre $\frac{1}{[S]}$ y $\frac{1}{v}$, con pendiente $\frac{K_s}{v_m}$ e intercepto $\frac{1}{v_m}$.

(b) Utilice el método de mínimos cuadrados ordinarios para encontrar las constantes v_m, K_s en base a los datos recopilados en el laboratorio. Muestre su código.

Solución:

```
from pylab import *
S = array([1.3, 1.8, 3, 4.5, 6, 8, 9])
v = array([0.07, 0.13, 0.22, 0.275, 0.335, 0.35, 0.36])
a,b = polyfit(1/S, 1/v, 1)
Ks = a/b
vm = 1/b
```

Donde $v_m = 5.25, K_s = 86.22$. Por lo que el modelo es:

$$v = \frac{5.25[S]}{86.22 + [S]}$$

(c) En una sola gráfica muestre los puntos del lab y el modelo que encontró.

Solución:

```
Smod = linspace(1,10)
vmod = vm*Smod/(Ks + Smod)
plot(S,v,'o', Smod, vmod)
grid()
show()
```

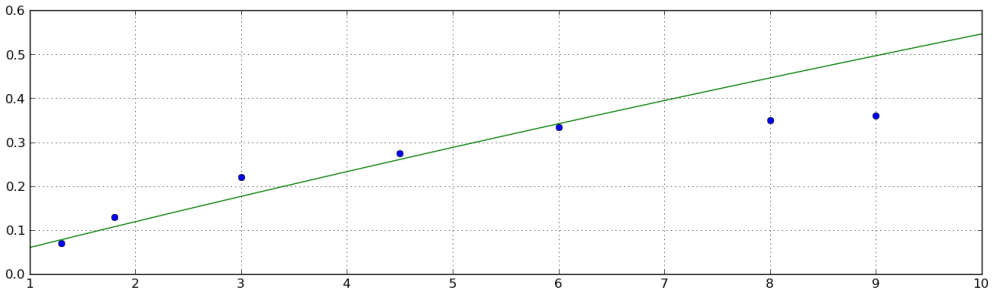


Figura 4: Modelo Michaelis-Menten de primer orden

(d) El modelo de Michaelis-Menten de segundo orden está dado por:

$$v = \frac{v_m[S]^2}{K_s^2 + [S]^2}$$

repita todos los incisos anteriores, pero ahora empleando este modelo.

Solución: Primero linealizamos el modelo, nuevamente tomando el recíproco:

$$\begin{aligned} v &= \frac{v_m[S]^2}{K_s^2 + [S]^2} \\ \frac{1}{v} &= \frac{K_s^2 + [S]^2}{v_m[S]^2} \\ &= \frac{K_s^2}{v_m}\frac{1}{[S]^2} + \frac{1}{v_m} \end{aligned}$$

que exhibe una relación lineal entre $\frac{1}{[S]^2}$ y $\frac{1}{v}$, con pendiente $\frac{K_s^2}{v_m}$ e intercepto $\frac{1}{v_m}$. Luego:

```
a,b = polyfit(1/S**2, 1/v, 1)
Ks = sqrt(a/b)
vm = 1/b

Smod = linspace(1,10)
vmod = vm*Smod**2/(Ks**2 + Smod**2)

plot(S,v,'o', Smod, vmod)
grid()
show()
```

con lo que obtenemos $v_m = 0.408, K_s = 2.812$; que lleva al modelo de segundo orden:

$$v = \frac{0.408[S]^2}{7.911 + [S]^2}$$

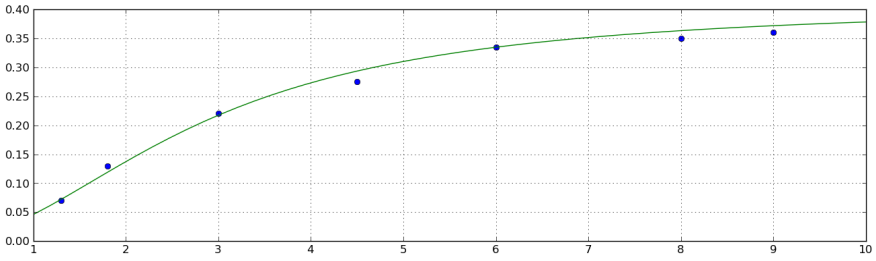


Figura 5: Modelo Michaelis-Menten de segundo orden

(e) ¿Qué modelo (primer o segundo orden) de Michaelis-Menten cree que es más apropiado para describir las enzimas de este problema? Explique.

Solución: Las gráficas obtenidas indican que el modelo Michaelis-Menten de segundo orden es el más apropiado.

2. En la naturaleza encontramos parábolas por todos lados. Un ejemplo es la trayectoria de los chorros de agua en una fuente:

Figura 6: Parábolas de agua

Experimentalmente recopilamos los siguientes datos:

x	0.35	0.71	1.07	1.42	1.78	2.14	2.50	2.85	3.21	3.57	3.92	4.28	4.64
y	-2.11	-0.69	0.18	0.75	2.46	2.42	2.24	2.07	1.52	0.85	-0.00	-1.28	-2.07

(a) Ingrese estos datos en arreglos de Python y grafique estos puntos. Claramente la relación entre las variables x, y no es lineal.

Solución:

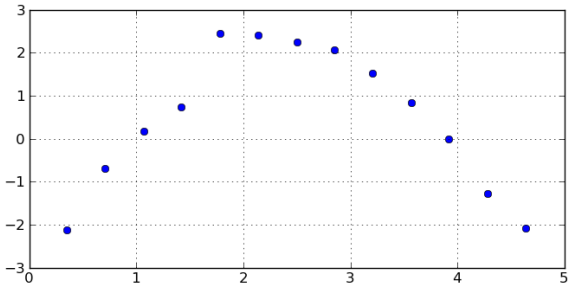


Figura 7: Datos

(b) Utilice el comando `polyfit` para encontrar un modelo cuadrático $f(x) = ax^2 + bx + c$ para estos datos (vea su documentación). Grafique ahora los puntos y el modelo.

Solución: A continuación se muestra parte de la documentación de `polyfit`:

```
In [22]: polyfit?
Type:           function
Definition:     polyfit(x, y, deg, rcond=None, full=False)
Docstring:
    Least squares polynomial fit.
    Fit a polynomial ‘‘p(x) = p[0] * x**deg + ... + p[deg]’’ of degree ‘deg‘
```

```
    to points ‘(x, y)‘. Returns a vector of coefficients ‘p‘ that minimises
    the squared error.
```

Finalmente:

```
In [24]: x
Out[24]:
array([ 0.35,  0.71,  1.07,  1.42,  1.78,  2.14,  2.5 ,  2.85,  3.21,
        3.57,  3.92,  4.28,  4.64])
In [25]: y
Out[25]:
array([-2.11, -0.69,  0.18,  0.75,  2.46,  2.42,  2.24,  2.07,  1.52,
        0.85, -0.  , -1.28, -2.07])
In [26]: polyfit(x,y,2)
Out[26]: array([-0.98113955,  4.81351875, -3.66180579])
In [27]: plot(x,y,'o')
In [28]: plot(x, -0.98*x**2 + 4.81*x - 3.66)
In [29]: grid()
```

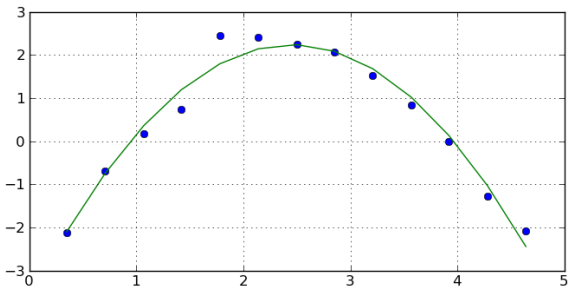


Figura 8: Datos y su modelo cuadrático

Métodos Numéricos
UVG-MM2010, MM2022

Laboratorio 11

Semana: del 12 al 16 de Abril

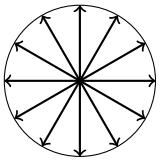
Instrucciones: En sus cursos de matemática se le ha recomendado que ataque los problemas desde varios enfoques: **analítico**, **gráfico**, y **numérico**. En este curso nos interesan particularmente los últimos dos enfoques. Responda las siguientes preguntas dejando constancia clara de su procedimiento; recuerde incluir **gráficas**, **código en Python**, y explicaciones que considere pertinentes.

1 Introducción

En el laboratorio anterior empezamos a tratar el problema de *mínimos cuadrados ordinarios*. En este lab trabajaremos con *mínimos cuadrados totales*[1] desde una perspectiva geométrica y algebraica (a diferencia de la tradicional perspectiva de optimización).

1.1 Un poco de magia vectorial

Considere vectores $\vec{x}_k$ en el círculo unitario como se muestran en la figura:



Sea A una matriz de 2×2, dado que los $\vec{x}_k \in \mathbb{R}^2$ tienen dimensión 2×1, tenemos entonces que el producto $A\vec{x}_k$ es también 2×1. Geométricamente, ¿qué le sucede a los vectores en el círculo unitario al multiplicarlos por la matriz A? Claramente ésto depende de dicha matriz y es lo que queremos explorar. Empecemos con una matriz sencilla:

$$A = \begin{bmatrix} \alpha & 0 \\ 0 & \beta \end{bmatrix}$$

Dado que $\vec{x}_k$ es un vector en el círculo unitario, podemos parametrizarlo en términos del ángulo θ, como

$$\vec{x}_k = \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix}$$

por lo que el producto $A\vec{x}_k$ está dado por:

$$\begin{bmatrix} \alpha & 0 \\ 0 & \beta \end{bmatrix} \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix} = \begin{bmatrix} \alpha\cos(\theta) \\ \beta\sin(\theta) \end{bmatrix}$$

si ahora eliminamos el parámetro θ tenemos entonces que

$$\begin{aligned} x(\theta) &= \alpha\cos(\theta) & \frac{x}{\alpha} &= \cos(\theta) & \left(\frac{x}{\alpha}\right)^2 &= \cos^2(\theta) \\ y(\theta) &= \beta\sin(\theta) & \frac{y}{\beta} &= \sin(\theta) & \left(\frac{y}{\beta}\right)^2 &= \sin^2(\theta) \end{aligned}$$

[1] también llamados mínimos cuadrados ortogonales.

Finalmente, al sumar reconocemos la ecuación cartesiana para una *elipse*:

$$\left(\frac{x}{\alpha}\right)^2 + \left(\frac{y}{\beta}\right)^2 = 1$$

Importante al multiplicar la matriz A con los vectores $\vec{x}_k$ en el *círculo unitario* obtenemos una *elipse*. Pero ésto fué para una matriz particular, en general ¿qué cree que pasa si consideramos otras matrices de 2×2? Por ejemplo, si

$$A = \begin{bmatrix} 2 & 7 \\ 6 & 10 \end{bmatrix}$$

¿será posible que otra vez obtengamos una elipse? Analíticamente es difícil ver el resultado de dicha operación, pero usando una computadora el trabajo es sencillo.

La siguiente función nos permitirá explorar los aspectos geométricos del producto $A\vec{x}_k$:

```
def geom_matr(A, n=12):
    """Explora la geometria detras del producto de la matriz 'A' con
    'n' vectores 'x_k' en el circulo unitario.
    Regresa arreglos 'x','y' originales (el circulo unitario); asi
    como arreglos 'x','y' transformados (el efecto de multiplicar por
    la matriz 'A')."""

    theta = linspace(0, 2*pi, n)
    X = vstack( (cos(theta), sin(theta)) )
    x_orig = X[0]
    y_orig = X[1]
    B = dot(A, X)
    x_trans = B[0]
    y_trans = B[1]
    return x_orig, y_orig, x_trans, y_trans
```

Dada una matriz A podemos llamar a esta función así (tomando sólo 12 vectores $\vec{x}_k$):

```
x_orig, y_orig, x_trans, y_trans = geom_matr(A)
```

o así (especificando el número de vectores $\vec{x}_k$):

```
x_orig, y_orig, x_trans, y_trans = geom_matr(A, 64)
```

Por ejemplo si tenemos un múltiplo de la matriz identidad, e.g. 3I, entonces (vea la documentación del comando eye):

```
A = 3*eye(2)
x_orig, y_orig, x_trans, y_trans = geom_matr(A,64)
```

Y graficamos:

```
scatter(x_orig, y_orig)
scatter(x_trans, y_trans, color='r')
axis('equal')
grid()
```

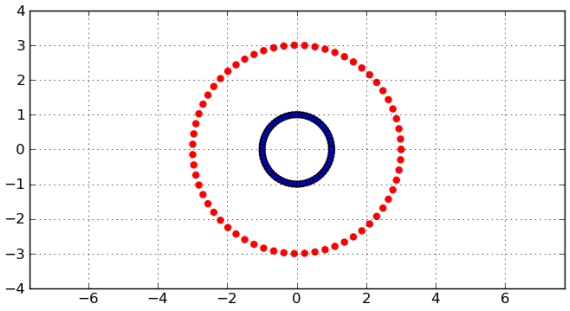


Figure 1: Efecto geométrico del producto $3I\vec{x}_k$: círculo de radio 3

1. Utilice la función **geom_matr** definida anteriormente para estudiar el efecto geométrico del producto de $\vec{x}_k$ con las siguientes matrices.

