

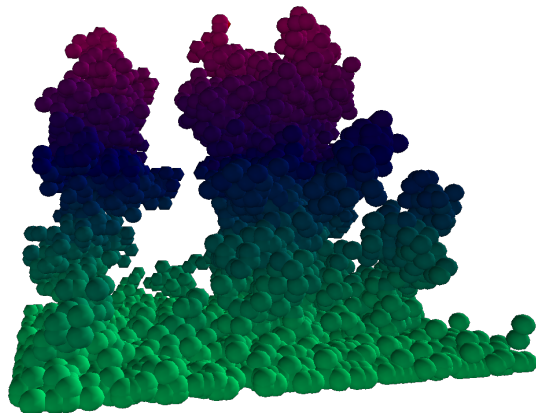
libro abierto / serie manuales

Pablo M. García Corzo

Proyecto Physthones

Simulaciones Físicas en Visual Python

~~~~ 0.1



▷ Un libro libre de Alqua



|            |                     |                |
|------------|---------------------|----------------|
| physthones | PROYECTO PHYSTHONES | 531(07)<br>ALQ |
|------------|---------------------|----------------|

† lomo para ediciones impresas

*Dedicado*

A todos los que se molestan en utilizar tizas de colores porque ayudan a enseñar

---

<http://alqua.org/documents/physthones>

Pablo M. García Corzo [ozrocpablo@gmail.com](mailto:ozrocpablo@gmail.com) <http://alqua.org>

# Proyecto Physthones

---

versión 0.1  
17 de octubre de 2008



alqua, **madeincommunity**



---

c o p y l e f t

---

Copyright (c) 2008 Pablo M. García Corzo.

Esta obra está bajo una licencia Reconocimiento - NoComercial - CompartirIgual 2.5 de Creative Commons. Para ver una copia de esta licencia visite <http://creativecommons.org/licenses/by-nc-sa/2.5/es> o escriba una carta a Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.

This work is licensed under the Creative Commons Reconocimiento - NoComercial - CompartirIgual 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/2.5/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.

---

**Serie** manuales

**Área** física computacional

**CDU** 531(07)

**Editores**

Pablo M. García Corzo [ozrocpablo@gmail.com](mailto:ozrocpablo@gmail.com)

Notas de producción

Plantilla `latex-book-es-b.tex`, v. 0.1 (C) Álvaro Tejero Cantero.

▷compuesto con software libre◁

# Índice general

|                                                                          |            |
|--------------------------------------------------------------------------|------------|
| <b>Portada</b>                                                           | <b>I</b>   |
| <b>Copyleft</b>                                                          | <b>VI</b>  |
| <b>Índice general</b>                                                    | <b>VII</b> |
| <b>1. Objetivos y capacidades (A modo de prólogo)</b>                    | <b>1</b>   |
| 1.1. Primeros pasos en la física computacional . . . . .                 | 1          |
| 1.2. ¿Por qué Python? . . . . .                                          | 2          |
| 1.3. Visual Python . . . . .                                             | 2          |
| 1.3.1. Instalación del sistema . . . . .                                 | 3          |
| <b>2. Toma de contacto: Python y Visual-Python</b>                       | <b>5</b>   |
| 2.1. Python e IDLE . . . . .                                             | 5          |
| 2.2. Visual . . . . .                                                    | 6          |
| 2.2.1. Operando con vectores . . . . .                                   | 7          |
| 2.2.2. Listas y diccionarios . . . . .                                   | 7          |
| 2.2.3. Lógica . . . . .                                                  | 7          |
| 2.2.4. Bucles . . . . .                                                  | 8          |
| 2.2.5. Rebota Mundo . . . . .                                            | 8          |
| <b>3. Modelos diferenciales</b>                                          | <b>11</b>  |
| 3.1. Tiro parabólico . . . . .                                           | 11         |
| 3.1.1. Física del tiro parabólico . . . . .                              | 11         |
| 3.1.2. Construcción de una escena . . . . .                              | 12         |
| 3.1.3. Animación . . . . .                                               | 13         |
| 3.1.4. Visualización de las fuerzas de Coriolis . . . . .                | 16         |
| 3.2. El péndulo simple y la aproximación para ángulos pequeños . . . . . | 19         |
| 3.2.1. La gracia del péndulo . . . . .                                   | 19         |
| 3.2.2. El oscilador armónico . . . . .                                   | 19         |
| 3.2.3. ¿Es el péndulo un oscilador armónico? . . . . .                   | 20         |
| 3.2.4. Péndulo simple y aproximación armónica . . . . .                  | 21         |
| 3.3. El péndulo inercial . . . . .                                       | 23         |
| 3.3.1. Un simpático sólido rígido en rotación . . . . .                  | 23         |
| 3.3.2. Construcción de la simulación . . . . .                           | 27         |
| 3.4. Breve introducción al caos . . . . .                                | 29         |
| 3.4.1. El atractor de Lorenz . . . . .                                   | 30         |

|                                                                              |           |
|------------------------------------------------------------------------------|-----------|
| 3.4.2. Construcción del módulo . . . . .                                     | 30        |
| <b>4. Modelos determinísticos y estocásticos</b>                             | <b>33</b> |
| 4.1. Simulación de un gas . . . . .                                          | 33        |
| 4.1.1. Precedentes históricos: Teoría de la gravitación de Le Sage . . . . . | 33        |
| 4.1.2. Teoría cinética . . . . .                                             | 34        |
| 4.1.3. Distribución de velocidades . . . . .                                 | 36        |
| 4.1.4. Magnitud de las velocidades moleculares . . . . .                     | 36        |
| 4.1.5. Magnitudes . . . . .                                                  | 36        |
| 4.1.6. Choques elásticos . . . . .                                           | 37        |
| 4.1.7. Reversibilidad e irreversibilidad . . . . .                           | 42        |
| 4.1.8. Conclusiones . . . . .                                                | 48        |
| 4.2. Breve introducción a los métodos Montecarlo . . . . .                   | 48        |
| 4.2.1. Simulación de crecimiento de dendritas . . . . .                      | 49        |
| <b>5. Jugando con Visual Python</b>                                          | <b>55</b> |
| 5.1. Visual Pong . . . . .                                                   | 55        |
| 5.2. Space Rebounder Racing . . . . .                                        | 58        |
| <b>Bibliografía</b>                                                          | <b>65</b> |
| <b>Historia</b>                                                              | <b>67</b> |
| <b>Creative Commons Deed</b>                                                 | <b>69</b> |
| <b>El proyecto libros abiertos de Alqua</b>                                  | <b>71</b> |
| <b>Otros documentos libres</b>                                               | <b>75</b> |



# 1 Objetivos y capacidades (A modo de prólogo)

Este documento pretende ser una presentación que sirva de guía sobre la que construir un taller de simulaciones físicas con *Visual Python*.

No debe entenderse, por tanto, como un tutorial de *Visual Python* como un texto de física computacional.

El proyecto Physthones pretende transmitir un carácter transgresor e innovador y parte de ello estará en el nuevo paradigma pedagógico hacker que viene a ser una actualización de la academia de Platón.

El hacker no aprende asistiendo a clases magistrales para escuchar la lección, el hacker decide modificar (hackear) un programa que otro había escrito para adaptarlo a una nueva necesidad y en el camino, a base de e-mails (con el propio autor u otros integrantes de la comunidad, a menudo verdaderos expertos) y búsquedas en la red obtiene los conocimientos necesarios.

Creemos que este modelo encaja a la perfección en la nueva ideología que está tratando de implantarse en nuestra universidad de menos clases presenciales y más trabajo por parte del alumno. Por eso creemos importante resaltar que lo que queremos conseguir no es tanto un curso sino un taller en el que haya lugar para la interactividad y el desarrollo comunitario.

## 1.1. Primeros pasos en la física computacional

La física computacional se ha ido convirtiendo en una herramienta de trabajo indispensable para cualquier físico ya sea experimental o teórico.

La experiencia directa de los integrantes del proyecto nos decía que los alumnos (incluso los ya licenciados) no tienen apenas capacidad para sacar provecho a las posibilidades que les brinda un ordenador más allá de el análisis estadístico de datos experimentales (conocimiento adquirido, todo sea dicho, de manera autodidacta por el alumno) y algunos ejercicios meramente académicos y poco aplicables de cálculo numérico y programación.

Consideramos importante reforzar, pues estos conocimientos en el currículum de un físico y si bien no pretendemos alcanzar este objetivo con el presente curso, sí que podemos allanar los primeros pasos en ese camino.

No creemos que deba ser nuestro objetivo en este pequeño curso dar una introducción formal a la física computacional. Sin embargo, un ordenador no excesivamente moderno permite obtener resultados suficientemente satisfactorios con los algoritmos más sencillos con sólo hacer suficientemente pequeño el paso diferencial y por tanto no se hace necesario

en esta primera aproximación el uso de técnicas sofisticadas de cálculo numérico ni meternos en complicaciones innecesarias.

Una primera aproximación a la física computacional integrada en los primeros cursos de física puede permitir, por ejemplo, que el alumno vea la mecánica como algo más que un pequeño conjunto de soluciones analíticas más o menos académicas. Chabbay y Sherwood introducen estas ideas en su curso de física general de primer año de carrera obteniendo excelentes resultados y proponiendo la experiencia para una reforma del currículum de física en la universidad [3].

### 1.2. ¿Por qué Python?

Cuando hemos presentado este proyecto en alguna conversación como la elaboración de un conjunto de simulaciones físicas para la docencia, surge inevitablemente la comparación inmediata con los extendidos “fislets” en Java y suelen sugerir que quizá estemos reinventando la rueda.

Desde luego, hay muchos lenguajes y sistemas posibles para llevar a cabo este proyecto, no sólo los applets de Java promiscuamente distribuidos por internet, también el menos conocido Box2D (ActonScript, Flash) o las opciones de visualización que integran cualquiera de los muchos paquetes numéricos y algebraicos como Matlab, Mathematica, Maple...

Por un lado, queríamos utilizar software libre (no en vano la idea surgió en el Grupo de Software Libre) y queríamos un sistema multiplataforma que pudiese utilizarse en cualquier máquina.

Otro de los requisitos que le pedíamos al sistema era que fuese lo más sencillo posible, pues queríamos romper esa “caja negra” en que se convierten a nuestros ojos los fislets java que encontramos por la red. La gran mayoría de los alumnos que llegan a los primeros cursos de física no han programado nada en su vida. Incluso en cursos superiores nos encontramos con que un porcentaje demasiado grande no han ido más allá de los ejercicios propuestos en el curso de introducción al cálculo numérico (actualmente impartido con Matlab en esta casa).

Por lo tanto el sistema debía ser lo más sencillo posible.

Python nos ofrecía esa sencillez y elegancia como lenguaje de iniciación[5] y además una gran cantidad de librerías y una importante comunidad de usuarios y desarrolladores que lo soportan.

### 1.3. Visual Python

La visualización y creación de escenas dinámicas tridimensionales (y no sólo el plotando de funciones) pueden tener un gran valor didáctico en muchos aspectos de la física básica como la dinámica del sólido rígido, la dinámica de fluidos o el electromagnetismo.

David Scherer crea desde la universidad Carnegie Mellon en el año 2000 un módulo de gráficos tridimensionales para Python extremadamente sencillo de usar llamado *Visual*.

El programador no tiene que lidiar con la complicación de las capacidades gráficas. Sólo debe describir los objetos en un lenguaje algebraico, natural para los físicos.

Con *Visual Python* no sólo se trata de que se haga sencilla la generación de escenas tridimensionales en movimiento, sino que el lenguaje algebraico que utiliza mantiene la coherencia con el utilizado en los cursos tradicionales, no suponiendo su inclusión en un curso de física fundamental una extensión sensible del temario sino un refuerzo positivo en el manejo de estas herramientas.

### 1.3.1. Instalación del sistema

Desde la página web del proyecto *Visual Python* (<http://vpython.org>) nos ofrecen unas instrucciones concisas y claras para realizar una instalación de todo lo necesario para windows, Linux y Mac con sólo un par de clicks.

Además, *Visual Python* está integrado en los repositorios de paquetes de la mayoría de las distribuciones Linux.

*1 Objetivos y capacidades (A modo de prólogo)*

## 2 Toma de contacto: Python y Visual-Python

No pretendemos convertir esto en un tutorial de python, sólo dar un breve paseo explicado sobre cómo trabajar en este sistema partiendo de cero.

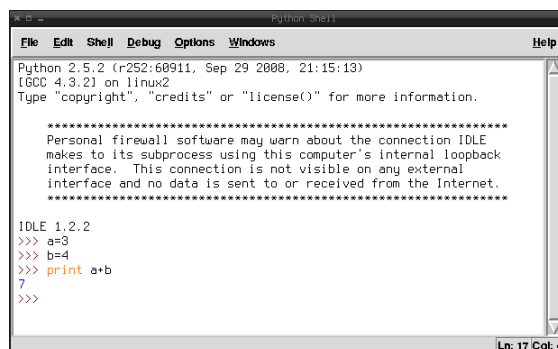
### 2.1. Python e IDLE

Python es un lenguaje interpretado, por lo que nos brinda la posibilidad de trabajar en sesiones interactivas utilizándolo como una calculadora programable.

No obstante, lo cómodo normalmente será escribir en archivos con cadenas de comandos (*scripts*) y pedir al sistema que los interprete todos de golpe.

Para programar necesitaremos, pues, el intérprete de python y un editor de texto. Como editor podríamos utilizar cualquiera capaz de leer y escribir en texto plano, como el notepad de windows (**no MSWord**), emacs, vi, pico, scite... o **IDLE**, que es el que viene incluido con la distribución de python, está especialmente pensado para el tipo de tareas que vamos a realizar y nos ayudará en determinadas ocasiones. Por ejemplo, cuando queramos ejecutar el archivo en el que estamos trabajando sólo tendremos que pulsar *F5*.

Cuando ejecutemos IDLE, tendremos una ventana con la consola interactiva de python en la que podríamos hacer, por ejemplo, una cuenta simple.