   (a) Una matriz de "estiramiento" (asigne usted valores particulares para α y β):

$$E = \begin{bmatrix} \alpha & 0 \\ 0 & \beta \end{bmatrix}$$

Solución: Tomando $\alpha = 2, \beta = 5$ tenemos,

```
In [14]: E = mat([ [2, 0], [0, 5] ])
In [15]: x_orig, y_orig, x_trans, y_trans = geom_matr(E,64)
In [16]: scatter(x_orig, y_orig)
In [17]: scatter(x_trans, y_trans, color='r')
In [18]: axis('equal')
In [19]: grid()
```

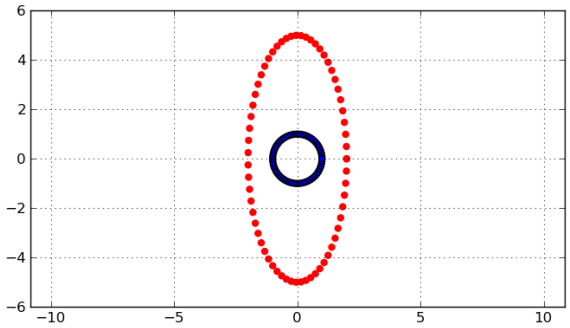


Figure 2: Efecto geométrico del producto $E\vec{x}_k$: elipse

   (b) Una matriz de rotación $R(\phi)$ (asigne usted un ángulo particular ϕ):

$$R = \begin{bmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{bmatrix}$$

Solución: Tomando $\phi = \pi/3$ tenemos,

```
In [23]: phi = pi/3
In [24]: R = mat([ [cos(phi), -sin(phi)], [sin(phi), cos(phi)] ])
In [15]: x_orig, y_orig, x_trans, y_trans = geom_matr(R)
In [16]: scatter(x_orig, y_orig)
In [17]: scatter(x_trans, y_trans, color='r')
In [18]: axis('equal')
In [19]: grid()
```

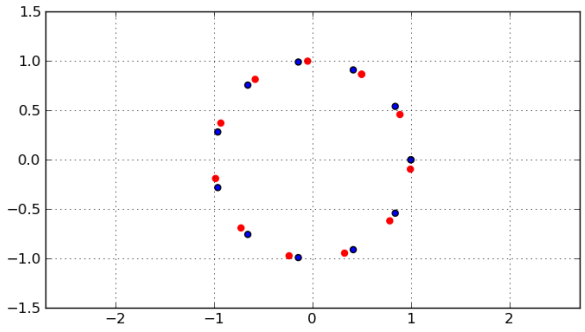


Figure 3: Efecto geométrico del producto $R\vec{x}_k$: círculo unitario rotado

(c) Una matriz que resulta del producto de R (una matriz de rotación) con E (una matriz de "estiramiento"), en símbolos $A = RE$. Tome las matrices que utilizó en los incisos anteriores.

Solución:

```
In [30]: R
Out[30]:
matrix([[ 0.5      , -0.8660254],
        [ 0.8660254,  0.5      ]])
In [31]: E
Out[31]:
matrix([[2, 0],
        [0, 5]])
In [32]: A = R*E; A
Out[32]:
matrix([[ 1.        , -4.33012702],
        [ 1.73205081,  2.5       ]])
In [33]: x_orig, y_orig, x_trans, y_trans = geom_matr(A,64)
In [34]: scatter(x_orig, y_orig)
In [35]: scatter(x_trans, y_trans, color='r')
In [36]: axis('equal')
In [37]: grid()
```

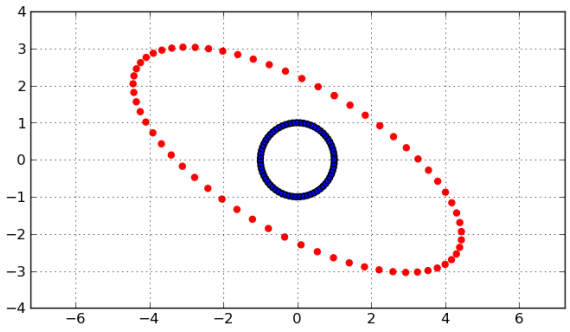


Figure 4: Efecto geométrico del producto $R\,E\vec{x}_k$: elipse rotada

(d) Una matriz arbitraria de 2×2, e.g.

$$A = \begin{bmatrix} 2 & 7 \\ 6 & 10 \end{bmatrix}$$

Solución:

```
In [38]: A = mat([ [2, 7], [6, 10] ])
In [39]: x_orig, y_orig, x_trans, y_trans = geom_matr(A,64)
In [40]: scatter(x_orig, y_orig)
In [41]: scatter(x_trans, y_trans, color='r')
In [42]: axis('equal')
In [43]: grid()
```

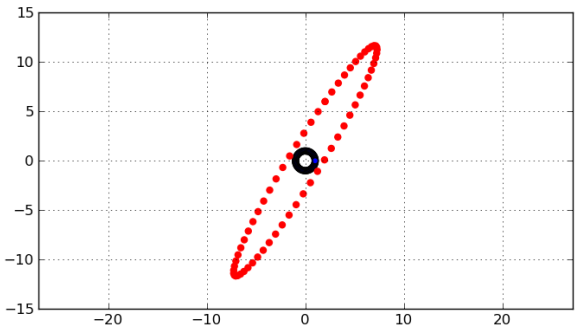


Figure 5: Efecto geométrico del producto $A\vec{x}_k$: elipse rotada

Conclusión El efecto de multiplicar vectores en el círculo unitario por una matriz A de 2×2 es, en general, una elipse rotada. De alguna manera podríamos entonces *descomponer* (factorizar) cualquier matriz A como el producto de matrices elementales, i.e. matriz de "estiramiento", matriz de rotación.

1.2 Descomposición de Valor Singular

De su trabajo en la sección anterior debería tener ahora la intuición que toda matriz A puede descomponerse (*factorizarse*) en un producto que involucre matrices de rotación y matrices de "estiramiento" (formalmente, estas son matrices diagonales). Precisamente ésta es la idea detrás de la Descomposición (factorización) de Valor Singular en la que una matriz A de $m \times n$ puede escribirse como el producto de matrices:

$$A = USV$$

donde U es de $m \times k$, V es de $k \times n$, y S es de $k \times k$ para $k = \min(m, n)$. En Python el comando es **svd** y se utiliza así (dada una matriz A):

```
U,s,V = svd(A, full_matrices=False)
```

Para ejemplificar consideremos los datos con los que trabajamos en el lab anterior:

x	y
0.0	0.000
0.1	0.078
0.2	0.138
0.3	0.192
0.4	0.244

Los ingresamos en Python:

```
Data = [(0.0, 0.000), (0.1, 0.078), (0.2, 0.138), (0.3, 0.192), (0.4, 0.244)]
x = array([k[0] for k in Data])
y = array([k[1] for k in Data])
xN = x - average(x)
yN = y - average(y)
```

Los graficamos:

```
scatter(x,y)
scatter(xN,yN, color='r')
axis('equal')
grid()
```

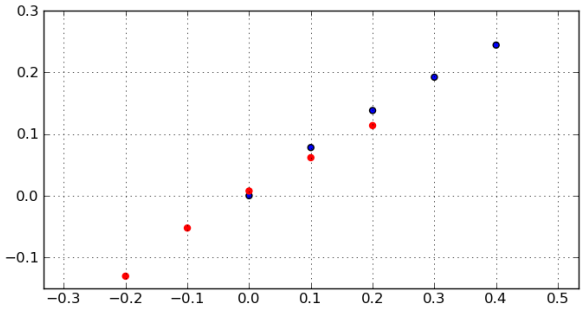


Figure 6: Data

Importante Restamos la media de los datos para tener al origen $(0,0)$ como referencia.

Construimos ahora la matriz A cuyas columnas están dadas por $x-\bar{x}$, $y-\bar{y}$, y obtenemos su descomposición de valor singular (*SVD*, por sus siglas en inglés):

```
A = vstack( (xN, yN) ).T
U,s,V = svd(A, full_matrices=False)
```

La clave está en que ahora las matrices U, S, V tienen mucha información geométrica relacionada a nuestra data.

Antes de seguir, sería bueno comprobar que la factorización anterior funciona, i.e. que en efecto el producto USV es igual a la matriz A. En Python (vea la documentación del comando `mat` y note cómo facilita el multiplicar matrices):

```
In [70]: U,s,V = svd(A, full_matrices=False)
In [71]: U.shape; s.shape; V.shape
Out[71]: (5, 2)
Out[71]: (2,)
Out[71]: (2, 2)
In [72]: U = mat(U); S = diag(s); V = mat(V)
In [73]: allclose( A, U*S*V )
Out[73]: True
```

Importante Los componentes de la matriz diagonal S son los *valores singulares* de A. Los valores singulares son una generalización del concepto de *eigenvalor* para matrices de $m \times n$.

Recuerde de álgebra lineal que una *matriz unitaria* es una matriz que cumple con:

$$AA^T = A^TA = I$$

Veamos si la matriz V cumple con ser una matriz unitaria:

```
In [282]: V
Out[282]:
matrix([[ 0.85628904,  0.51649693],
        [ 0.51649693, -0.85628904]])
In [283]: V * V.T
Out[283]:
matrix([[ 1.,  0.],
        [ 0.,  1.]])
In [284]: V.T * V
Out[284]:
matrix([[ 1.,  0.],
        [ 0.,  1.]])
```

Sí, en efecto, V es una matriz unitaria. La importancia de esta observación es que si A es una matriz unitaria de $n \times n$ entonces tenemos que los vectores columna de A forman una *base ortonormal* para $\mathbb{R}^n$.

Recordemos algunos términos de álgebra lineal (limitémonos por el momento al espacio vectorial $\mathbb{R}^2$). ¿Cuál es el *espacio generado* de un solo vector $\vec{x} \in \mathbb{R}^2$, distinto de cero? ¿Cuál es el *espacio generado* de dos vectores no colineales en $\mathbb{R}^2$? Denotaremos el espacio generado de un vector $\vec{x}$ con $\text{span}(\vec{x})$.

Grafiquemos entonces el espacio generado por cada uno de los vectores columna de la matriz V.

```
U,s,V = svd(A, full_matrices=False)
v0 = V[:,0]
v1 = V[:,1]

t = linspace(-0.5, 0.5)
plot( t*float(v0[0]), t*float(v0[1]) )
plot( t*float(v1[0]), t*float(v1[1]) )
axis('equal')
grid()
```

Coloquemos encima de esta gráfica los datos ya centrados:

```
scatter(xN,yN)
```

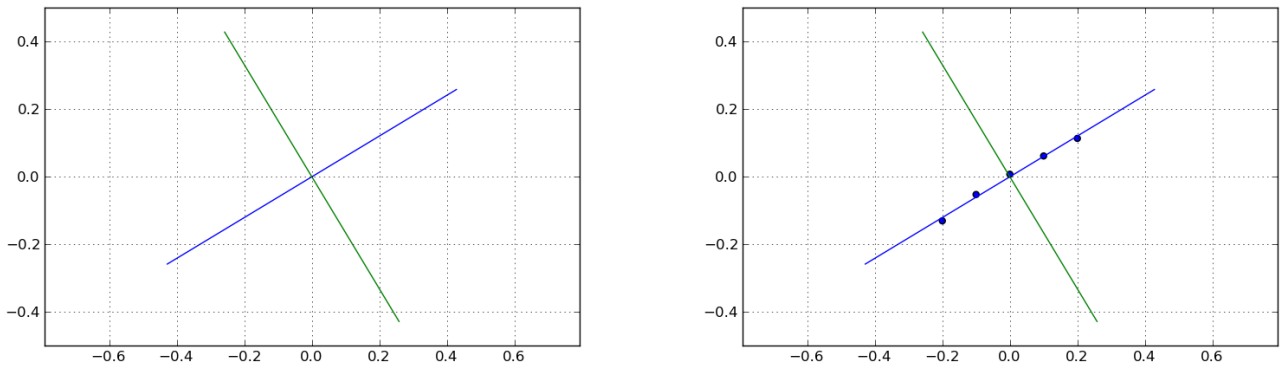


Figure 7: Datos y span($\vec{v}_0$), span($\vec{v}_1$)

¡Como por arte de magia nuestros datos aparecen alineados con el span($\vec{v}_0$)! [2] En otras palabras, esa recta es una recta de mejor ajuste para nuestros datos.

2 Preguntas

2. Considere los vectores columna $\vec{v}_0, \vec{v}_1$ de la matriz V en el ejemplo anterior. Encuentre el producto punto $\vec{v}_0 \cdot \vec{v}_1$ y dele una interpretación geométrica.

Solución: Como ya sabemos que la matriz V es una *matriz unitaria*, tenemos entonces que los vectores columna de V forman una *base ortonormal*, i.e. esperamos que $v_0 \perp v_1$. En Python:

```
In [83]: U,s,V = svd(A, full_matrices=False)
In [84]: V
Out[84]:
array([[ 0.85628904,  0.51649693],
       [ 0.51649693, -0.85628904]])
```

[2] el espacio generado del primer vector columna de la matriz V.

```
In [85]: v0 = V[:,0]; v0
Out[85]: array([ 0.85628904,  0.51649693])
In [86]: v1 = V[:,1]; v1
Out[86]: array([ 0.51649693, -0.85628904])
In [87]: dot(v0, v1)
Out[87]: 0.0
```

3. En el ejemplo anterior, ¿cuál es la pendiente de la recta generada por el span($\vec{v}_0$)?

Solución: Tomamos $\vec{v}_0$ y recordamos que $m = \Delta y/\Delta x$:

```
In [88]: v0
Out[88]: array([ 0.85628904,  0.51649693])
In [89]: v0[1] / v0[0]
Out[89]: 0.60318059456414919
```

por lo que $m \approx 0.603$.

4. Encuentre la ecuación de la recta de mejor ajuste para los datos centrados (x_N, y_N) y la ecuación de la recta para los datos originales (x, y).

Solución: En los datos centrados tenemos que la recta de mejor ajuste pasa por el origen $(0, 0)$, además en el inciso anterior encontramos que su pendiente $m \approx 0.603$:

$$y_N = 0.603x$$

Ahora bien, para los datos originales la recta de mejor ajuste tendrá la misma pendiente pero necesitamos encontrar uno de sus puntos. No debería ser muy difícil convercerse que $(\bar{x}, \bar{y})$ es un punto sobre esta recta:

$$\begin{aligned} y - 0.1304 &= 0.603(x - 0.2) \\ y &= 0.603(x - 0.2) + 0.1304 \\ y &= 0.603x + 0.0098 \end{aligned}$$

En Python:

```
In [94]: scatter(xN, yN)
In [95]: plot(xN, 0.603*xN)
In [96]: axis('equal')
In [97]: grid()

In [104]: scatter(x, y)
In [105]: plot(x, 0.603*x + 0.0098)
In [106]: axis('equal')
In [107]: grid()
```

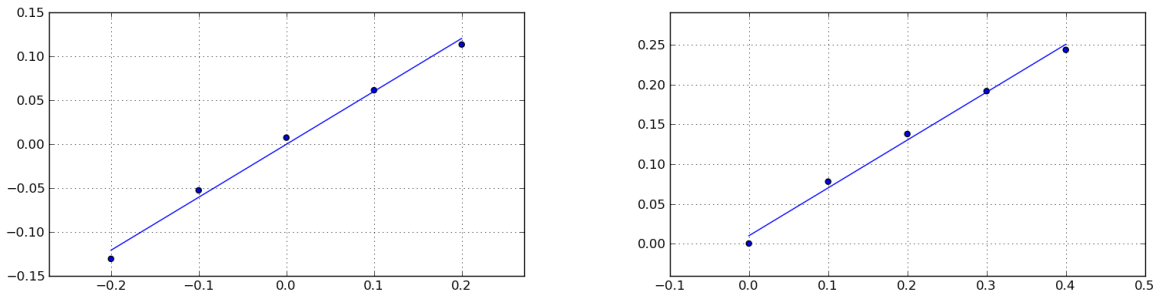


Figure 8: Data y sus rectas de mejor ajuste

5. Se han recopilado las siguientes 7 muestras de peso (en libras) y estatura (en centímetros) de estudiantes varones.

cm	172	180	184	178	172	181	186
lb	136	196	169	143	185	143	176

Claramente, podemos tener errores en ambas variables; por lo que decidimos aplicar *mínimos cuadrados ortogonales*. Encuentre dicha recta de mejor ajuste empleando la *SVD*.

Solución:

Encapsulemos lo que hemos hecho en este lab en un par de funciones: una que regrese la recta de mejor ajuste usando la SVD (`recta_svd`) y otra que grafique los datos (`grafica_svd`):

```
def recta_svd(x,y):
    """Toma datos 'x', 'y'; regresa parametros para
    la recta de mejor ajuste empleando la SVD.

    'm': pendiente
    'b': intercepto
    'v0': primera columna de V
    'v1': segunda columna de V"""

    xN = x - average(x)
    yN = y - average(y)
    A = vstack( (xN, yN) ).T
    U,s,V = svd(A, full_matrices=False)
    v0 = V[:, 0]
    v1 = V[:, 1]
    m = v0[1] / v0[0]
    b = average(y) - m*average(x)
    return m,b,v0,v1,xN,yN

########################################
def grafica_svd(x,y):
    """Grafica los datos originales y los datos
    centrados en (0,0).
```

```
    Llama a la funcion 'recta_svd', por lo que tiene
    que estar definida."""

    m,b,v0,v1,xN,yN = recta_svd(x,y)
    subplot(1,2,1)
    scatter(x,y)
    axis('equal')
    grid()
    subplot(1,2,2)
    scatter(xN,yN, color='r')
    axis('equal')
    axis_tmp = axis()
    # span(v0), span(v1)
    tmin = min(xN)/v0[0]
    tmax = max(xN)/v0[0]
    t = linspace(tmin, tmax)
    plot( t*v0[0], t*v0[1] )
    plot( t*v1[0], t*v1[1] )
    axis(axis_tmp)
    grid()
```

Asegúrese de entender las funciones anteriores. Ingresamos ahora los datos y ejecutamos estas funciones:

```
In [442]: x = array([172, 180, 184, 178, 172, 181, 186])
In [443]: y = array([136, 196, 169, 143, 185, 143, 176])
In [444]: m,b,v0,v1,xN,yN = recta_svd(x,y)
In [445]: m,b
Out[445]: (18.527347049336583, -3152.3951218312486)
In [446]: grafica_svd(x,y)
```

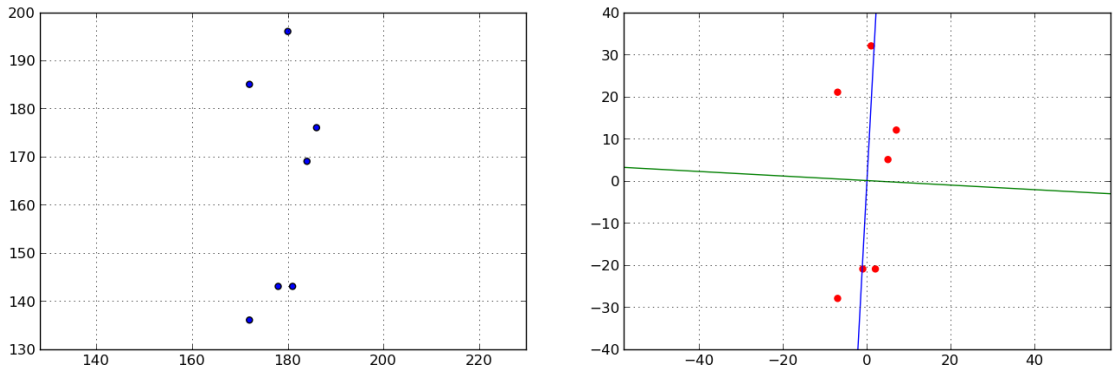


Figure 9: Datos y span($\vec{v}_0$), span($\vec{v}_1$)

Métodos Numéricos **Guía 12**
UVG-MM2010, MM2022

Instrucciones: En sus cursos de matemática se le ha recomendado que ataque los problemas desde varios enfoques: **analítico**, **gráfico**, y **numérico**. En este curso nos interesan particularmente los últimos dos enfoques. Responda las siguientes preguntas dejando constancia clara de su procedimiento; recuerde incluir **gráficas**, **código en Python**, y explicaciones que considere pertinentes.

Introducción

En este laboratorio exploraremos algunas de las ideas de derivación numérica, recuerde que en Cálculo 1 definimos la derivada de una función $f(x)$ como el límite

$$\frac{df}{dx} = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h}$$

Desde el punto de vista numérico tenemos que prescindir del concepto de límite. En otras palabras, tenemos la siguiente aproximación:

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

a este tipo de expresiones le llamamos *diferencias finitas*. La clave para analizar diferencias finitas está en utilizar una de las variantes del *Teorema de Taylor*:

$$f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2!}h^2 + \frac{f'''(x)}{3!}h^3 + \cdots$$

Noten que manipulando esta expresión podemos redescubrir la vieja definición de derivada dada en Cálculo 1:

$$\begin{aligned}
f(x+h) &= f(x) + f'(x)h + \frac{f''(x)}{2!}h^2 + \frac{f'''(x)}{3!}h^3 + \cdots \\
f(x+h) - f(x) &= f'(x)h + \frac{f''(x)}{2!}h^2 + \frac{f'''(x)}{3!}h^3 + \cdots \\
\frac{f(x+h) - f(x)}{h} &= f'(x) + \frac{f''(x)}{2!}h + \frac{f'''(x)}{3!}h^2 + \cdots \\
\frac{f(x+h) - f(x)}{h} &= f'(x) + O(h)
\end{aligned}$$

donde el término $O(h)$ nos sirve para indicar que esta diferencia finita tiene un *error lineal* con respecto a h. El objetivo es tratar de encontrar una diferencia finita para $f'(x)$ con un error más pequeño. Nuevamente, con el Teorema de Taylor:

$$\begin{aligned}
f(x+h) &= f(x) + f'(x)h + \frac{f''(x)}{2!}h^2 + \frac{f'''(x)}{3!}h^3 + \cdots \\
f(x-h) &= f(x) - f'(x)h + \frac{f''(x)}{2!}h^2 - \frac{f'''(x)}{3!}h^3 + \cdots
\end{aligned}$$

restando estas dos expresiones obtenemos

$$\begin{aligned}
f(x+h) - f(x-h) &= 2f'(x)h + 2\frac{f'''(x)}{3!}h^3 + \cdots \\
\frac{f(x+h) - f(x-h)}{2h} &= f'(x) + \frac{f'''(x)}{3!}h^2 + \cdots \\
\frac{f(x+h) - f(x-h)}{2h} &= f'(x) + O(h^2)
\end{aligned}$$

tenemos ahora otra diferencia finita para $f'(x)$, pero ésta tiene un error cuadrático $O(h^2)$. Recuerde que si $0 < h < 1$, entonces $h^2 < h$; por lo que preferimos trabajar con esta último diferencia finita. Nuevamente nos preguntamos, ¿será posible encontrar una diferencia finita para $f'(x)$ con un error más pequeño? Nuevamente la clave está en el Teorema de Taylor. Retomemos las expresiones anteriores:

$$\begin{aligned}
f(x+h) - f(x-h) &= 2f'(x)h + 2\frac{f'''(x)}{3!}h^3 + 2\frac{f^{v}(x)}{5!}h^5 + \cdots \\
\frac{f(x+h) - f(x-h)}{2h} &= f'(x) + \frac{f'''(x)}{3!}h^2 + \frac{f^{v}(x)}{5!}h^4 + \cdots
\end{aligned}$$

denotemos el lado izquierdo de esta ecuación con $\phi(h)$:

$$\phi(h) = f'(x) + \frac{f'''(x)}{3!}h^2 + \frac{f^{v}(x)}{5!}h^4 + \cdots$$

evaluemos ahora $\phi(h/2)$

$$\begin{aligned}
\phi\left(\frac{h}{2}\right) &= f'(x) + \frac{f'''(x)}{3!}\left(\frac{h}{2}\right)^2 + \frac{f^{v}(x)}{5!}\left(\frac{h}{2}\right)^4 + \cdots \\
&= f'(x) + \frac{1}{4}\frac{f'''(x)}{3!}h^2 + \frac{1}{16}\frac{f^{v}(x)}{5!}h^4 + \cdots
\end{aligned}$$

finalmente, consideramos la expresión $4\phi(h/2) - \phi(h)$:

$$\begin{aligned}
4\phi\left(\frac{h}{2}\right) - \phi(h) &= 3f'(x) - \frac{3}{4}\frac{f^{v}(x)}{5!}h^4 + \cdots \\
\frac{4}{3}\phi\left(\frac{h}{2}\right) - \frac{1}{3}\phi(h) &= f'(x) - \frac{1}{4}\frac{f^{v}(x)}{5!}h^4 + \cdots \\
\frac{4}{3}\phi\left(\frac{h}{2}\right) - \frac{1}{3}\phi(h) &= f'(x) + O(h^4)
\end{aligned}$$

encontramos entonces una diferencia finita para $f'(x)$ de orden 4. Mucho mejor que las anteriores. Resumamos estos resultados en una tabla:

Diferencia finita	Expresión	Error
Hacia adelante	$f'(x) \approx \frac{f(x+h)-f(x)}{h}$	$O(h)$
Central	$f'(x) \approx \frac{f(x+h)-f(x-h)}{2h}$	$O(h^2)$
Extrapolación de Richardson	$f'(x) \approx \frac{4}{3}\phi\left(\frac{h}{2}\right) - \frac{1}{3}\phi(h)$	$O(h^4)$

Las diferencias finitas son fáciles de implementar en la computadora. Veamos por ejemplo el código para la diferencia finita hacia adelante:

```
def dy_forward(y, h):
    N = len(y)
    dy = zeros(N)
    for k in range(N - 1):
        dy[k] = (y[k+1] - y[k])/h
    return dy
```

En la terminal, llamamos a esta función así:

```
In [27]: a,b,h = 0, 2*pi, 0.1
In [28]: x = arange(a,b,h)
In [29]: y = sin(x)
In [31]: dy = dy_forward(y,h)
In [32]: scatter(x,y)
In [33]: scatter(x,dy,color='r')
In [34]: grid()
```

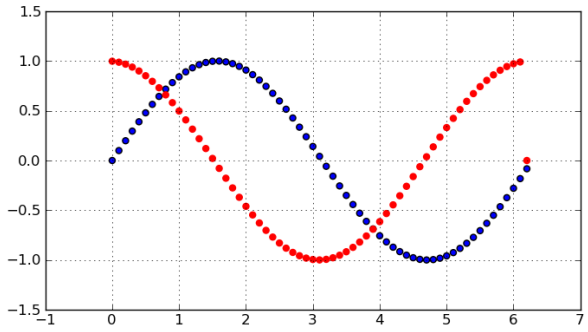


Figura 1: Gráficas de $y = \sin(x)$ y su derivada numérica

Importante: noten que el último punto de la derivada numérica es cero, ¿por qué?

Preguntas

1. Escriba una función para aproximar numéricamente $f'(x)$ empleando la diferencia finita central. Utilice este código para elaborar gráficas de $y = \sin(x)$ y su derivada numérica como en el ejemplo de la introducción.

Solución: Ver la solución del siguiente problema.

2. Escriba una función para aproximar numéricamente $f'(x)$ empleando la extrapolación de Richardson. Utilice este código para elaborar gráficas de $y = \sin(x)$ y su derivada numérica como en el ejemplo de la introducción.