```
Python 2.5.2 (r252:60911, Sep 29 2008, 21:15:13)
[GCC 4.3.2] on linux2
Type "copyright", "credits" or "license()" for more information.

.....
Personal firewall software may warn about the connection IDLE
makes to its subprocess using this computer's internal loopback
interface. This connection is not visible on any external
interface and no data is sent to or received from the Internet.
.....

IDLE 1.2.2
>>> a=3
>>> b=4
>>> print a+b
7
>>>
```

Figura 2.1: Ventana interactiva de IDLE

## 2.2. Visual

En cuanto a *Visual Python*, serán un módulo que debemos cargar en python y nos permitirá dibujar sobre la marcha escenas tridimensionales:

```
from visual import *
caja=box(pos=vector(4,2,3), size=(8.,4.,6.), color=color.red)
bola=sphere(pos=vector(4,7,3), radius=2, color=color.green)
```

La primera línea simplemente le dice a python que queremos utilizar las herramientas de la clase visual (es decir, *Visual Python*). Lo normal y recomendable en python sería llamar a la librería simplemente con un *import visual*. Lo que hacemos al llamarlo con el *from* es escribir simplemente *box(...)* en lugar de *visual.box(...)*. Esto es peligroso hacerlo por norma pues dos librerías diferentes pueden tener funciones con nombres iguales y causar problemas que del otro modo evitaríamos. Sin embargo por comodidad en el tipo de cosas que vamos a hacer utilizaremos este otro método.

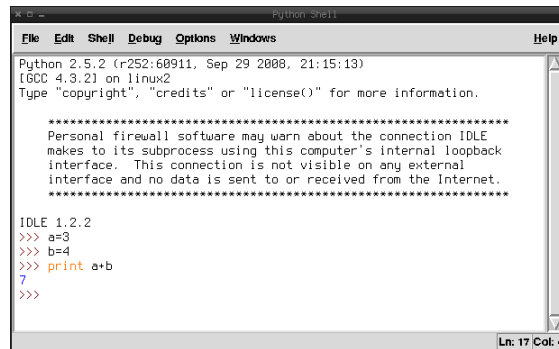


Figura 2.2: Nuestro "hola mundo" en python

Vemos que el lenguaje es bastante intuitivo. Definiremos las coordenadas con vectores en los que la coordenada  $x$  es la horizontal, la  $y$  la vertical y la  $z$  la perpendicular a la pantalla.

Una vez definido un objeto con unas ciertas propiedades, podemos cambiar éstas sobre la marcha o añadir nuevas que no estuvieran definidas:

```
caja.size=(8,0.1,8)
caja.pos=(0,-1,0)
bola.pos=(0,3,0)
bola.radius=1
bola.masa=1
bola.vel=vector(0,-1,0)
```

Al definir la velocidad de la bola, hemos especificado que se trata de un vector. En general esto es necesario en *Visual Python* para poder operar. No es necesario en el caso de las posiciones porque el sistema sabe ya que deben ser vectores, pero en los casos restantes deberemos especificarlo.

### 2.2.1. Operando con vectores

Con vectores podemos hacer las operaciones básicas de suma, producto por un escalar, producto escalar, producto vectorial, obtener la norma de un vector o su magnitud.

```
a = vector (1. ,2. ,3.)
b = vector (4. ,5. ,6.)
c=a+b
c=3.5*a
c=dot (a ,b)
c=cross (a ,b)
c=norm (a)
c=mag (a)
c=mag (a)**2
c=a/1
c=a/1.
```

Para la exponencial se utiliza en doble asterisco (como en gnuplot).

Hay que tener cuidado con las divisiones entre enteros. Aunque python no es tipado (no hay que definir explícitamente qué tipo de variable vamos a usar) sí que diferencia entre tipos y debemos tener cuidado con eso al operar.

### 2.2.2. Listas y diccionarios

Una cosa básica y tremendamente útil en python son las listas y los diccionarios.

Una lista es una serie de cosas (variables, objetos, vectores, nuevas listas anidadas...) ordenadas con un índice **que empieza en cero**:

```
moons = [Io , Europa , Ganymede , Callisto ]
print moons [1]
```

Un diccionario es una lista de pares asociados:

```
edad={ 'Margarita ': 12 , 'Pedro ': 10 , 'Luis ': 15 }
print edad [ 'Pedro ]
```

### 2.2.3. Lógica

Es una base importantísima de todo lenguaje de programación. Supongamos que queremos hacer que una pelota rebote en el suelo:

```
si la coordenada y de la pelota es menor que cero :
    entonces su velocidad pasa a ser positiva
    y su posición , cero .
```

Llevando esto a la realidad de python:

```
if bola.pos.y < 0:
    bola.vel*=-1
    bola.pos.y=0
```

Nótese que no hay, como en otros lenguajes de programación, un cierre de la condición. Python interpreta el indentado de la sintaxis como que todo lo que venga “metido hacia la derecha” del if será lo que deba leer sólo si la condición se satisface.

Esto es una gran ventaja pedagógica para aprender a programar con un orden adecuado. El indentado estándar en python es de 4 espacios.

#### 2.2.4. Bucles

Serán la base del movimiento. Podemos decir que mientras se cumpla tal condición se repitan una serie de comandos. En el siguiente ejemplo vemos cómo dada una velocidad para una partícula y un intervalo de tiempo pequeñito (diferencial) podemos animar el sistema. Es un esbozo de lo que se llama el método de diferencias finitas.

```
t=0
dt=0.001
while t < 100:
    t=t+dt

    bola.vel=bola.acel*dt
    bola.pos=bola.vel*dt
```

Otra opción interesante de los bucles es la de recorrer listas con, por ejemplo, cada una de las partículas de un sistema.