Solución: En Python:

```
from pylab import *

def dy_forward(y, h):
    N = len(y)
    dy = zeros(N)
    for k in range(N - 1):
        dy[k] = (y[k+1] - y[k])/h
    return dy

def dy_central(y, h):
    N = len(y)
    dy = zeros(N)
    for k in range(1, N - 1):
        dy[k] = (y[k+1] - y[k-1])/(2*h)
    return dy

def dy_richardson(y, h):
    N = len(y)
    dy = zeros(N)
    for k in range(2, N - 2):
        dy[k] = 4.0/3*((y[k+1] - y[k-1])/(2*h)) - \
                1.0/3*((y[k+2] - y[k-2])/(4*h))
    return dy

########################################
a,b,h = 0, 2*pi, 0.1
x = arange(a,b,h)
y = sin(x)
dy1 = dy_forward(y, h)
dy2 = dy_central(y, h)
dy3 = dy_richardson(y, h)

P = [(dy1, 'Dif. Forward'), (dy2, 'Dif. Central'), (dy3, 'Richardson')]

for k,Dy in enumerate(P):
    dy, tit = Dy
```

```
    subplot(3,1,k+1)
    scatter(x,y)
    scatter(x,dy, color='r')
    axis('equal')
    grid()
    title(tit)

show()
```

al correr este script obtenemos la siguiente gráfica:

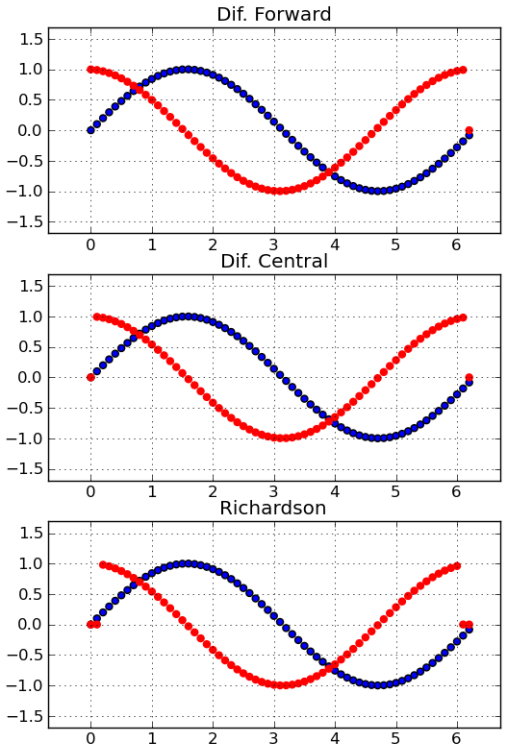


Figura 2: Gráficas de $y = \sin(x)$ y sus derivadas numéricas

3. Empleando las variantes del Teorema de Taylor dadas en la introducción, encuentre una diferencia finita para la segunda derivada $f''(x)$. Determine el orden del error, $O(h^n)$. Finalmente, escriba una función para aproximar numéricamente $f''(x)$ y utilice este código para elaborar gráficas de $y = \sin(x)$ y su segunda derivada numérica.

Solución: Nuevamente, con el Teorema de Taylor:

$$f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2!}h^2 + \frac{f'''(x)}{3!}h^3 + \cdots$$
$$f(x-h) = f(x) - f'(x)h + \frac{f''(x)}{2!}h^2 - \frac{f'''(x)}{3!}h^3 + \cdots$$

sumando estas dos expresiones obtenemos

$$f(x+h) + f(x-h) = 2f(x) + 2\frac{f''(x)}{2!}h^2 + 2\frac{f^{iv}(x)}{4!}h^4 + \cdots$$
$$f(x+h) - 2f(x) + f(x-h) = 2\frac{f''(x)}{2!}h^2 + 2\frac{f^{iv}(x)}{4!}h^4 + \cdots$$
$$\frac{f(x+h) - 2f(x) + f(x-h)}{h^2} = f''(x) + 2\frac{f^{iv}(x)}{4!}h^2 + \cdots$$
$$\frac{f(x+h) - 2f(x) + f(x-h)}{h^2} = f''(x) + O(h^2)$$

En Python:

```
def d2y_diff(y, h):
    N = len(y)
    d2y = zeros(N)
    for k in range(1, N - 1):
        d2y[k] = (y[k+1] - 2*y[k] + y[k-1])/(h**2)
    return d2y
```

que puede ejecutarse así:

```
a,b,h = 0, 2*pi, 0.1
x = arange(a,b,h)
y = sin(x)
d2y = d2y_diff(y, h)

scatter(x,y)
scatter(x,d2y, color='r')
axis('equal')
grid()
title('Segunda derivada')
```

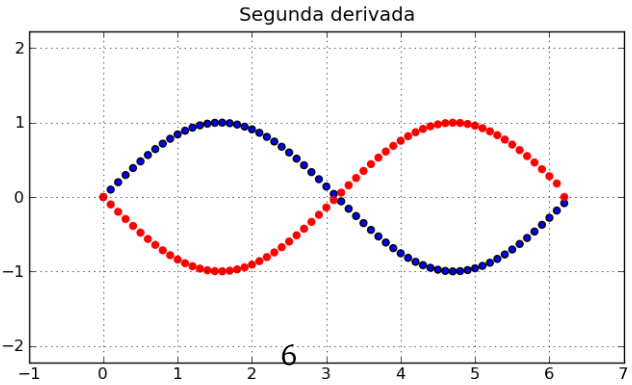


Figura 3: Gráficas de $y = \sin(x)$ y su segunda derivada numérica

4. Al trabajar con diferencias finitas asumimos que contamos con datos en una *partición regular*, i.e. h es constante. No siempre es éste el caso. Cuando tenemos datos con h variable podemos primero encontrar el *trazador cúbico* $s(x)$ para dichos datos y luego calcular $s'(x)$. Considere el siguiente escenario:

   El Derby de Kentucky[1] de 1995 lo ganó un caballo llamado *Thunder Gulch* en un tiempo de $2:01\frac{1}{5}$ (2 minutos, $1\frac{1}{5}$ segundos). La carrera es de $1\frac{1}{4}$ millas. Los tiempos en los postes a $1/4$ de milla, $1/2$ milla, y una milla fueron $22\frac{2}{5}$, $45\frac{4}{5}$, y $1:35\frac{3}{5}$.

   (a) Use estos puntos, así como el punto inicial $(0,0)$, para construir un trazador cúbico. Grafique.

   Solución: En Python:

```
import scipy.interpolate as si

# Datos en (millas, segundos)
P = [(0,0), (0.25, 22.4), (0.5, 45.8), (1, 95.6), (1.25, 121.2)]
T = array([k[1] for k in P]) # tiempo
D = array([k[0] for k in P]) # distancia

tck = si.splrep(T,D,s=0)
t = linspace(min(T), max(T))
d = si.splev(t, tck)

scatter(T,D)
plot(t,d)
xlabel('Tiempo [seg]')
ylabel('Distancia [millas]')
grid()
```

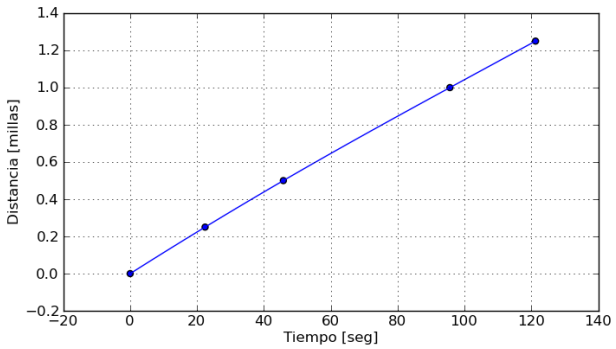


Figura 4: Distancia vs. Tiempo para *Thunder Gulch*

   (b) Use el trazador anterior para predecir el tiempo en el poste a $3/4$ de milla y compárelo con el tiempo medido en la carrera que fue de $1:10\frac{1}{5}$.

[1] similar a las carreras de caballos de Santa Lucía Cotzumalguapa, Escuintla.

Solución: Debemos notar que este problema pide encontrar el tiempo en función de la distancia $t(d)$; ésto es complementario al escenario usual en el que damos la distancia en función del tiempo $d(t)$. Encontremos entonces otro trazador invirtiendo las variables dependiente e independiente:

```
In [152]: tck_inv = si.splrep(D,T,s=0)
In [153]: si.splev(3.0/4, tck_inv)
Out[153]: 70.314285714285717
```

Ahora bien, el tiempo en la carrera fue de $1:10\frac{1}{5} = 70.2$ seg, por lo que el *spline* nos ha dado una buena aproximación.

(c) Encuentre una aproximación para la rapidez de *Thunder Gulch* al alcanzar la $1/2$ milla. Investigue en la referencia: `http://docs.scipy.org/doc/scipy/reference/interpolate.html`

Solución: Ya sabemos que el tiempo en $1/2$ milla fue de 45.8 segundos. Si tenemos un modelo para distancia en función del tiempo $d(t)$, tenemos entonces que la rapidez está dada por $r(t) = d'(t)$, i.e. necesitamos encontrar la derivada de nuestro trazador. En Python:

```
In [174]: si.splev(45.8, tck, der=1)
Out[174]: 0.010434809904837878
```

Por lo que la rapidez del caballo a $1/2$ milla fue de aproximadamente 0.01 millas por segundo. La clave está en el argumento `der=1` al llamar a la función `splev`.

Métodos Numéricos
UVG-MM2010, MM2022

Guía 13

Instrucciones: En sus cursos de matemática se le ha recomendado que ataque los problemas desde varios enfoques: **analítico**, **gráfico**, y **numérico**. En este curso nos interesan particularmente los últimos dos enfoques. Responda las siguientes preguntas dejando constancia clara de su procedimiento; recuerde incluir **gráficas**, **código en Python**, y explicaciones que considere pertinentes.

Introducción

En sus cursos de Cálculo aprendió técnicas analíticas para resolver integrales de la forma:

$$\int_a^b f(x)\,dx$$

donde la técnica a utilizar dependía de la función $f(x)$, e.g. sustitución simple, sustitución trigonométrica, integración por partes, etc.

En la contraparte numérica nuevamente tenemos puntos que representan una función:

x	x_0	x_1	$\cdots$	x_{N-1}
y	y_0	y_1	$\cdots$	y_{N-1}

Si $\Delta x = x_k - x_{k-1}$ es constante, tenemos entonces una *partición regular* del intervalo $[x_0, x_{N-1}]$ y la simple idea de considerar *trazadores lineales* nos lleva al *método del trapecio*. Si consideramos *trazadores cuadráticos* encontramos un equivalente al *método de Simpson*. Por otro lado, si Δx es variable se recomiendo encontrar el trazador cúbico y luego integrar dicho trazador.

Consideremos por ejemplo la siguiente integral:

$$I = \int_1^3 (\sin(2x) + x)\ dx$$

empleando el método del trapecio podemos encontrar una aproximación numérica. Geométricamente, la siguiente gráfica describe esta situación. En Python:

```
from pylab import *

f = lambda x: sin(2*x) + x
x = linspace(0,4)
y = f(x)

n = 4                               # numero de trapecios
xi = linspace(1,3, n+1)
yi = f(xi)
xi = hstack( (xi[0], xi, xi[-1]) )  # poligonos
yi = hstack( (0, yi, 0) )           # correctos

```

```
plot(x,y)
plot(xi, yi, 'r')
stem(xi, yi, 'r')
fill(xi, yi, 'r', alpha=0.1)
grid()
```

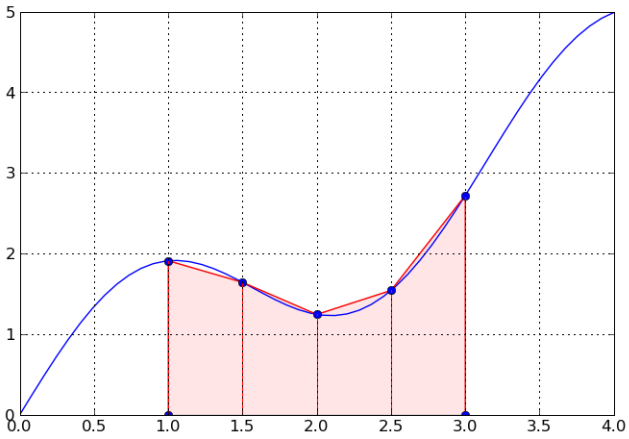


Figura 1: Aproximación a $\int_1^3 (\sin(2x) + x)\, dx$

Nos aprovecharemos de las implementaciones que trae `Scipy`, para más referencias vea:
`http://docs.scipy.org/doc/scipy/reference/tutorial/integrate.html`
`http://docs.scipy.org/doc/scipy/reference/integrate.html`

Regresemos a nuestro ejemplo. Esta integral es sencilla y puede resolverse empleando técnicas analíticas:

$$\begin{aligned} I &= \int_1^3 (\sin(2x) + x)\ dx \\ &= 4 + \frac{1}{2}\cos(2) - \frac{1}{2}\cos(6) \\ &\approx 3.3118 \end{aligned}$$

En Python ilustramos simultáneamente las funciones `trapz`, `simps`, `quad`:

```
In [216]: import scipy.integrate as sinteg
In [217]: f = lambda x: sin(2*x) + x
In [218]: n = 4
In [219]: xi = linspace(1,3, n+1)
In [220]: yi = f(xi)
In [221]: sinteg.trapz(yi, xi)
Out[221]: 3.3701671012010892
In [222]: sinteg.simps(yi, xi)
Out[222]: 3.3075099785996356
In [223]: sinteg.quad(f, 1, 3)
Out[223]: (3.3118414384012458, 3.6768826188207603e-14)
```

Preguntas

1. La *fórmula de Debye* para la *capacidad calorífica* C_V está dada por:

$$\frac{C_V}{9Nk} = \left(\frac{T}{T_D}\right)^3 \int_0^{T_D/T} \frac{x^4 e^x}{(e^x - 1)^2}\, dx$$

donde los términos de la ecuación son:

- N, número de partículas del sólido
- k, constante de Boltzmann
- T, temperatura absoluta
- T_D, temperatura de Debye

Encuentre una fórmula equivalente en el caso extremo $T = T_D$.

Solución: En $T = T_D$ tenemos que la fórmula de Debye es:

$$\begin{aligned}\frac{C_V}{9Nk} &= \left(\frac{T_D}{T_D}\right)^3 \int_0^{T_D/T_D} \frac{x^4 e^x}{(e^x - 1)^2}\, dx \\ &= \int_0^1 \frac{x^4 e^x}{(e^x - 1)^2}\, dx\end{aligned}$$

Notamos entonces que la función se indefine en $x = 0$ (¿por qué?), por lo que conviene elaborar una gráfica para tener una idea más clara de su comportamiento. En Python:

```
In [242]: f = lambda x: (x**4)*exp(x)/(exp(x) - 1)**2
In [243]: x = linspace(0,1)
In [244]: plot(x, f(x))
In [245]: grid()
```

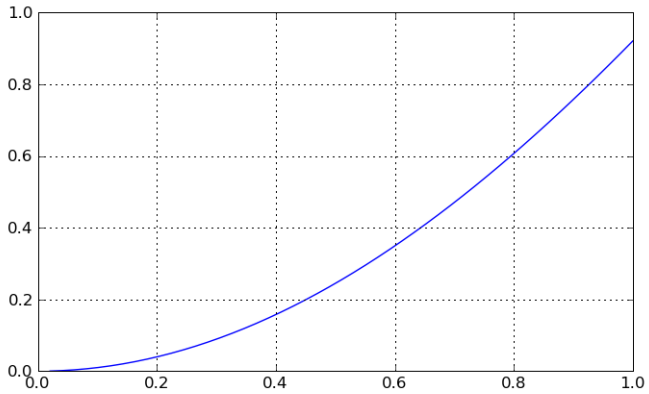


Figura 2: $\frac{x^4 e^x}{(e^x-1)^2}$

Dado que la gráfica no presenta irregularidades procedemos a encontrar la integral.

```
In [250]: import scipy.integrate as sinteg
In [251]: f = lambda x: (x**4)*exp(x)/(exp(x) - 1)**2
In [252]: sinteg.quad(f, 0, 1)
Out[252]: (0.31724404523442651, 3.5221164344448498e-15)
```

Por lo que la fórmula de Debye para $T = T_D$ es:

$$\frac{C_V}{9Nk} = \int_0^1 \frac{x^4 e^x}{(e^x - 1)^2}\, dx$$
$$\frac{C_V}{9Nk} \approx 0.3172$$

2. Considere la función $f(x) = e^{-\pi x^2}$.

   (a) Grafique $f(x)$ en el intervalo $[-2, 2]$.

Solución: En Python:

```
In [255]: f = lambda x: exp(-pi*x**2)
In [256]: x = linspace(-2,2)
In [257]: plot(x, f(x))
In [258]: grid()
In [259]: axis('equal')
```

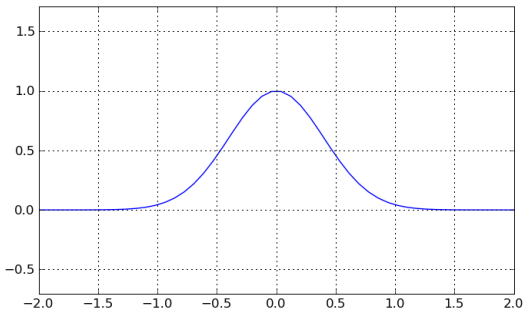


Figura 3: $f(x) = e^{-\pi x^2}$

   (b) Trate de resolver analíticamente la integral indefinida:

$$\int_{-\infty}^{\infty} e^{-\pi x^2}\, dx$$

Nota: esta integral sí puede resolverse, pero el procedimiento involucra integrales en varias variables y cambio de variables a coordenadas polares. Como éste es un curso de métodos numéricos no será penalizado si no puede contestar este inciso.

Solución: Considere:

$$I = \int_{-\infty}^{\infty} e^{-\pi x^2}\, dx$$

es claro que podemos cambiar variables, sin cambiar el valor de I,

$$I = \int_{-\infty}^{\infty} e^{-\pi y^2}\, dy$$

por lo que

$$\begin{aligned} I^2 &= \left(\int_{-\infty}^{\infty} e^{-\pi x^2}\, dx\right)\left(\int_{-\infty}^{\infty} e^{-\pi y^2}\, dy\right) \\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-\pi x^2} e^{-\pi y^2}\, dx\, dy \\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-\pi(x^2+y^2)}\, dx\, dy \end{aligned}$$

donde $x^2 + y^2$ sugiere considerar coordenadas polares; una nota importante es que el diferencial $dA = dx\, dy$ es $r\, dr\, d\theta$ en polares,

$$\begin{aligned} I^2 &= \int_0^{2\pi}\int_0^{\infty} e^{-\pi r^2} r\, dr\, d\theta \\ &= \int_0^{2\pi} \frac{1}{2\pi}\, d\theta \\ &= 1 \end{aligned}$$

finalmente,

$$\int_{-\infty}^{\infty} e^{-\pi x^2}\, dx = 1$$

(c) Numéricamente también tenemos problemas, e.g. ¿cómo resolvemos integrales impropias en una computadora? Una técnica usual es *plantear sustituciones*; por ejemplo, si $x = \tan(\theta)$ entonces $\theta = \arctan(x)$. Ahora bien, si $x \to -\infty$ entonces $\theta \to -\pi/2$, similarmente para $x \to \infty$. Empleando esta sustitución encuentre una integral equivalente.

Solución:

$$I = \int_{-\infty}^{\infty} e^{-\pi x^2}\, dx$$

con $x = \tan(\theta)$, tenemos $dx = \sec^2(\theta)\, d\theta$:

$$I = \int_{-\pi/2}^{\pi/2} e^{-\pi \tan^2(\theta)} \sec^2(\theta)\, d\theta$$

(d) Resuelva la integral del inciso anterior empleando las funciones `trapz`, `simps`, `quad` ilustradas en la introducción.

Solución: En Python:

```
In [264]: import scipy.integrate as sinteg
In [265]: f = lambda theta: exp(-pi*tan(theta)**2)/cos(theta)**2
In [266]: theta = linspace(-pi/2, pi/2)
In [267]: plot(theta, f(theta))
In [268]: grid()
In [269]: axis('equal')

In [270]: sinteg.trapz( f(theta), theta )
Out[270]: 1.0000000000024205
In [272]: sinteg.simps( f(theta), theta )
Out[272]: 1.0000000000024205
In [274]: sinteg.quad(f, -pi/2, pi/2)
Out[274]: (0.99999999999999989, 2.104653101673954e-12)
```

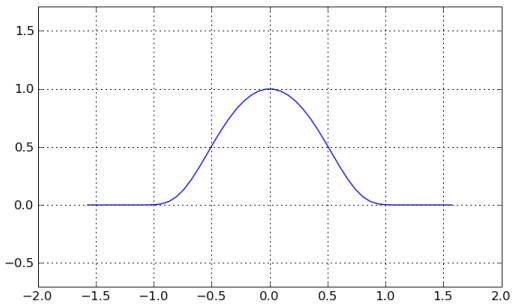


Figura 4: $f(\theta) = e^{-\pi \tan^2(\theta)} \sec^2(\theta)$

Los tres métodos empleados oscilan cerca de la respuesta $I = 1$.

(e) ¿Cuál es la relación entre esta integral y lo que aprendió en su curso de Estadística?

Solución: La función $f(x) = e^{-\pi x^2}$ es una manera de caracterizar el Gaussiano, i.e. es la función de densidad de probabilidad de una distribución normal.

3. Un pico de potencia en un circuito eléctrico resulta en la corriente:

$$i(t) = i_0 e^{-t/t_0} \sin(2t/t_0)$$

a través de una resistencia. La energía E disipada por la resistencia es:

$$E = \int_0^\infty Ri^2 \, dt$$

Encuentre la energía disipada dadas las condiciones $i_0 = 100$ amperios, $R = 0.5$ ohms, $t_0 = 0.01$ segundos.

Solución: Tomando en cuenta las condiciones del problema tenemos:

$$\begin{aligned} i(t) &= i_0 e^{-t/t_0} \sin(2t/t_0) \\ &= 100 e^{-100t} \sin(200t) \\ E &= \int_0^\infty 1/2 \left(100 e^{-100t} \sin(200t)\right)^2 \, dt \end{aligned}$$

esta última integral, aunque es complicada, también puede resolverse empleando técnicas analíticas. Sin embargo, nuevamente sólo nos interesan las aproximaciones numéricas. Con un poco de experimentación llegamos a la siguiente gráfica (este paso no es mandatorio, pero recuerde que siempre ganamos intuición al graficar):

```
In [295]: f = lambda t: 0.5*(100*exp(-100*t)*sin(200*t))**2
In [296]: t = linspace(0, 0.05, 1000)
In [297]: plot(t, f(t))
In [298]: grid()
```

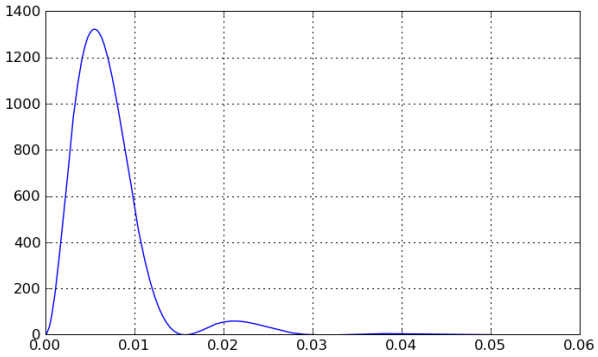


Figura 5: $f(x) = 1/2 \left(100 e^{-100t} \sin(200t)\right)^2$

Resulta ahora claro que debido a la parte exponencial de la función la energía se atenúa después de un corto tiempo. Lo que esto sugiere es que podemos prescindir de la integral impropia y conformarnos con

tomar límites de integración finitos, por ejemplo:

$$\begin{aligned} E &= \int_0^{\infty} 1/2 \left(100e^{-100t}\sin(200t)\right)^2 \, dt \\ &\approx \int_0^{0.07} 1/2 \left(100e^{-100t}\sin(200t)\right)^2 \, dt \end{aligned}$$

En Python:

```
In [309]: import scipy.integrate as sinteg
In [310]: f = lambda t: 0.5*(100*exp(-100*t)*sin(200*t))**2
In [311]: sinteg.quad(f, 0, 0.07)
Out[311]: (9.9999864784752379, 1.8899068957648204e-11)
In [312]: sinteg.quad(f, 0, 0.1)
Out[312]: (9.9999999631199614, 6.7936328027946561e-08)
In [313]: sinteg.quad(f, 0, 1)
Out[313]: (10.0, 2.3377212767099927e-08)
```

Vemos que la energía disipada converge a 10 watts.

Métodos Numéricos **Guía 14**
UVG-MM2010, MM2022

Instrucciones: En sus cursos de matemática se le ha recomendado que ataque los problemas desde varios enfoques: **analítico**, **gráfico**, y **numérico**. En este curso nos interesan particularmente los últimos dos enfoques. Responda las siguientes preguntas dejando constancia clara de su procedimiento; recuerde incluir **gráficas**, **código en Python**, y explicaciones que considere pertinentes.

1. Introducción

Hasta ahora hemos estado trabajando con métodos que involucran *algoritmos determinísticos*, empezaremos a explorar las *simulaciones de Monte Carlo* para resolver problemas empleando *algoritmos probabilísticos*. Monte Carlo es una familia de métodos que utilizan simulaciones con números aleatorios. Este método tomó auge con el uso de las computadoras a partir de los 1940's. Ejemplificamos ahora algunas funciones en Python que nos permiten trabajar con números aleatorios, e.g. `randint`, `uniform`, `normal`; recuerde ver su documentación:

```
In [48]: randint(100)
Out[48]: 98
In [49]: randint(100)
Out[49]: 45
In [51]: randint(25,100,4)
Out[51]: array([29, 48, 80, 28])
In [56]: uniform()
Out[56]: 0.24802909132378059
In [58]: uniform(size=5)
Out[58]: array([ 0.72053141,  0.2993958 ,  0.20633783,  0.49859363,  0.13243805])
In [59]: normal()
Out[59]: 0.67432444182854934
In [62]: normal(12, 0.1)
Out[62]: 11.912666707121497
```

Otras funciones útiles, e.g. `choice` y `sample`, están dentro del módulo `random`:

```
In [66]: from random import choice, sample
In [67]: L = ['a', 'b', 'c', 'd']
In [68]: choice(L)
Out[68]: 'c'
In [71]: sample(xrange(10000), 4)
Out[71]: [8822, 584, 9006, 1231]
In [72]: sample(xrange(10000), 4)
Out[72]: [8971, 9222, 3324, 9931]
```

Consideremos ahora unos ejemplos que resolveremos con métodos de Monte Carlo.

1.1. Lecturas en el disco duro

Queremos determinar la distancia promedio que recorre la cabeza del disco duro en distintos escenarios.