```
t=0
dt=0.001
while t < 100:
    t=t+dt
    for bola in listabolas:
        bola.vel=bola.acel*dt
        bola.pos=bola.vel*dt
```

#### 2.2.5. Rebota Mundo

Con lo que hemos aprendido ya podemos construir escenas y animarlas. Veamos una bola que cae por acción de la gravedad y rebota en el suelo:

```
from visual import *
grav=vector(0,-10,0)
bola=sphere(pos=(0,5,0), radius=1, color=color.green)
suelo=box(size=(8,0.1,8), pos=(0,-1,0), color=color.red)
```



```
bola.vel=vector(0,0,0)
dt=0.01
t=0
while t < 30:
    t+=dt
    bola.vel+=grav*dt
    bola.pos+=bola.vel*dt
    if bola.pos.y < 0:
        bola.vel*=-1
        bola.pos.y=0
    rate(50)
```

El `rate(50)` define el número de frames por segundo que deben visualizarse. Sin él, la animación no sería observable. Jugando con este valor y el intervalo de tiempo  $dt$  podemos crear las animaciones en tiempo real.



## 3 Modelos diferenciales

En este capítulo vamos a ver varios ejemplos de modelos diferenciales.

En la física se utilizan constantemente modelos descritos a través de ecuaciones diferenciales, la mayoría de los cuales no tienen soluciones analíticas.

Por norma general, en los primeros cursos de física sólo se estudian problemas con soluciones analíticas sencillas dejando aquellos cuyas soluciones son más complejas para cursos superiores.

De los problemas sin soluciones analíticas se puede hacer un estudio para obtener datos importantes acerca de su comportamiento y se pueden obtener soluciones aproximadas. A continuación vamos a ver el modo más sencillo de obtener una gran cantidad de información de sistemas de ecuaciones diferenciales arbitrarios.

### 3.1. Tiro parabólico

Vamos a construir un ejemplo muy simple que sirva para comenzar a entender el lenguaje y la forma de implementar modelos diferenciales. Si bien las soluciones del tiro parabólico no debieran tener ya secretos para nosotros nos servirá para entender el método numérico que vamos a utilizar.

#### 3.1.1. Física del tiro parabólico

Vamos a tratar este problema con un análisis newtoniano lo más simple e intuitivo posible.

Tenemos un sistema de ecuaciones diferenciales tal que:

$$\left\{ \begin{array}{l} \vec{v} = \frac{d\vec{x}}{dt} \\ \vec{a} = \frac{d\vec{v}}{dt} = (0, -g, 0) \end{array} \right\} \rightarrow \left\{ \begin{array}{l} \vec{x} = \vec{x}_0 + \vec{v}dt \\ \vec{v} = \vec{v}_0 + \vec{a}dt \end{array} \right. \quad (3.1)$$

Donde  $g$  es la aceleración de la gravedad (constante).

Partiendo de una posición inicial  $\vec{x}_0$  y una velocidad inicial  $\vec{v}_0$ , aplicaremos las ecuaciones utilizando un  $dt$  arbitrario lo más pequeño posible. Con ello obtenemos la posición y velocidad en el instante siguiente. Repetimos el procedimiento tomando como  $\vec{v}_0$  y  $\vec{x}_0$  los nuevos valores obteniendo entonces el siguiente paso, y así sucesivamente. Esta es la base del bucle que construiremos para la simulación.

Cuando  $dt \rightarrow 0$  la solución será exacta.

### 3.1.2. Construcción de una escena

En las primeras líneas debemos llamar a las librerías de Python necesarias, en este caso sólo el módulo Visual que generará los gráficos tridimensionales.

Después generamos la ventana donde se visualizarán los gráficos llamándola *scene* y le daremos como atributos un título y le decimos que no queremos que haga *autoscale*. El *autoscale* lo que hace es mover el zoom de la cámara para que todos los objetos se vean. Suele resultar bastante incómodo y el botón central del ratón nos permitirá hacer eso manualmente sobre la marcha.

```
from visual import *
scene=display()
scene.title='Tiro_parabolico'
scene.autoscale=0
```

Ahora vamos a definir unos valores iniciales para el problema, que serán la posición y velocidad de la partícula en el instante inicial.

```
pos=vector(-10,0,0)
vel=vector(10,10,0)
```

A continuación, definimos los objetos que formarán parte de nuestra escena. Básicamente necesitaremos el suelo, que definiremos como un paralelepípedo (*box*) muy estrecho, el proyectil será una esfera y dibujaremos un cilindro que haga las veces de cañón (por motivos meramente estéticos).

```
suelo=box(pos=(0,-1,0),size=(25,0.1,25),color=color.red)
cannon=cylinder(pos=pos0,axis=norm(vel))
proyectil=sphere(pos=(r0x,r0y,r0z),color=color.blue)
```

Veamos con un poco más de profundidad cómo se definen los elementos que acabamos de mostrar:

**box** : Es un paralelepípedo recto centrado en *pos*. El ancho, alto y largo del paralelepípedo los definimos con *size*. Por defecto tiene un eje definido de manera vertical en el sistema de referencia del objeto. Ese eje lo situaremos en el sistema de referencia externo con *axis* y podemos modificarlo en el sistema de referencia del objeto con *up*.

**cylinder** : Se trata de un cilindro circular al que definimos la posición del centro de su base, su eje y su radio.

**sphere** : Simplemente necesitamos definir su posición y radio.

En este primer ejemplo hemos explicado un poco los objetos que se definen en *Visual Python*, no obstante en ejemplos posteriores pasaremos por alto en general las descripciones de objetos, entendiendo que la sintaxis es suficientemente autoexplicativa y que hay una excelente documentación del proyecto[6].

### 3.1.3. Animación

Ahora viene el momento en que animamos la escena. Hemos puesto un paso  $dt$  muy pequeño, cuanto más pequeño más lenta (y más precisa) será la simulación, de modo que sólo cuando  $t \rightarrow 0$  la solución coincidirá exactamente con la trayectoria analítica.

```
grav=10
acel=vector(0,-grav,0)
dt=0.001
while proyectil.pos >= 0 :
    vel=vel+acel*dt
    pos=pos+vel*dt
    proyectil.pos=pos
    t=t+dt
```

El núcleo de la animación es un bucle `while` que viene a decir que mientras la coordenada  $y$  del proyectil sea mayor o igual que 0 el sistema repita los pasos de dentro del bucle.

Esos pasos consisten sencillamente en avanzar intervalos diferenciales y actualizar a cada paso los valores de la posición, velocidad y aceleración dados por el sistema de ecuaciones diferenciales.

Según lo hemos escrito, en un ordenador lento el movimiento se producirá muy despacio y en un ordenador rápido se producirá deprisa. Para poder controlar el tiempo de manera realista debemos jugar con el número de fotogramas por segundo y el intervalo diferencial. Para eso se utiliza el comando `rate`. Si queremos, por ejemplo, 25 fotogramas por segundo (aproximadamente los que ofrece una pantalla de televisión) pondremos:

```
dt=1./25.
...
while ...
    ...
    rate(25)
```

#### Elementos adicionales: Vectores y trayectoria

A continuación vamos a añadir algunos elementos más a la escena, como la trayectoria de la bola, el vector velocidad y sus componentes.

```
from visual import *
scene=display()
scene.title='Tiro_parabolico'
scene.autoscale=0
pos=vector(-10,0,0)
vel=vector(10,10,0)
grav=10
acel=vector(0,-grav,0)
```

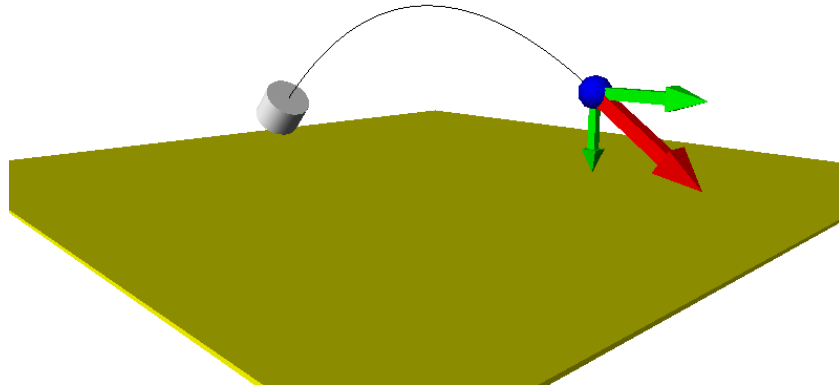


Figura 3.1: Nuestro programa de tiro parabólico en acción

```

dt=1./50.
t=0
proyector=sphere(pos=pos,color=color.blue,radius=0.5)
suelo=box(pos=(0,-1,0),size=(25,0.1,25),color=color.red)
cannon=cylinder(pos=pos,axis=(1,1,0))
trayectoria=curve(color=color.white)
velocidadtotal=arrow(color=color.red,pos=proyector.pos,axis=
    vel/3.)
velocidadx=arrow(color=color.green,pos=proyector.pos,axis=(vel
    .x/3.,0,0))
velocidady=arrow(color=color.green,pos=proyector.pos,axis=(0,
    vel.y/3.,0))
while pos.y >= 0 :
    vel=vel+acel*dt
    pos=pos+vel*dt
    trayectoria.append(pos)
    proyector.pos=pos
    velocidadtotal.pos=pos
    velocidadx.pos=pos
    velocidady.pos=pos
    velocidadtotal.axis=vel/3.
    velocidadx.axis=vector(vel.x/3.,0,0)
    velocidady.axis=vector(0,vel.y/3.,0)
    t=t+dt
    rate(50)

```

### Gráfica de energías

Ahora vamos a agregar una nueva ventana que nos muestre una gráfica de con la evolución de la energía cinética, potencial y total a lo largo del tiempo.

```

from visual import *
from visual.graph import *

scene=display()
scene.title='Tiro_parabolico'
scene.autoscale=0
scene.background=(1,1,1)

```

Veamos cómo agregar la nueva ventana para la gráfica

```

graph1 = gdisplay(x=0, y=0, width=300, height=150,
                  title='E_vs_t', xtitle='t', ytitle='E',
                  foreground=color.black, background=color.white)

```

Ahora añadiremos tres curvas en la ventana de gráficas:

```

potencial = gcurve(gdisplay=graph1, color=color.blue)
cinetica = gcurve(gdisplay=graph1, color=color.red)
etotal = gcurve(gdisplay=graph1, color=color.green)

pos=vector(-10,0,0)
vel=vector(10,10,0)
grav=10
m=1
acel=vector(0,-grav,0)
dt=1./50.
t=0
proyectil=sphere(pos=pos, color=color.blue, radius=0.5)
suelo=box(pos=(0,-1,0), size=(25,0.1,25), color=color.yellow)
cannon=cylinder(pos=pos, axis=(1,1,0))
trayectoria=curve(color=color.black)
velocidadtotal=arrow(color=color.red, pos=proyectil.pos, axis=
    vel/3.)
velocidadx=arrow(color=color.green, pos=proyectil.pos, axis=(vel
    .x/3.,0,0))
velocidady=arrow(color=color.green, pos=proyectil.pos, axis=(0,
    vel.y/3.,0))
while pos.y >= 0 :
    vel=vel+acel*dt
    pos=pos+vel*dt
    trayectoria.append(pos)

```

```

proyectil.pos=pos
velocidadtotal.pos=pos
velocidadx.pos=pos
velocidady.pos=pos
velocidadtotal.axis=vel/3.
velocidadx.axis=vector(vel.x/3.,0,0)
velocidady.axis=vector(0,vel.y/3.,0)

```

Y añadimos fácilmente puntos a las gráficas

```

cinetica.plot(pos=(t,0.5*m*mag2(vel)))
potencial.plot(pos=(t,m*grav*proyectil.pos.y))
etotal.plot(pos=(t,m*grav*proyectil.pos.y+0.5*m*mag2(vel)))
t=t+dt
rate(50)

```

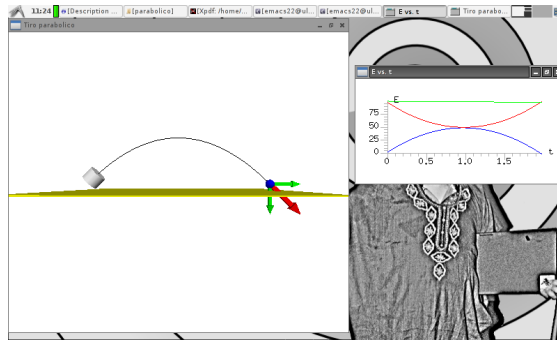


Figura 3.2: Tiro parabólico con gráfica de energías

### 3.1.4. Visualización de las fuerzas de Coriolis

Con lo que sabemos y un poquito de imaginación es sencillo poner el sistema a rotar y agregar efectos de coriolis, que vendrán descritos sencillamente como:

$$\sum \vec{F} = \begin{cases} -m\vec{\omega} \times (\vec{\omega} \times \vec{r}) & \rightarrow \text{Centrífuga} \\ -2m\vec{\omega} \times \vec{v} & \rightarrow \text{Coriolis} \\ -m\vec{\omega} \times \vec{r} & \rightarrow \text{Azimutal} \end{cases} \quad (3.2)$$

```

from visual import *
scene=display()
scene.title='Coriolis'
scene.autoscale=0
scene.background=(1,1,1)
# Velocidad de rotacion
omega=0.0

```



```

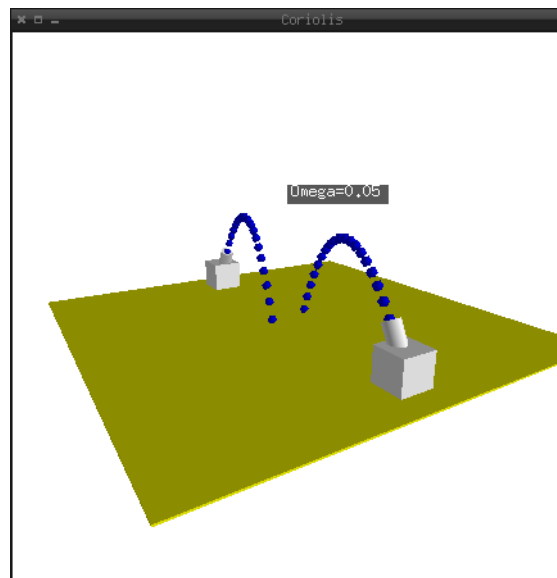
# Ira aumentandola en...
domega=0.00005
# Hasta una omega maxima de...
omegamax=0.05
# Nos sentamos en el sistema de referencia no inercial?
# 1 no
# -1 si
inercial=-1
chorros=[]
SRNI=frame()
box(frame=SRNI, pos=(0,-1,0), size=(25,0.1,25), color=color.
    yellow)
cylinder(frame=SRNI, pos=(-10,0,0), axis=(1,2,0), radius=0.5)
cylinder(frame=SRNI, pos=(10,0,0), axis=(-1,2,0), radius=0.5)
box(frame=SRNI, pos=(-10,0,0), size=(2,2,2), color=color.white)
box(frame=SRNI, pos=(10,0,0), size=(2,2,2), color=color.white)
leyenda=label(text="Omega="+str(omega), pos=(2,7,2))
def dispara(chorros):
    bola1=sphere(frame=SRNI, pos=vector(-10,0,0), color=color.
        blue, radius=0.3)
    bola2=sphere(frame=SRNI, pos=vector(10,0,0), color=color.blue
        , radius=0.3)
    chorros.append([bola1, vector(5,10,0)])
    chorros.append([bola2, vector(-5,10,0)])
    return chorros
def recoloca(i):
    if i[1].x > 0:
        i[0].pos=vector(-10,0,0)
        i[1]=vector(5,10,0)
    else:
        i[0].pos=vector(10,0,0)
        i[1]=vector(-5,10,0)
    return i
chorros=dispara(chorros)
grav=10
m=1
#acel=vector(0,-grav,0)
dt=1./50.
t=0
frec=0.1
Omega=vector(0,omega,0)
dOmega=vector(0,domega,0)
dentro=1
while true:

```

```

for i in chorros:
    r=vector(i[0].pos.x,0,0)
    acel=cross(Omega,cross(Omega,i[0].pos))+2*cross(Omega,r)
        +vector(0,-grav,0)+cross(dOmega,r)
    i[0].pos+=i[1]*dt
    i[1]+=acel*dt
    if i[0].pos.y <= -5:
        dentro=0
        i=recoloca(i)
t=t+dt
if t>=frec and dentro == 1:
    t=0
    chorros=dispara(chorros)
if inercial == -1:
    SRNI.rotate(angle=omega*dt, axis=(0,1,0))
rate(50)
if omega < omegamax:
    omega+=domega
    Omega=vector(0,omega,0)
    leyenda.text='Omega='+str(omega)

```



## 3.2. El péndulo simple y la aproximación para ángulos pequeños

### 3.2.1. La gracia del péndulo

Suele contarse que todo comenzó con un aburrido Galileo oyendo misa en la catedral de Pisa. Una lámpara que había sido empujada por un monaguillo al encenderla comenzó a describir unas oscilaciones que sin duda tenían más interés para Galileo que las palabras del sacerdote. Midió el período de las oscilaciones de la lámpara usando como reloj los latidos de su corazón y quedó fascinado al comprobar que aunque las oscilaciones descritas por la lámpara eran cada vez menores, dicho período se mantenía tozudamente constante.

Al llegar a casa, amarró una piedra a una cuerda y la colgó del techo para observar su movimiento. Lo hizo con diferentes cuerdas y piedras y su sorpresa no pudo ser mayor al descubrir que tampoco importaba lo gorda que fuese la piedra, sino únicamente la longitud de la cuerda.

Antes de abandonar definitivamente su carrera como médico, Galileo dejó su legado en ese mundo con el “pulsómetro”, que no era más que usar el péndulo como regla de medida para tomar el pulso a los pacientes. Invirtió el proceso que comenzara en la catedral.

El péndulo fue el sistema experimental que inspiró las leyes de caída de los cuerpos de Galileo, rompiendo por fin la idea aristotélica de que una bola de hierro caería más rápido que una de madera. Había nacido el nuevo paradigma, “la edad del péndulo”.

Naturalmente, luego llegarían los famosos (aunque supuestamente ficticios) experimentos de Galileo arrojando pesos desde la torre de Pisa, aunque eso fue sólo un segundo plato, quizá más adornado y aplaudido, pero nada interesante comparado con la magnificencia de un péndulo.

Observar lo que Galileo supuestamente mostró desde la torre de Pisa era muy complicado. Las cosas caían demasiado rápido por aquel entonces.

Las ligaduras (las del péndulo y las del plano inclinado) fueron las que permitieron a Galileo hacer caer los cuerpos lo suficientemente despacio como para tomar medidas con una “clepsidra”, estudiar su dinámica y predecir lo que sucedería al arrojarlos desde la torre de Pisa.

### 3.2.2. El oscilador armónico

Una manera de definir un oscilador armónico es decir que se trata de un movimiento con un tiempo característico al que llamaremos período ( $T$ ) en el que el sistema vuelve al estado de movimiento inicial:

$$x(t + T) = x(t) \quad (3.3)$$

Aplicando Lagrange sobre dicho sistema, y considerando para simplificar que el origen

### 3 Modelos diferenciales

de coordenadas coincide con el mínimo de potencial ( $U(x = 0) = 0$ ):

$$\begin{aligned} T(x) &= \frac{1}{2}m\dot{x}^2 \\ U(x) &= \frac{1}{2}kx^2 \\ \mathcal{L} &= T - U \\ \frac{d}{dt} \frac{\partial \mathcal{L}}{\partial \dot{q}} - \frac{\partial \mathcal{L}}{\partial q} &= 0 \end{aligned} \tag{3.4}$$

donde estamos denotando  $\dot{x} = \frac{dx}{dt}$

Obtenemos fácilmente la ecuación de movimiento:

$$m\ddot{x} + kx = 0 \tag{3.5}$$

donde  $\ddot{x} = \frac{d^2x}{dt^2}$

Llamando  $\omega = \frac{2\pi}{T} = \sqrt{\frac{k}{m}}$  a la frecuencia, que es un parámetro inversamente proporcional al período,

$$\ddot{x} + \omega^2 x = 0 \tag{3.6}$$

Las soluciones para este tipo de potencial son de tipo armónico, están bien estudiadas y son sencillas de tratar:

$$\left. \begin{aligned} x &= A\sin(\omega t) \\ &\text{ó} \\ x &= A\cos(\omega t) \end{aligned} \right\} \text{una c. lineal: } x(t) = A\cos(\omega t) + B\sin(\omega t) \tag{3.7}$$

#### 3.2.3. ¿Es el péndulo un oscilador armónico?

Un péndulo es, sencillamente una masa pendiente de un hilo. Para estudiar su dinámica hemos de considerar el hilo como una ligadura que expresamos matemáticamente como:

$$\left. \begin{aligned} x &= -L \sin \phi \\ y &= -L \cos \phi \end{aligned} \right\} \rightarrow \left\{ \begin{aligned} \dot{x} &= -L\dot{\phi} \cos \phi \\ \dot{y} &= L\dot{\phi} \sin \phi \end{aligned} \right. \tag{3.8}$$

A continuación procedemos a un análisis lagrangiano del sistema situando el origen de potenciales en el punto más bajo del péndulo.

$$\begin{aligned} T &= \frac{1}{2}m\dot{x}^2 + \dot{y}^2 = \frac{1}{2}mL^2\dot{\phi}^2 (\sin^2 \phi + \cos^2 \phi) = \frac{1}{2}mL^2\dot{\phi}^2 \\ U &= mgh = mgL(1 - \cos \phi) \\ \mathcal{L} &= \frac{1}{2}mL^2\dot{\phi}^2 - mgL(1 - \cos \phi) \end{aligned} \tag{3.9}$$

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \dot{\phi}} &= mL^2 \dot{\phi} \\
\frac{d}{dt} &= mL^2 \ddot{\phi} \\
\frac{\partial \mathcal{L}}{\partial \phi} &= -mgL \sin \phi \\
\frac{d}{dt} \left( \frac{\partial \mathcal{L}}{\partial \dot{\phi}} \right) - \frac{\partial \mathcal{L}}{\partial \phi} &= 0 \rightarrow mL^2 \ddot{\phi} + mgL \sin \phi = 0
\end{aligned}
\tag{3.10}$$

La ecuación diferencial que describe la dinámica del péndulo simple es, pues:

$$mL^2 \ddot{\phi} + mgL \sin \phi = 0 \tag{3.11}$$

Que es bastante compleja comparada con la ecuación del oscilador armónico (a pesar de ser tan parecidas) ya que para resolverla se requieren funciones elípticas.

No obstante se puede hacer la llamada aproximación para oscilaciones pequeñas que consiste básicamente en aproximar el seno por el ángulo  $\sin \phi \approx \phi$ .

Con ello, es inmediato obtener resultados aplicando las soluciones conocidas del oscilador armónico.

$$mL^2 \ddot{\phi} + mgL \phi = 0 \tag{3.12}$$

Sin embargo, no nos queda claro exactamente hasta qué punto es buena esta aproximación (cómo de pequeños tienen que ser esos ángulos) y, sobre todo, qué significa el hecho de que la apliquemos. ¿Cómo se comporta el sistema en realidad y cómo lo hace nuestra aproximación algebraica?

A veces, alumnos que hemos estudiado el péndulo simple hasta la saciedad nos hemos quedado intrigados al hacernos esta pregunta que debería resultarnos tan obvia. Construyamos una simulación sencilla que nos aclare las ideas.

### 3.2.4. Péndulo simple y aproximación armónica

Vamos a construir una simulación, en principio, de un péndulo simple.