Disco duro I. Considere la posición inicial de un disco duro como cero (0) y la posición final como uno (1). La cabeza del disco duro lee archivos en posiciones aleatorias del disco. Luego de cada lectura, la cabeza del disco regresa a la posición inicial, i.e. cero (0). ¿Cuál es la distancia promedio que recorre la cabeza del disco duro? *Ayuda:* asuma una distribución uniforme.

Con el siguiente código diseñamos un experimento, asegúrese de entenderlo:

```
def disco_retorno(N):
    dist = 0                # distancia acumulada
    for k in range(N):      # N lecturas en disco
        pos = uniform()     # pos. de lectura
        dist = dist + 2*pos # dist. acum.
    return dist/N           # dist. promedio
```

En la terminal, con cada corrida del **experimento** obtenemos distintos resultados; ¿cómo podemos obtener cierta garantía que los resultados obtenidos convergen a algún punto específico? Una manera de responder esta pregunta es diseñar un **super-experimento**[1] (un experimento de experimentos) y representar los resultados en un *histograma*:

```
In [91]: disco_retorno(1e4)
Out[91]: 1.001336887700536
In [92]: disco_retorno(1e4)
Out[92]: 0.99185213581836618
In [93]: disco_retorno(1e4)
Out[93]: 1.001171755335186
In [94]: hist([disco_retorno(1e4) for k in xrange(1e3)])  # super-experimento
```

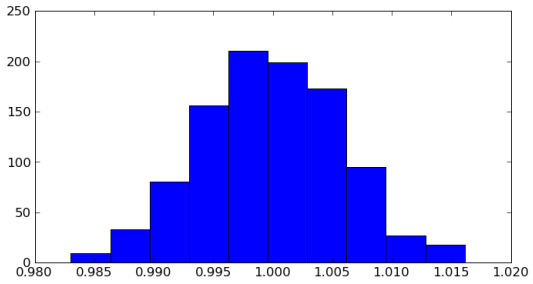


Figura 1: Histograma del super-experimento

Con el histograma resulta claro entonces que la distancia promedio es de 1.

Disco duro II. Considere el escenario del problema anterior con la variante que la cabeza del disco duro *no* regresa a la posición inicial (cero) luego de cada lectura. ¿Cuál es la distancia promedio que se mueve la cabeza del disco duro?

El experimento:

[1] Una forma sencilla de implementar un super-experimento es empleando *comprehensión de listas*.

```
def disco_perezoso(N):
    dist = 0                # distancia acumulada
    posprev = 0             # posicion previa
    for k in range(N):      # N lecturas en disco
        pos = uniform()     # pos. de lectura
        dist = dist + abs(pos - posprev) # dist. acum.
        posprev = pos       # actualiza la posicion
    return dist/N           # dist. promedio
```

Un par de corridas del experimento y luego el super-experimento:

```
In [103]: disco_perezoso(1e4)
Out[103]: 0.33471487282362561
In [104]: disco_perezoso(1e4)
Out[104]: 0.33690491696522301
In [105]: hist([disco_perezoso(1e4) for k in xrange(1e3)])
```

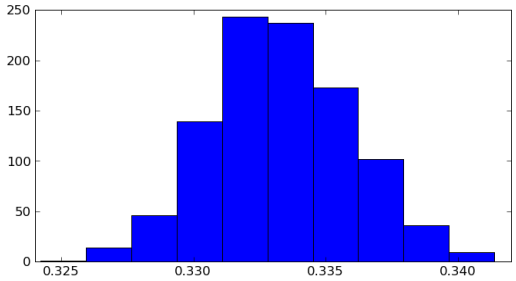


Figura 2: Histograma del super-experimento

En este nuevo escenario la distancia promedio es de $1/3$. *Nota:* ¿cómo cree que cambia el histograma si el número de lecturas por experimento es mayor?

Otra manera de estudiar la variación de los super-experimentos es graficar (k, E_k) donde k es el número de experimento y E_k es el resultado de dicho experimento. Esta gráfica se denomina “sample path”.

```
In [107]: plot([disco_retorno(1e4) for k in xrange(1e3)])
In [108]: plot([disco_perezoso(1e4) for k in xrange(1e3)])
In [109]: grid()
In [110]: axis([0,1000,0,1.1])
```

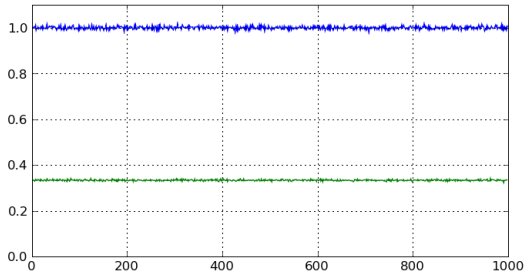


Figura 3: “Sample-path” del super-experimento

1.2. Aproximación para π

Considere el círculo unitario centrado en el origen inscrito en el rectángulo $(x, y) \in [-1, 1] \times [-1, 1]$. Utilizaremos el método de Monte Carlo para aproximar π con dos cifras decimales. Con el siguiente código obtenemos las gráficas y las respectivas aproximaciones para π:

```
def MCpi(N):
    x = uniform(-1, 1, N)      # coordenada en x
    y = uniform(-1, 1, N)      # coordenada en y
    T = (x**2 + y**2) < 1      # test
    n = len(find(T == True))   # pasan test
    return (4.0*n)/N, x, y     # regresa: area, x, y

def plotMCpi(x,y,piEst):
    xc = x[ x**2 + y**2 < 1]
    yc = y[ x**2 + y**2 < 1]
    xp = x[ x**2 + y**2 >= 1]
    yp = y[ x**2 + y**2 >= 1]
    plot(xp,yp,'sk')
    plot(xc,yc,'o',color='sandybrown')
    cir = Circle( (0,0), 1, facecolor='lightblue', edgecolor='k')
    gca().add_patch(cir)
    axis('equal')
    title("pi aprox.: %g; N: %g" % (piEst,len(x)))

for k in range(1,5):
    subplot(2,2,k)
    piEst,x,y = MCpi(10**(k+1))
    plotMCpi(x,y,piEst)
```

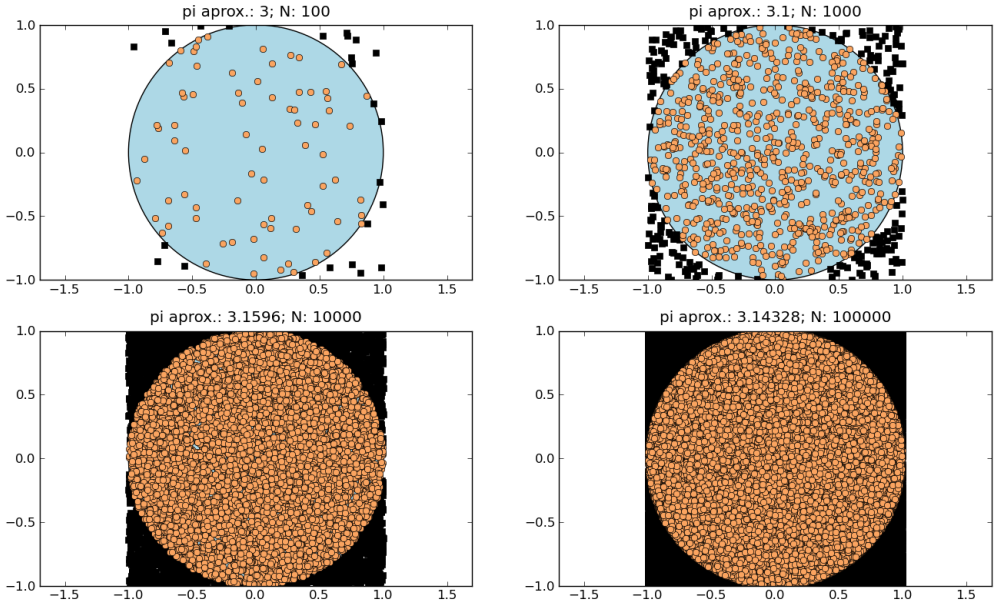


Figura 4: Monte Carlo en el círculo unitario

2. Preguntas

1. **Gaussiano en 3D.** Considere el gaussiano en dos variables:

$$f(x, y) = e^{-(x^2+y^2)}$$

(a) Grafique esta función. *Referencia:* vea el Lab 7.

Solución: En Python:

```
In [31]: x,y = mgrid[-2:2:100j, -2:2:100j]
In [32]: z = exp(-(x**2 + y**2))
In [33]: mlab.surf(x,y,z)
In [35]: mlab.contour_surf(x,y,z, contours=15)
```

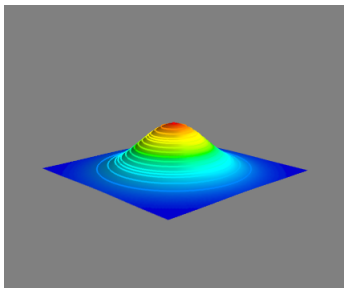


Figura 5: Gaussiano en 3D

(b) Utilice el método de Monte Carlo para aproximar la integral

$$I = \iint_{\mathbb{R}^2} e^{-(x^2+y^2)}\, dx\, dy$$

Solución: El diseño del experimento:

```
def gauss3D(N):
    x = uniform(-3,3,N)
    y = uniform(-3,3,N)
    z = uniform(0,1,N)
    T = exp(-(x**2 + y**2)) > z
    n = len(find(T == True))
    V = 6*6*1.0
    return V*n/N, x, y
```

las corridas del experimento y el histograma del super-experimento:

```
In [15]: I,x,y = gauss3D(1e5); I
Out[15]: 3.1392000000000002
In [16]: I,x,y = gauss3D(1e6); I
Out[16]: 3.1429800000000001
In [21]: hist( [gauss3D(1e6)[0] for k in xrange(1e2)] )
```

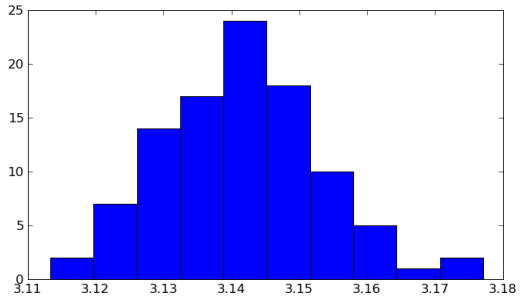


Figura 6: Histograma para el super-experimento de Gauss3D

Vemos entonces que esta doble integral converge a π. Para una perspectiva analítica de este problema vea el Lab 13, problema 2(b).

Por cierto, otra forma de resolver este problema numéricamente es emplear la función `dblquad` del módulo `scipy.integrate` que trabajamos en el laboratorio pasado:

```
In [42]: import scipy.integrate as sinteg
In [43]: f = lambda x,y : exp(-(x**2 + y**2))
In [44]: sinteg.dblquad(f, -Inf, Inf, lambda x: -Inf, lambda x: Inf)
Out[44]: (3.1415926535897762, 2.5173087568311278e-08)
```

Vemos nuevamente que la integral converge a π.

2. **Área elíptica.** Utilice el método de Monte Carlo para aproximar el área encerrada por la elipse:

$$\frac{x^2}{4} + y^2 = 1$$

Solución: El experimento:

```
def elipse(N):
    x = uniform(-2,2,N)
    y = uniform(-1,1,N)
    T = (0.25*x**2 + y**2) < 1
    n = len(find(T == True))
    A = 4*2.0
    return A*n/N, x, y
```

Las corridas en la terminal y el histograma del super-experimento:

```
In [141]: a,x,y = elipse(1e5); a
Out[141]: 6.2999200000000002
In [142]: a,x,y = elipse(1e5); a
Out[142]: 6.2862400000000003
In [145]: hist( [elipse(1e6)[0] for k in xrange(1e2)] )
In [146]: 2*pi
Out[146]: 6.2831853071795862
```

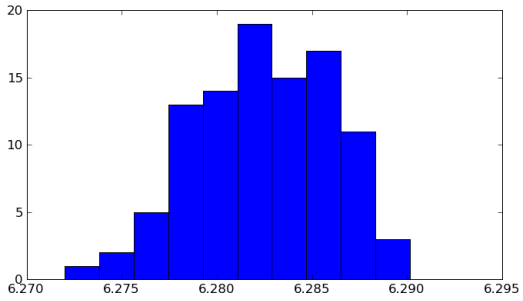


Figura 7: Histograma para el super-experimento del área elíptica

Tenemos entonces que el área de la elipse converge a 6.28 (la respuesta analítica es 2π).

3. **Reglas de conteo.** Un tablero de control tiene 10 switches (o interruptores, según el DRAE[2]) que pueden estar encendidos o apagados. Si el estado de cada switch es independiente de los otros, ¿cuál es la probabilidad de tener 4 switches encendidos? Utilice un experimento de Monte Carlo. *Ayuda:* Piense en cuántas secuencias de 10 bits (secuencias de ceros o unos) contienen 4 unos.

[2]Diccionario de la Real Academia Española de la Lengua.