```
#!/usr/bin/python
from visual import *
scene=display()
scene.title='Pendulo_simple'
scene.autoscale=0
```

Queremos introducir un ángulo inicial  $\phi_0$  (en radianes) desde el que activar el péndulo con una velocidad inicial nula.

```
phi0=pi*0.1
```

### 3 Modelos diferenciales

A continuación damos valores a los parámetros del sistema y definimos nuestro  $dt$  como paso

```
m=1.  
g=10.  
l=4.  
  
dt=1./50.
```

Queremos construir el péndulo como un cilindro estrechito y una esfera en el extremo. Para manejar varios objetos como uno sólo más complejo es útil utilizar los frames. Un frame hará el papel de un sistema de referencia de modo que los objetos que definamos como pertenecientes a ese frame utilizarán coordenadas propias a ese sistema de referencia y se moverían solidariamente con él.

```
SR=frame()  
masa=sphere(frame=SR, pos=(0., -1, 0.), radius=0.2, color=color.blue  
)  
hilo=cylinder(frame=SR, axis=masa.pos, radius=0.05, color=color.  
blue)
```

A continuación, vamos a preparar nuestra coordenada  $\phi$  en su posición inicial y su derivada respecto del tiempo ajustada a cero.

Colocaremos el sistema de referencia que contiene al péndulo en su posición inicial girándolo un ángulo  $\phi$  en torno a un eje perpendicular al plano de la pantalla:

```
phi=phi0  
phip=0  
SR.rotate(axis=(0,0,1), angle=phi)
```

Finalmente queda plantear el bucle que anima la escena teniendo en cuenta que en cada paso diferencial lo que hacemos es rotar el sistema de referencia un ángulo  $d\phi = \dot{\phi}dt$

```
while 1 :  
    phipp=-(m*g*l)*sin(phi)  
    phip=phip+phipp*dt  
    phi=phi+phip*dt  
    SR.rotate(axis=(0,0,1), angle=phip*dt)  
    rate(50)
```

Si, sencillamente, sustituyésemos la línea

```
phipp=-(m*g*l)*sin(phi)
```

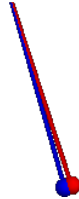
por

```
phipp=-(m*g*l)*phi
```

Obtendríamos la simulación del problema en aproximación para oscilaciones pequeñas.

Puede ser pedagógico dibujar dos péndulos de diferente color superpuestos con una de las soluciones cada uno para compararlos en tiempo real y ver cuál es exactamente el significado de hacer la aproximación para oscilaciones pequeñas.