Solución: Resolvamos primero este problema desde una perspectiva analítica; basta con recordar un poco de teoría de probabilidad. El número de maneras de tener 4 de 10 switches encendidos está dada por la combinatoria de 10 en 4, en símbolos:

$$\begin{aligned}\binom{10}{4} &= \frac{10!}{4!6!} \\ &= \frac{10 \cdot 9 \cdot 8 \cdot 7}{4 \cdot 3 \cdot 2 \cdot 1} = 210\end{aligned}$$

El número total de configuraciones de los switches es $2^{10} = 1024$, por lo que la probabilidad de tener 4 switches encedidos está dada por:

$$\frac{\binom{10}{4}}{2^{10}} = \frac{210}{1024} \approx 0.205$$

Diseñemos ahora un experimento de Monte Carlo, claramente hay varias formas de resolver estos problemas.

```
def switches(N):
    S = [True, False]                       # encendido, apagado
    Tb = lambda : len(find([choice(S) for k in xrange(10)]))  # funcion, tablero
    T = array([Tb() for k in xrange(N)])                      # N tableros
    n = len(find( T == 4 ))                 # tableros con 4 switches encendidos
    return float(n)/N                       # probabilidad
```

En la terminal corremos un par de experimentos y el histograma del super-experimento:

```
In [21]: switches(1e3)
Out[21]: 0.20799999999999999
In [24]: switches(1e6)
Out[24]: 0.20494200000000001
In [28]: hist( [switches(1e4) for k in xrange(1e3)] )
```

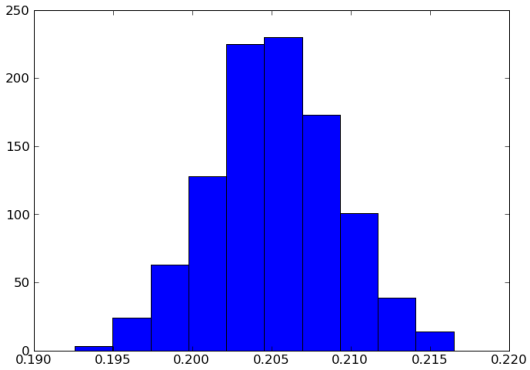


Figura 8: Histograma para el super-experimento del tablero con 10 switches

4. **Distribución Normal.** El volumen en las botellas de 12 onzas de Cerveza Gallo (orgullosamente nacional) obedece una distribución normal. El *valor esperado* es $\mathrm{E}(\mathcal{X}) = \mu = 12$ oz, y la desviación estándar es $\sigma = 0.1$ oz. ¿Cuál es la probabilidad de que una botella tenga un volumen entre 11.9 oz y 12.1 oz?

Solución: Notemos que estamos interesados en $\mathrm{pr}(\mu - \sigma \leq \mathcal{X} \leq \mu + \sigma) = \mathrm{pr}(11.9 \leq \mathcal{X} \leq 12.1)$. Dado que tenemos una *distribución normal* podemos emplear la *Regla 68-95-99*. Por lo que la respuesta es 0.68.

Tomando ahora un enfoque numérico, empleando un experimento de Monte Carlo.

```
def botellas(N):
    v = normal(12, 0.1, N)     # N muestras de distr. normal
    F = vectorize(lambda v: 11.9 < v < 12.1)  # vectorizacion de test
    T = F(v)                   # test del intervalo
    n = len(find(T))           # pasan test
    return float(n)/N          # probabilidad
```

Algunas corridas en la terminal:

```
In [74]: botellas(1e4)
Out[74]: 0.68759999999999999
In [75]: botellas(1e5)
Out[75]: 0.68027000000000004
In [77]: hist( [botellas(1e5) for k in xrange(1e3)] )
```

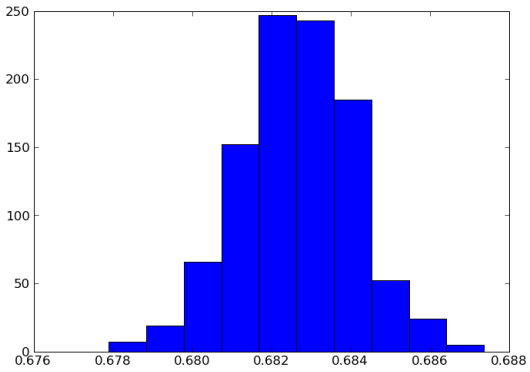


Figura 9: Histograma para el super-experimento del volumen en las botellas

5. **Entre amigos.** Dos amigos, Ana y Beto, deciden encontrarse entre 6pm y 7pm en el Café von Neumann[3]. Lamentablemente ambos tienen mucho trabajo y no están seguros del momento preciso en el que podrán llegar al café. También deciden que, al llegar, a lo sumo esperaran 10 minutos a que llegue el otro. ¿Cual es la probabilidad de que estos amigos se encuentren? *Ayuda:* Esboce una gráfica de tiempo de llegada de Ana vs. tiempo de llegada de Beto.

[3] John von Neumann es el padre del método de Monte Carlo. Stanislaw Ulam es la madre.

Solución: El experimento:

```
def amigos(N):
    A = uniform(size=N)
    B = uniform(size=N)
    T = abs(A - B) < 1.0/6
    n = len(find(T))
    x = A[abs(A - B) < 1.0/6]
    y = B[abs(A - B) < 1.0/6]
    return float(n)/N, x, y
```

En la terminal creamos una representación gráfica de este experimento:

```
In [263]: p,x,y = amigos(1e5)
In [264]: scatter(x,y)
In [265]: axis('equal')
In [266]: grid()

In [270]: p,x,y = amigos(1e5); p
Out[270]: 0.30531000000000003
In [271]: p,x,y = amigos(1e5); p
Out[271]: 0.30420000000000003
```

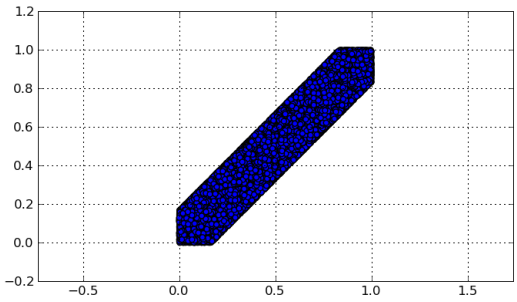


Figura 10: Representación gráfica del encuentro de Ana y Beto

Vemos que el experimento converge a la solución analítica: $11/36$.

6. **Aracnofobia.** Las tarántulas son, en general, temidas por los humanos aunque su efecto venenoso ha sido exagerado a través de los años. En los tiempos de la UFCo[4], el 3 % de los contenedores de banano proveniente de Honduras tenían tarántulas. De igual manera, 6 % de los contenedores de banano guatemalteco incluían al temido arácnido. Del total de la producción, 40 % era de Honduras y el resto era de Guatemala. En un muestreo aleatorio se encontró una tarántula. Utilice el método de Monte Carlo para determinar la probabilidad de que el contenedor era guatemalteco.

[4] United Fruit Company

Solución: La simbología de la información proporcionada es:

- $p(T|H) = 0.03$ (contenedores hondureños con tarántulas)
- $p(T|G) = 0.06$ (contenedores guatemaltecos con tarántulas)
- $p(H) = 0.4$ (banano hondureño)
- $p(G) = 0.6$ (banano guatemalteco)

Ahora bien, debemos encontrar la probabilidad condicional $p(G|T)$. Sabemos que la regla de Bayes nos permite intercambiar condicionalidades, i.e. pasar de $p(T|G)$ a $p(G|T)$; en símbolos:

$$\begin{aligned}
p(G|T)p(T) &= p(T|G)p(G) \\
p(G|T) &= \frac{p(T|G)p(G)}{p(T)} \\
&= \frac{p(T|G)p(G)}{p(TG) + p(T\bar{G})} \\
&= \frac{p(T|G)p(G)}{p(TG) + p(TH)} \\
&= \frac{p(T|G)p(G)}{p(T|G)p(G) + p(T|H)p(H)} \\
&= \frac{(0.06)(0.6)}{(0.06)(0.6) + (0.03)(0.4)} \\
p(G|T) &= 0.75
\end{aligned}$$

Por lo que tenemos 0.75 de probabilidad de que el contenedor era guatemalteco.

Diseñamos ahora un experimento de Monte Carlo:

```
def aracnofobia(N):
    D = uniform(size=(N,2))    # pais, aracnido
    TG = [(k[0] < 0.6) and (k[1] < 0.06) for k in D]  # GT con aracnido
    TH = [(k[0] > 0.6) and (k[1] < 0.03) for k in D]  # HN con aracnido
    n = float(len(find(TG)))   # numero de GT con aracnido
    m = float(len(find(TH)))   # numero de HN con aracnido
    return n/(n+m)             # prob.
```

En la terminal, un par de corridas del experimento y el histograma del super-experimento:

```
In [281]: aracnofobia(1e5)
Out[281]: 0.75571878279118576
In [282]: aracnofobia(1e5)
Out[282]: 0.74002089864158827
In [287]: hist( [ aracnofobia(1e4) for k in xrange(1e3) ] )
```

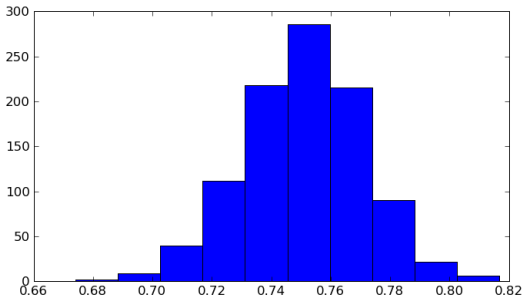


Figura 11: Histograma del super-experimento

Métodos Numéricos **Guía 15**
UVG-MM2010, MM2022

Instrucciones: En sus cursos de matemática se le ha recomendado que ataque los problemas desde varios enfoques: **analítico**, **gráfico**, y **numérico**. En este curso nos interesan particularmente los últimos dos enfoques. Responda las siguientes preguntas dejando constancia clara de su procedimiento; recuerde incluir **gráficas**, **código en Python**, y explicaciones que considere pertinentes.

1. Introducción

Le damos ahora un tratamiento numérico a las ecuaciones diferenciales ordinarias[1]. Consideremos por el momento el caso más sencillo, una ODE de primer orden:

$$\frac{dy}{dx} = f(x, y)$$

en su curso de Ecuaciones Diferenciales estudió métodos analíticos para resolver esta ecuación que dependían en gran medida de la función $f(x, y)$, e.g. separables, homogéneas, factor integrante, etc. Desde el punto de vista numérico el tratamiento es mucho más unificado, la clave está en *sustituir el diferencial por una diferencia finita*. Recuerde que cuando estudiamos derivación numérica encontramos que

$$\frac{dy}{dx} \approx \frac{\Delta y}{\Delta x} = \frac{y(x+h) - y(x)}{h}$$

ésta es la diferencia finita hacia adelante. Retomemos entonces la ecuación diferencial

$$\begin{aligned} \frac{dy}{dx} &= f(x, y) \\ \frac{y(x+h) - y(x)}{h} &= f(x, y) \\ y(x+h) &= y(x) + h\, f(x, y) \end{aligned}$$

usando subíndices, tenemos la *ecuación de recurrencia*

$$y_{k+1} = y_k + h\, f(x_k, y_k)$$

Nota: para resolver numéricamente una ecuación diferencial siempre necesitamos *condiciones iniciales*, o bien, *condiciones en la frontera*. Por cierto, esta sencilla idea se conoce como el **método de Euler**.

1.1. ODEs de primer orden

Consideremos por ejemplo la ecuación diferencial

$$\frac{dy}{dx} = \frac{y - x}{y + x}$$

[1] ODEs, "ordinary differential equations"

con condiciones iniciales $x = 0, y = 1$. Esta ecuación diferencial puede resolverse analíticamente, sin embargo este camino es largo y tortuoso (a menos que tenga fresco el material del curso de Ecuaciones Diferenciales). Una manera de resolver esta ODE analíticamente es notando que $f(x, y) = (y - x)/(y + x)$ es una *ecuación homogénea*, pues $f(tx, ty) = f(x, y)$. Lo usual es entonces proponer la sustitución $z = y/x$ para llegar a una ecuación diferencial separable. Otra manera de resolverla es emplear álgebra polar: $x = r\cos\theta, y = r\sin\theta$, encontrar los diferenciales dx, dy en términos de las nuevas variables r, θ y llegar a una ecuación diferencial separable.

Preferimos entonces resolver numéricamente. Tomemos el intervalo $x \in [0, 1]$, con $h = 0.1$.

$$\begin{aligned} \frac{dy}{dx} &= \frac{y - x}{y + x} \\ \frac{y(x + h) - y(x)}{h} &= \frac{y - x}{y + x} \\ y(x + h) &= y(x) + h\left(\frac{y - x}{y + x}\right) \end{aligned}$$

utilizando notación con subíndices:

$$y_{n+1} = y_n + h\left(\frac{y_n - x_n}{y_n + x_n}\right)$$

Con esta última ecuación de recurrencia podemos empezar a iterar para obtener $y_0, y_1, \ldots, y_n$. En lugar de seguir este camino, ilustraremos el uso del comando `odeint` de Scipy; cabe mencionar que `odeint` utiliza técnicas más robustas (y más complejas) que el simple método de Euler. Con el siguiente código obtenemos la tabla de valores para x_n, y_n. Es frecuente encontrar en la literatura que la variable independiente es t y no x; el código sigue esta convención.

```
# dy/dt = (y-t)/(y+t), y(0)=1
from scipy.integrate import odeint
f = lambda y,t: (y - t)/(y + t)   # f(t,y)
y0 = 1.                           # cond. inic.
a,b,h = 0, 1, 0.1                 # intervalo [a,b], delta t = h
tp = arange(a, b+h, h)            # --
yp = odeint(f, y0, tp)            # solver
```

x	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
y	1.0	1.0911	1.1678	1.2334	1.2901	1.3392	1.3816	1.4183	1.4496	1.4762	1.4982

Finalmente graficamos la solución de la ecuación diferencial:

```
In [159]: scatter(tp, yp)
In [160]: grid()
```

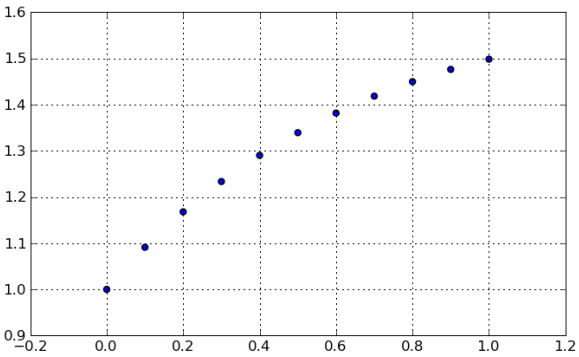


Figura 1: Puntos (x, y) que satisfacen $dy/dx = (y - x)/(y + x)$

1.2. ODEs de orden superior

Para resolver una ODE de orden superior primero la representamos como un sistema de ODEs de primer orden. Por ejemplo, considere la *ecuación de Airy* empleada en óptica:

$$y'' - ty = 0$$

con condiciones iniciales

$$y(0) = \frac{1}{3^{2/3}\Gamma(2/3)}$$
$$y'(0) = \frac{-1}{3^{1/3}\Gamma(1/3)}$$

donde $\Gamma(t)$ es la *función gamma*, una generalización de la función factorial, $n!$; ésta y otras funciones especiales pueden encontrarse en el módulo `scipy.special`. Ahora bien, para reescribir esta ODE de orden 2 en un sistema de ODEs de primer orden *proponemos* las siguientes variables $y_0 = y$, $y_1 = y'$ con estas nuevas variables la ecuación de Airy está dada por

$$y_1' - ty_0 = 0$$

y el sistema de ODEs de primer orden es

$$\begin{cases} y_0' &= y_1, \qquad y_0(0) = \frac{1}{3^{2/3}\Gamma(2/3)} \\ y_1' &= ty_0, \qquad y_1(0) = \frac{-1}{3^{1/3}\Gamma(1/3)} \end{cases}$$

En general, usando notación vectorial:

$$\frac{d\vec{y}}{dt} = f(\vec{y}, t)$$

donde, en este caso, $\vec{y} = \langle y_0, y_1 \rangle$.