```
sin(x) -> x
x = 0,1*pi
```



**Figura 3.3:** Comparación de un péndulo simple con y sin aproximación para oscilaciones pequeñas

### 3.3. El péndulo inercial

En la mecánica del sólido rígido, donde aparecen frecuentemente osciladores armónicos, se une a la posibilidad de simular soluciones numéricas, la capacidad de *Visual Python* de visualizar escenas tridimensionales manejando por parte del usuario diferentes sistemas de referencia.

#### 3.3.1. Un simpático sólido rígido en rotación

El montaje propuesto consta sencillamente de una plataforma horizontal que haremos girar con una velocidad angular constante  $\omega_0$ . Sobre ella se monta un soporte para sostener un eje libre paralelo a la plataforma.

El sólido rígido irá insertado en dicho eje, que lo atravesará por su centro de masas de manera que éste punto sea un punto fijo de movimiento.

#### Grados de libertad y coordenadas generalizadas

Por el hecho de ser un sólido rígido en el espacio posee 6 grados de libertad, tres de rotación y tres de traslación.

El punto fijo le hará perder sus tres grados de libertad de traslación y el eje que lo atraviesa dejará un único grado de libertad de rotación al sólido.

Tras estas consideraciones, podemos expresar la velocidad angular  $\omega_0$  en términos de un sistema de referencia no inercial ligado a la estructura como:

$$\vec{\omega}_0 = \omega_0 (0, \cos \phi, \sin \phi) \quad (3.13)$$

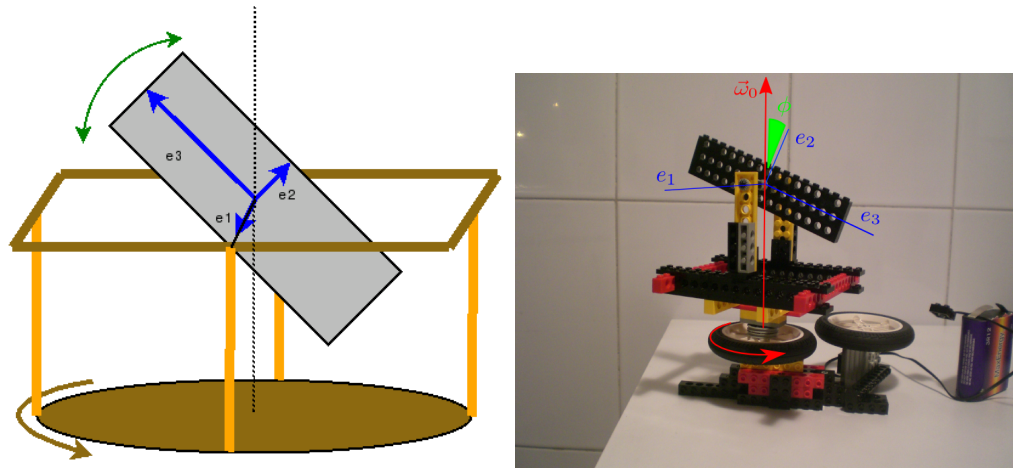


Figura 3.4: Montaje experimental

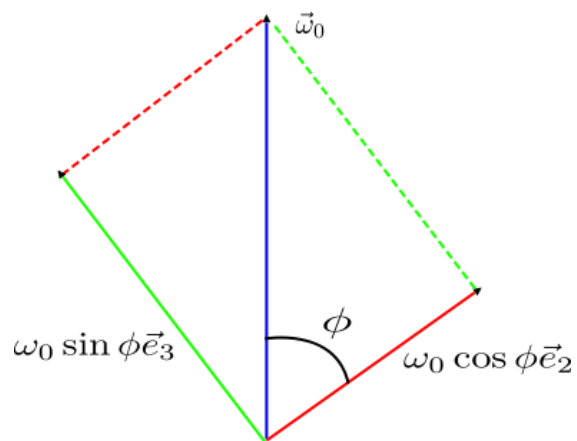


Figura 3.5: Descomposición de la velocidad angular constante



## Energía

La energía cinética la expresamos de manera sencilla en forma tensorial como:

$$\begin{aligned}
 T &= \frac{1}{2} \vec{\omega} \hat{I} \vec{\omega} \\
 \vec{\omega} &= \vec{\omega}_0 + \vec{\omega}_1 \\
 \vec{\omega}_0 &= \omega_0 (0, \cos \phi, \sin \phi) \\
 \vec{\omega}_1 &= (\dot{\phi}, 0, 0) \\
 \hat{I} &= \begin{pmatrix} I_1 & 0 & 0 \\ 0 & I_2 & 0 \\ 0 & 0 & I_3 \end{pmatrix} \\
 T &= \frac{1}{2} (\dot{\phi}, \cos \phi, \sin \phi) \begin{pmatrix} I_1 & 0 & 0 \\ 0 & I_2 & 0 \\ 0 & 0 & I_3 \end{pmatrix} \begin{pmatrix} \dot{\phi} \\ \cos \phi \\ \sin \phi \end{pmatrix} \\
 &= \frac{1}{2} I_1 \dot{\phi}^2 + \frac{\omega_0^2}{2} (I_2 \cos^2 \phi + I_3 \sin^2 \phi)
 \end{aligned} \tag{3.14}$$

La energía cinética presenta dos términos, uno cuadrático con las velocidades y otro que no depende de variaciones temporales de las coordenadas:

$$T = \underbrace{\frac{1}{2} I_1 \dot{\phi}^2}_{T_2} + \underbrace{\frac{\omega_0^2}{2} (I_2 \cos^2 \phi + I_3 \sin^2 \phi)}_{T_0} = T_2 + T_0 \tag{3.15}$$

Situar el centro de masas en un punto fijo nos permite fijar adecuadamente el origen del potencial gravitatorio para que no aparezca en nuestras ecuaciones.

El lagrangiano, por tanto, sólo tiene términos de energía cinética:

$$\mathcal{L} = \mathcal{L}_2 + \mathcal{L}_0 = \frac{1}{2} I_1 \dot{\phi}^2 + \frac{\omega_0^2}{2} (I_2 \cos^2 \phi + I_3 \sin^2 \phi) \tag{3.16}$$

## Ecuaciones de movimiento

Ataquemos ahora a la ecuaciones de movimiento:

$$\begin{aligned}
 0 &= \frac{d}{dt} \frac{\partial \mathcal{L}}{\partial \dot{\phi}} - \frac{\partial \mathcal{L}}{\partial \phi} \\
 \partial_t \frac{\partial \mathcal{L}}{\partial \dot{\phi}} &= \partial_t I_1 \dot{\phi} = I_1 \ddot{\phi} \\
 \frac{\partial \mathcal{L}}{\partial \phi} &= \omega_0^2 (I_2 \cos \phi \sin \phi - I_3 \cos \phi \sin \phi) = \\
 &= \omega_0^2 (I_2 - I_3) (\sin \phi \cos \phi) = \\
 &= \frac{\omega_0^2}{2} (I_2 - I_3) \sin 2\phi
 \end{aligned} \tag{3.17}$$

Llegamos a una expresión bastante elegante:

$$I_1 \ddot{\phi} - \frac{\omega_0^2}{2} (I_2 - I_3) \sin 2\phi = 0 \quad (3.18)$$

### El péndulo inercial

Lo que nos llama la atención al ver la ecuación 3.18 es el enorme parecido que tiene con la ecuación del oscilador armónico. Además, cuando nos ponemos a estudiar el potencial y encontramos puntos de equilibrio estable lo primero que se nos ocurre es ponerlo a oscilar.

Veamos si encontramos algo interesante.

**Aproximación para oscilaciones pequeñas** Podemos poner el sistema a describir oscilaciones tan pequeñas en torno a  $\phi_0$  como para que podamos considerar  $\sin 2\phi \approx 2\phi$ .

La ecuación de movimiento se nos queda entonces como un oscilador armónico perfectamente reconocible:

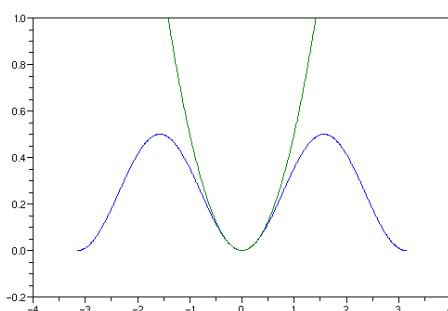
$$\underbrace{I_1}_{M_i} \ddot{\phi} - \underbrace{\omega_0^2 (I_2 - I_3)}_{K_i} \phi = 0 \quad (3.19)$$

El potencial generalizado del oscilador armónico viene dado por:

$$U_i = \frac{1}{2} K_i \phi^2 = \frac{\omega_0^2}{2} (I_2 - I_3) \phi^2 \quad (3.20)$$

Vemos que en el proceso de aproximación hemos perdido un término constante:

$$U \approx -\frac{\omega_0^2}{2} \left( I_2 \left( 1 - \frac{\phi^2}{2} \right)^2 + I_3 \phi^2 \right) \quad (3.21)$$



**Figura 3.6:** Aproximación por oscilador armónico. Comparación de potenciales.

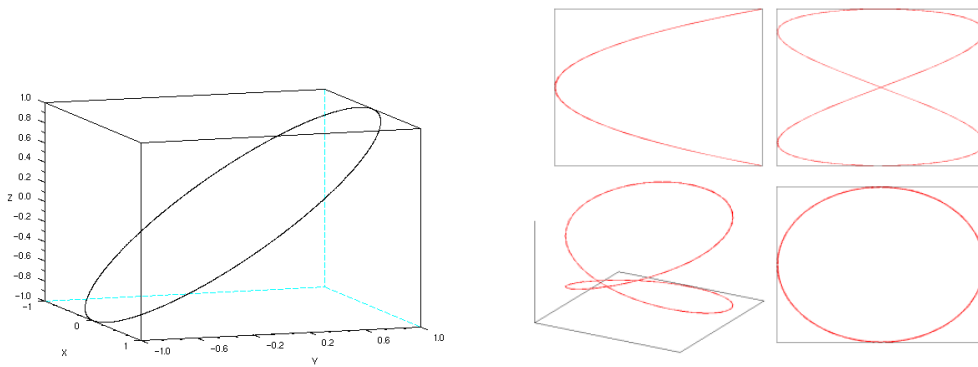
**Frecuencia característica del oscilador** Tener frente a nosotros un oscilador armónico nos pide buscar la frecuencia normal del sistema:

$$\omega_c^2 = \frac{\omega_0^2 (I_2 - I_3)}{I_1} \quad (3.22)$$

Puede ser divertido jugar con los parámetros del problema de modo que se acople la frecuencia característica del péndulo inercial con la frecuencia que le metemos externamente al sistema para dibujar figuras de Lissajous.

Si, por ejemplo,  $I_2 - I_3 = I_1$  estas frecuencias se igualarían y mirando el sistema de perfil veríamos una hermosa elipse (una recta con el desfase adecuado).

El caso más simple al que se nos ocurre tratar de aplicar esto es a la barra delgada en la que  $I_3 \approx 0$  y  $I_1 = I_2$ .



**Figura 3.7:** Figuras de Lissajous 1/1 y 1/2

Exactamente el mismo resultado sería el obtenido para cualquier figura plana que pudiésemos en el sistema utilizando un eje de rotación libre distinto de aquel cuyo momento de inercia es suma de los otros dos ( $I_2 = I_1 + I_3$ ).

*Parece cosa de magia que obtener esas figuras de Lissajous dependa exclusivamente de las proporciones geométricas del sólido y no de la frecuencia que imponemos al sistema no inercial ni de la amplitud con que produzcamos las oscilaciones del péndulo inercial.*

### 3.3.2. Construcción de la simulación

A continuación vamos a construir una simulación computacional sin la necesidad de la aproximación para ángulos pequeños de modo que nos ofrezca soluciones un poco más cercanas a la realidad.

La única complicación a la hora de llevar este montaje a Python estará en expresar correctamente las transformaciones de coordenadas y las relaciones que las ligan.

Empezaremos como siempre:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
from visual import *
```

```
scene=display ()
scene.title='Inertial_Pendulum'
scene.autoscale=0
```

Defino las dimensiones del trompo:

```
l1=0.7
l2=5
l3=12.5
```

El coeficiente que relaciona las frecuencias para un paralelepípedo recto es:

$$w_c^2 = w_0^2(I_2 - I_3)/I_1 \quad (3.23)$$

```
coef=((l3**2-l2**2)/(l3**2+l2**2))*(-0.5)
dim=l3*3/4
```

El coeficiente *dim* solamente se usa para dar proporciones estéticas a la representación. A continuación definimos la posición inicial (ángulo)

```
phi0=2
```

Para definir el sistema vamos a utilizar dos sistemas de referencia no inerciales (a parte del inercial predefinido por el sistema). Uno de ellos soportará la plataforma giratoria y el otro (ligado a éste) girará con el trompo. Montaremos toda la parafernalia sobre ellos.

```
# Sistema de referencia en rotacion
SRR=frame(pos=(0,0,0))
# Sistema de referencia del trompo
SRP=frame(frame=SRR)

# Ejes y soporte
ejes=curve(pos=[(dim*3/4,0,0),(0,0,0),(0,dim*3/4,0),(0,0,0)
,(0,0,dim*3/4)],color=color.blue)
Soporte1=box(frame=SRR, pos=[0,-dim,0],size=[dim,0.5,dim],color
=color.green)
Soporte2=box(frame=SRR, pos=[dim/2,-dim/2,0],size=[0.5,dim
,0.5],color=color.green)
Soporte3=box(frame=SRR, pos=[-dim/2,-dim/2,0],size=[0.5,dim
,0.5],color=color.green)
Eje=cylinder(frame=SRR, pos=[-dim/2,0,],axis=[dim,0,0],radius
=0.2,color=color.green)

# Defino dos bolas como marcadores (no tienen sentido fisico)
bola1=sphere(frame=SRP, pos=[0,l3/2,0])
```

```

bola2=sphere ( frame=SRP , pos=[0, -13 / 2 , 0])
bola1 . radius=0.5
bola1 . color=color . red
bola2 . radius=0.5
bola2 . color=color . blue

# El trompo :
top=box ( frame=SRP , pos=(0,0,0) , size=(12 , 13 , 11) , axis=(0,0,13) ,
        color=color . blue)

```

Las bolas las hemos puesto sobre los extremos del trompo sólo como marcadores para seguir la trayectoria.

Ahora terminaremos de preparar el sistema en las condiciones iniciales para empezar a moverlo:

```

omega=1.
dt=0.02
phi=pi/2-phi0
phip=0.
phipp=omega**2.*coef*sin(2.*phi)/2.
SRP . rotate ( frame=SRR , axis=(-1,0,0) , angle=phi)

```

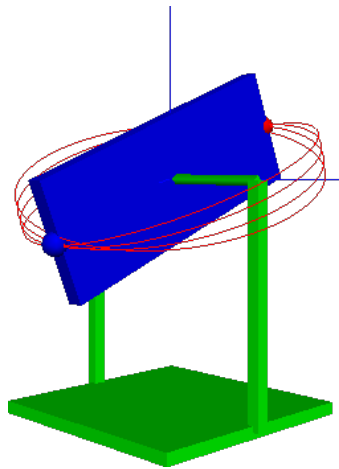


Figura 3.8: Simulación del péndulo inercial

### 3.4. Breve introducción al caos

En los años 60, Edward Lorenz trabajaba en un modelo de predicción del tiempo en el MIT. Había construido un sistema de doce ecuaciones que ligaban los efectos de la temperatura, presión, velocidad del viento... en la dinámica atmosférica.

Mediante simulación numérica por computador estudiaba cómo evolucionaba el sistema a partir de unas condiciones iniciales tratando de encontrar pautas y repeticiones.

En cierta ocasión introdujo unos valores iniciales que ya había utilizado antes pero redondeados a un menor número de cifras decimales. Esperaba encontrar un resultado muy similar al obtenido con esos valores sin redondear pero no fue así.

El sistema de ecuaciones de Lorenz era completamente determinista, es decir que no había azar de por medio. De unos ciertos valores iniciales debían necesariamente obtenerse siempre las mismas soluciones. Sin embargo una mínima variación producía cambios catastróficos a largo plazo.

Al efecto que tienen esos pequeños cambios de las condiciones iniciales en las condiciones finales se lo denominó *dependencia sensitiva de las condiciones iniciales* o más vulgarmente conocido como *efecto mariposa* por la siguiente reflexión de Lorenz para explicarlo.

Imaginemos que un meteorólogo fuese capaz de calcular con infinita precisión el estado de la atmósfera en un instante determinado pero olvidase tener en cuenta el aleteo de una mariposa. Ese aleteo podría inducir al cabo de un cierto tiempo un huracán en el otro extremo del planeta.

#### 3.4.1. El atractor de Lorenz

Si bien hemos dicho que el modelo de Lorenz era de doce ecuaciones, cuando comenzó a estudiar este nuevo campo de la física y la matemática abandonando las predicciones meteorológicas lo redujo a tres ecuaciones no lineales con tres variables.

$$\frac{dx}{dt} = S(y - x) ; \frac{dy}{dt} = x(b - z) - y ; \frac{dz}{dt} = xy - rz \quad (3.24)$$

Aunque el modelo de Lorenz no tiene un análogo inmediato en la realidad, suele asemejarse al movimiento cilíndrico de un gas o líquido caliente.

#### 3.4.2. Construcción del módulo

Queremos construir un módulo que nos muestre las peculiaridades del atractor de Lorenz, para ello pondremos un paquete de partículas muy juntas en un momento determinado en unas condiciones iniciales muy similares y veremos cómo evoluciona el sistema hasta convertirse en un caos total.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from visual import *
from random import *

scene=display()
scene.title='Atractor_de_Lorenz'
scene.autoscale=0
```

```
scene.range=(100,100,100)
```

Aunque utilicemos el paquete random, que sirve para obtener números aleatorios, sólo lo usaremos a la hora de fijar las condiciones iniciales para no generar siempre los mismos resultados.

Con *scene.range* definimos el tamaño de la escena.

Los siguientes valores de *r*, *S* y *b* dan soluciones caóticas.

```
r=28.
S=10.
b=8./3.
```

Como queremos mostrar unas cuantas partículas las meteremos en una lista. Crearemos asimismo una lista para guardar velocidades, trayectorias y posiciones:

```
tray=[]
bola=[]
vel=[]
pos=[]
```

Decidimos que queremos 5 partículas, y situamos la primera en el punto pos0 (aleatorio).

```
parts=5.
pos0=vector(randint(-10,10),randint(-10,10),randint(-10,10))
def empieza():
    i=0
    while i < parts :
        tray.append(curve(color=(i/parts,1-i/parts,0)))
        bola.append(sphere(color=(i/parts,1-i/parts,0)))
        vel.append(vector(0,0,0))
        pos.append(vector(pos0.x,pos0.y,pos0.z+0.01*i))
        bola[i].pos=pos[i]
        i+=1
```

Con este primer bucle (la función empieza) simplemente generaremos cada una de las partículas en sus posiciones iniciales (todas muy juntitas).

```
t=0
fps=25
dt=1./(3.*fps)
```

Ajustamos los parámetros temporales para la animación y empezamos a mover las partículas.

```
empieza()
while 1 :
    i=0
```

```

while i < parts :
    vel[i].x=-S*pos[i].x+S*pos[i].y
    vel[i].y=-pos[i].x*pos[i].z+r*pos[i].x-pos[i].y
    vel[i].z=pos[i].x*pos[i].y-b*pos[i].z
    pos[i].x=pos[i].x+vel[i].x*dt
    pos[i].y=pos[i].y+vel[i].y*dt
    pos[i].z=pos[i].z+vel[i].z*dt
    bola[i].pos=pos[i]
    tray[i].append(pos[i])
    i+=1

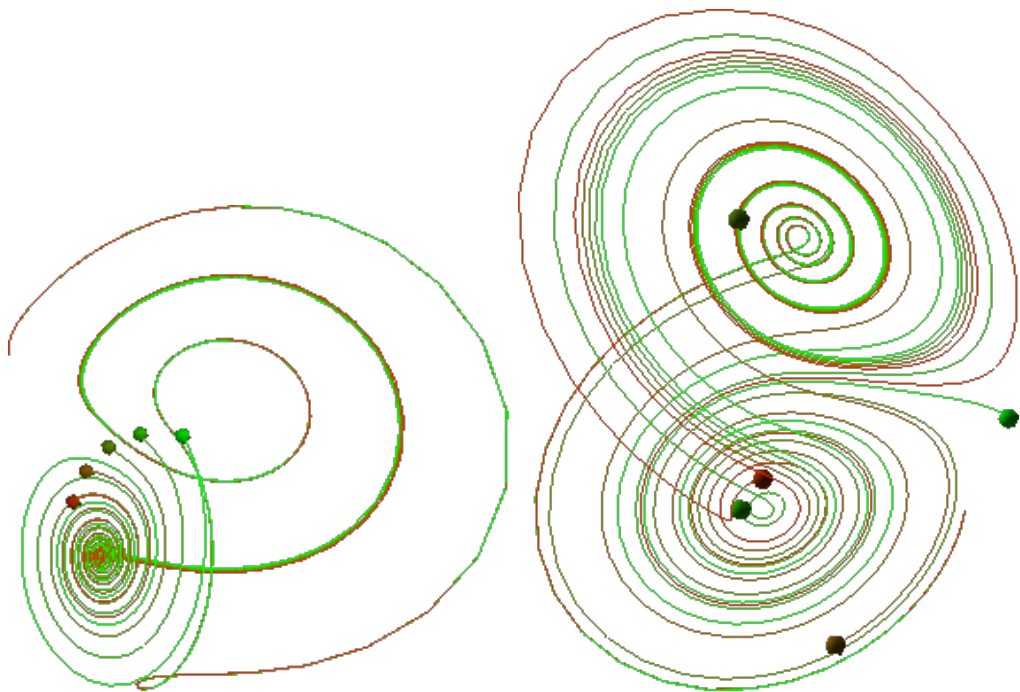
```

Este bucle recorre todas las partículas en cada paso del bucle principal.