En Python:

```
from pylab import *
from scipy.integrate import odeint
from scipy.special import gamma

y0_0 = 1/(3**(2/3.0)*gamma(2/3.0))
y1_0 = -1/(3**(1/3.0)*gamma(1/3.0))

y0 = [y0_0, y1_0]
f = lambda y,t : [y[1], t*y[0]]

tp = arange(0, 4, 0.01)
yp = odeint(f, y0, tp)
```

En la terminal:

```
In [192]: plot(tp, yp[:,0], lw=3)
In [193]: grid()
```

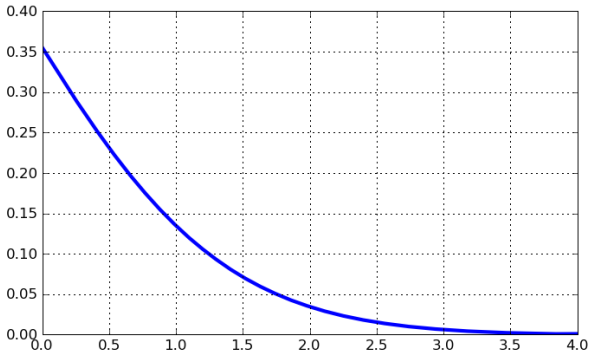


Figura 2: Puntos (x, y) que satisfacen $y'' - xy = 0$

2. Preguntas

1. Con el perdón de Shakespeare, consideremos la siguiente situación:

   Romeo está enamorado de Julieta, pero en nuestra versión de la historia Julieta tiene una personalidad bastante cambiante. Mientras más la ama Romeo, más piensa Julieta huir de la relación. Pero cuando Romeo se siente descorazonado y se aleja, Julieta empieza a encontrarlo extrañamente atractivo. Romeo, por el otro lado, tiende a hacer eco a los sentimientos de Julieta: se acerca si ella le quiere, y se aleja si ella le desprecia.

   Tomemos las descripciones (funciones):

   - $R(t)$: amor-odio de Romeo hacia Julieta en el tiempo t

- $J(t)$: amor-odio de Julieta hacia Romeo en el tiempo t

Convénzase que el siguiente sistema de ecuaciones diferenciales ordinarias lineales es una buena representación de la relación entre estos amantes.

$$\begin{cases} \frac{dR}{dt} = J, & R(0) = 1 \\ \frac{dJ}{dt} = -R, & J(0) = 0 \end{cases}$$

Resuelva numéricamente para $R(t), J(t)$. Considere $t \in [0, 2\pi]$, con $h = 0.1$. Grafique $J(t)$ y $R(t)$ vs. tiempo.

Solución: Tomamos $\vec{y} = \langle y_0, y_1 \rangle = \langle R(t), J(t) \rangle$.

```
################################################################
# Romeo y Julieta
y0_0 = 1.0          # R(0) = 1
y1_0 = 0.0          # J(0) = 0
y0 = [y0_0, y1_0]  # cond. inic.

# y = [R, J]
f = lambda y,t: [y[1], -y[0]]

tp = arange(0, 2*pi, 0.1)
yp = odeint(f, y0, tp)

plot(tp, yp[:,0], label='Romeo', lw=2, c='sandybrown')
plot(tp, yp[:,1], label='Julieta', lw=2, c='black')
grid()
legend(loc=0)
show()
```

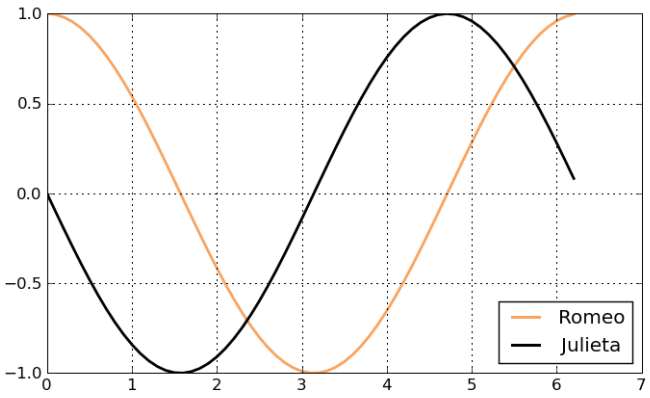


Figura 3: Relaciones amor-odio entre Romeo y Julieta

2. **Modelo para Tiros Libres.**

Las ecuaciones de movimiento para una pelota de futbol lanzada de tiro libre pueden obtenerse del análisis de fuerzas en el siguiente diagrama:

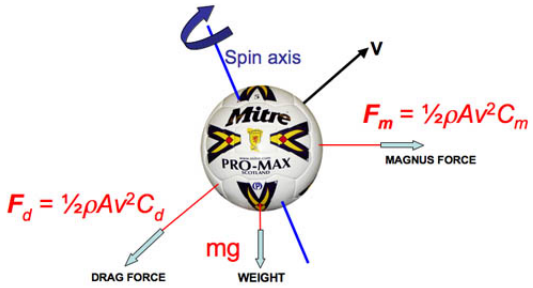


Figura 4: Fuerzas en una pelota lanzada de tiro libre. Fuente: `plus.maths.org`

Tenemos entonces el sistema de ecuaciones (las derivadas se toman con respecto al tiempo t)

$$\begin{cases} x'' &= -\nu k\,(C_d x' + C_l y') \\ y'' &= -\nu k\,(C_d y' - C_l x') \\ z'' &= -\nu k C_d z' - g \end{cases}$$

donde x, y, z son las coordenadas de la pelota, C_d es el coeficiente de resistencia aerodinámica ("drag coefficient"), C_l es el coeficiente de sustentación ("lift coefficient"), k es una constante dada por $\frac{\rho A}{2M}$ donde ρ es la densidad del aire, A es la sección de área cubierta por la pelota ("cross-sectional area"), M es la masa de la pelota; y ν es la velocidad de la pelota dada por $\sqrt{(x')^2 + (y')^2 + (z')^2}$.

Típicamente tenemos que $0.25 < C_d < 0.30$, $0.23 < C_l < 0.29$. Investigue los valores típicos para las otras constantes ρ, A, M así como para las condiciones iniciales (posición, velocidad) y resuelva este sistema de ODEs para las coordenadas x, y, z para poder así describir la trayectoria de la pelota como una función vectorial $\vec{r}(t) = \langle x(t), y(t), z(t)\rangle$. Finalmente, grafique la trayectoria de la pelota (referencia, Lab 7).

Nota: este problema requiere de experimentación numérica, existen varias respuestas correctas; en el proceso de solución tiene que tomar decisiones, a estas alturas del partido está listo para atacar situaciones como ésta.

Solución: A continuación se presenta una posible solución; recuerde que este tipo de problemas tienen muchas soluciones aceptables, como es usual en ingeniería.

```
from pylab import *
from scipy.integrate import odeint, cumtrapz
import scipy.interpolate as si
from enthought.mayavi import mlab

########################################
```

```
# Constantes
Cd = 0.28     # [adim.]
Cl = 0.26     # [adim.]
g = 9.8       # [m/s**2]
rho = 1.225   # [kg/m**3]
A = 0.038     # [m**2]
M = 0.44      # [kg]
K = (rho*A)/(2*M)

########################################
# Condiciones iniciales
v0 = 25           # [m/s]
psi = 21*pi/180   # angulos de pateo
theta = 4*pi/180  # --
vx_0 = v0*cos(psi)*sin(theta)
vy_0 = v0*cos(psi)*cos(theta)
vz_0 = v0*sin(psi)
V_0 = [vx_0, vy_0, vz_0]

########################################
# Solucion numerica
# V = [vx, vy, vz]
def f(V,t):
    vx,vy,vz = V
    return [-sqrt(vx**2 + vy**2 + vz**2)*K*(Cd*vx + Cl*vy), \
            -sqrt(vx**2 + vy**2 + vz**2)*K*(Cd*vy - Cl*vx), \
            -sqrt(vx**2 + vy**2 + vz**2)*K*Cd*vz - g ]
# arreglos de tiempo (tp) y de velocidades (Vp)
a,b,h = 0, 1.5, 0.1
tp = arange(a,b,h)
Vp = odeint(f, V_0, tp)
# arreglos de posicion (x,y,z)
x = hstack( (0, cumtrapz(Vp[:,0], tp)) )
y = hstack( (0, cumtrapz(Vp[:,1], tp)) )
z = hstack( (0, cumtrapz(Vp[:,2], tp)) )

########################################
# Graficas
xp,yp = mgrid[-15:15:100j, 0:30:100j]
mlab.surf(xp, yp, zeros_like(xp), color=(0.31, 0.66, 0.24))  # plano de la cancha
mlab.points3d(x,y,z)   # trayectoria de la pelota
```

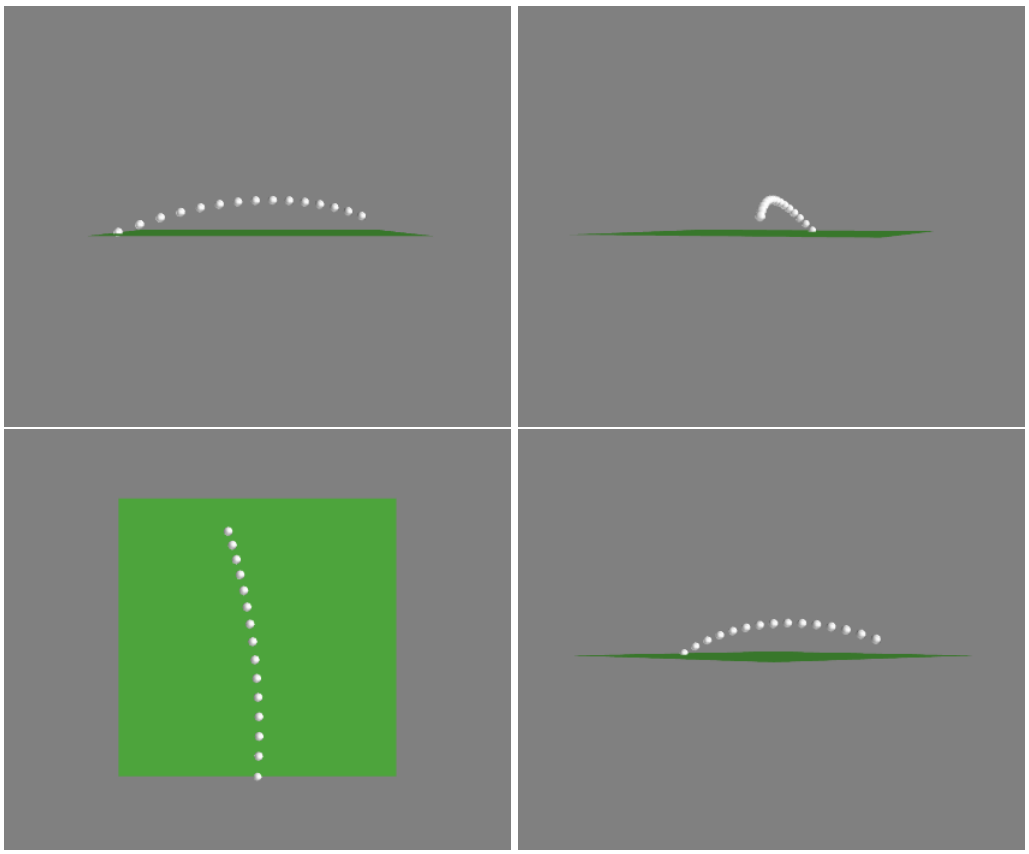


Figura 5: Trayectoria de la pelota.

En estos problemas la parte más interesante es considerar distintas condiciones iniciales (como velocidad, ángulos de pateo) para aconsejarle a los jugadores la técnica adecuada para patear un tiro libre. En breve, entender la ciencia del deporte.