```

t+=dt
rate(fps)

```



**Figura 3.9:** Evolución del atractor de Lorenz



## 4 Modelos determinísticos y estocásticos

Los modelos estocásticos son aquellos que están afectados por ruido, es decir, en los que interviene el azar. El primer modelo de este tipo en la física fue propuesto en 1908 por Langevin para describir el movimiento Browniano.

En el caso del atractor de Lorenz partíamos de algo impecablemente mecanicista, unas condiciones iniciales infinitamente bien determinadas ofrecían resultados infinitamente idénticos. Sin embargo, una mínima indeterminación en esas condiciones iniciales introducía resultados aparentemente caóticos en las soluciones.

En los dos casos que vamos a ver a continuación podría decirse que sucederá lo contrario. Introduciremos caos en el sistema y éste tenderá en cierto modo a auto-organizarse matemáticamente. En el primer caso será un caos un tanto falso al inicio salvo por las condiciones iniciales, pero en el siguiente ejemplo un movimiento puramente aleatorio generará algo que tiene un cierto y misterioso orden.

### 4.1. Simulación de un gas

Por norma general suelen estudiarse en los primeros cursos de física la mecánica y la termodinámica como si se tratase de dos ramas de la física totalmente disconexas que apenas parecen pertenecer a una misma disciplina. Sin embargo, la física estadística llega a enlazarlas de un modo bastante elegante y convincente, pero se trata de un desarrollo relativamente complejo que precisa de conocimientos de cursos bastante avanzados.

Con el ejemplo que vamos a ver, pretendemos poder explicar algunos conceptos de la termodinámica desde la naturaleza atómica de la materia, enlazándolos con la mecánica más básica de los primeros cursos de física.

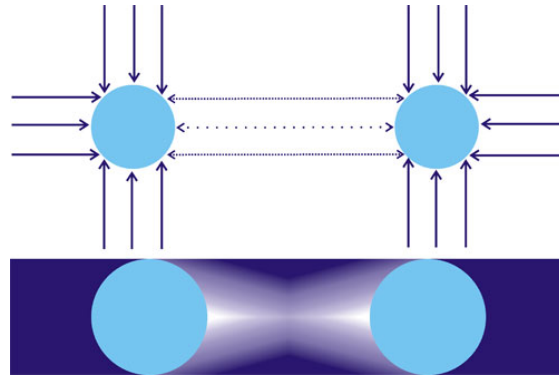
Para ello veremos un breve desarrollo de lo necesario de física estadística comparándolo con un desarrollo computacional de los mismos conceptos.

#### 4.1.1. Precedentes históricos: Teoría de la gravitación de Le Sage

Nicolas Fatio propone en 1660 una simpática teoría cinética de la gravitación.

Le Sage continúa su trabajo y da nombre al modelo. Según su teoría, existen unas ciertas particulillas en contante movimiento y velocidades aleatorias que empujan los cuerpos. La masa de los cuerpos determina su densidad (vista ésta como si los cuerpos fuesen de carácter esponjoso) y hace que más o menos corpúsculos puedan atravesarlos.

Se trata de un intento de explicar las leyes de Newton de atracción universal con sólo una concepción atomista de la materia que logra explicar la ley inversa del cuadrado pero recurre a feos argumentos “infinitos” para explicar la dependencia con la masa.



**Figura 4.1:** Los corpúsculos son apantallados en una dirección y se descompensan los momentos transferidos por colisiones.

No obstante, esta teoría una vez falseada, inspiraría la teoría cinética de la termodinámica que sí funcionó.

#### 4.1.2. Teoría cinética

La teoría cinética de los gases explica el comportamiento macroscópico de estos a partir de sus propiedades microscópicas.

##### Hipótesis cinemáticas

Para poder aplicar esta teoría deben cumplirse una serie de hipótesis en nuestro sistema:

- El sistema ha de ser lo suficientemente grande para considerarlo homogéneo. Cada partícula se visualiza como una esfera maciza en movimiento. Las partículas son suficientemente pequeñas comparadas con las distancias intermoleculares.
- Se supone que no existen fuerzas intermoleculares salvo en el instante de la colisión.
- Las moléculas se desplazan libremente chocando elásticamente entre sí.

Veamos a continuación cuáles serían los primeros pasos a la hora de simular este modelo:

- Construyamos una caja cúbica de lado 8 con cuidado de que las dimensiones de las esferas y las paredes se respeten:

```
thk = 0.3
side = 4.0
s2 = 2*side + thk
s3 = 2*side - thk
wallR = box (pos=vector( side, 0, 0), length=thk, height=s2,
            width=s3, color = color.green)
```

```
wallL = box (pos=vector(-side, 0, 0), length=thk, height=s2,
            width=s3, color = color.green)
wallB = box (pos=vector(0, -side, 0), length=s3, height=thk,
            width=s3, color = color.green)
ball=sphere(radius=0.5)
```

- Las esferas se moverán de manera rectilínea mientras nada las haga cambiar de opinión:

```
ball.pos = ball.pos + ball.velocity*timestep
```

- Se producirán choques elásticos con las paredes

```
if ball.x > 4:
    ball.velocity.x = -ball.velocity.x
    ball.x=2*4-ball.x
```

- En los choques entre partículas se producirá un intercambio de momentos:

```
distance=mag(ball_list[i].pos-ball_list[j].pos)
if distance < 2*ball_radius :
    direction=norm(ball_list[j].pos-ball_list[i].pos)
    vi=dot(ball_list[i].velocity, direction)
    vj=dot(ball_list[j].velocity, direction)
    exchange=vj-vi
    ball_list[i].velocity=ball_list[i].velocity + exchange*
        direction
    ball_list[j].velocity=ball_list[j].velocity - exchange*
        direction
    overlap=2*ball_radius-distance
    ball_list[i].pos=ball_list[i].pos - overlap*direction
    ball_list[j].pos=ball_list[j].pos + overlap*direction
```

En el sistema habrá dos magnitudes trascendentales:

**Frecuencia de colisión:** Frecuencia con que se producen colisiones en el sistema.

$$\nu = \frac{n_{col}}{t \cdot N}$$

**Recorrido libre medio:** Longitud media recorrida por una molécula entre colisiones.

$$\lambda = \frac{v_m}{\nu}$$

Durante una simulación, son valores que podemos calcular trivialmente mientras que en la realidad no son estimables por métodos directos.

### 4.1.3. Distribución de velocidades

Si consideramos el medio isótropo, la distribución de velocidades ha de serlo también.

Para visualizar esto, colocamos todos los vectores velocidad asociados a cada molécula en un mismo punto  $O$ .

```
velocidad=arrow(pos=(9,2,0), shaftwidth=0.1)
velocidad.axis=ball.velocity
```

Donde los vectores velocidad cortan a una esfera de radio  $R$  centrada en  $O$ , los llamaremos puntos figurativos. Su distribución debe ser uniforme:

$$n = \frac{N}{S} = \frac{N}{4\pi R^2} = \frac{dN}{dS} \rightarrow dN = \frac{N}{4\pi R^2} dS$$

En polares:

$$dS = R d\theta R \sin \theta d\phi \rightarrow dN = \frac{N}{4\pi} \sin \theta d\theta d\phi$$

$dN$  es el número de moléculas con velocidad entre  $\theta$  y  $\theta + d\theta$  y entre  $\phi$  y  $\phi + d\phi$ . Lo llamaremos  $d^2 N_{\theta,\phi}$

### 4.1.4. Magnitud de las velocidades moleculares

Separando por rangos de velocidad, obtendremos esferas concéntricas de puntos figurativos.

Llamemos  $dN_v$  a las moléculas con velocidades comprendidas entre  $v$  y  $v + dv$ .

Finalmente, podemos clasificar las moléculas como moléculas  $d^3 N_{\phi,\theta,v}$  :

$$d^3 N_{\phi,\theta,v} = \frac{dN_v}{4\pi} \sin \theta d\theta d\phi \rightarrow d^3 n_{\phi,\theta,v} = \frac{dn_v}{4\pi} \sin \theta d\theta d\phi$$

### 4.1.5. Magnitudes

#### Presión: choques contra una pared

La presión es efecto de los choques de las moléculas del gas contra una pared.

- Consideremos un elemento de pared  $dS$
- Vamos a contabilizar las moléculas que chocan contra la pared en función de sus velocidades, es decir de qué tipo sean ( $v, \phi, \theta$ ).
- Para que una molécula de tipo  $\theta, \phi, v$  choque con  $dS$  en un intervalo de tiempo  $d\tau$ , debe encontrarse justo antes de ese intervalo en un cilindro oblicuo de generatriz  $L = v d\tau$  inclinado según los ángulos  $\phi, \theta$ .
- Las moléculas que son de ese tipo:

$$d^3 n_{\theta,\phi,v} = \frac{dn_v}{4\pi} \sin \theta d\theta d\phi$$

- El volumen del cilindro:

$$dV = v d\tau \cos \theta dS$$

- Y por tanto, las moléculas de este tipo que chocan con  $dS$ :

$$d^3 n_{\theta, \phi, v} dV = \left( \frac{dn_v}{4\pi} \sin \theta v \cos \theta d\theta d\phi \right) d\tau dS$$

Para totalizar, integramos a  $0 < \theta < \pi/2$  y  $0 < \phi < 2\pi$ :

$$v \frac{dn_v}{4\pi} \int_0^{\pi/2} \sin \theta \cos \theta d\theta \int_0^{2\pi} d\phi = v \frac{dn_v}{4}$$

Ahora debemos integrar en velocidades.

- Habrá  $N_i$  partículas con velocidad entre  $v_i$  y  $v_i + dv_i$

$$\bar{v} = \frac{\sum_i N_i v_i}{\sum_i N_i} = \frac{\sum_i N_i v_i}{N}$$

Por unidad de volumen:

$$\overline{v_V} = \sum n_i v_i = n \bar{v}$$

En el límite de muchas partículas podemos considerar el espectro de velocidades continuo:

$$\overline{v_V} = \int_0^{\infty} v dn_v = n \bar{v}$$

Finalmente, el número de choques por unidad de área y tiempo:

$$n_{col} = \frac{1}{4} \int_0^{\infty} v dn_v = \frac{1}{4} n \bar{v}$$

En la simulación podemos contabilizar uno a uno los choques en el momento que se producen, por lo que todo este tedioso proceso de integración no es en absoluto necesario.

#### 4.1.6. Choques elásticos

Vamos a contabilizar el efecto de la presión como un intercambio de momento entre las paredes y las moléculas.

$$dp = p_f - p_i = 2mv \cos \theta$$

De nuevo procedemos a una integración en  $\phi$  y  $\theta$  análoga a la anterior, con la que la variación de la cantidad de movimiento debido a moléculas tipo  $\phi, \theta, v$ :

$$2mv \cos \theta \left( \frac{dn_v}{4\pi} dS v d\tau \sin \theta \cos \theta d\theta d\phi \right)$$

Integramos:

$$\frac{1}{3}m dS d\tau \int_0^\infty v^2 dn_v = dp = d\bar{F} d\tau$$

La variación de la cantidad de movimiento es la acción de una fuerza durante un intervalo de tiempo.

La presión es el resultado de la fuerza por unidad de superficie:

$$\bar{P} = \frac{d\bar{F}}{dS} = \frac{1}{3}m \int_0^\infty v^2 dn_v$$

Introduciendo aquí los resultados que habíamos obtenido para la integración en velocidades:

$$P = \bar{P} = \frac{1}{3}mn\bar{v}^2$$

que es la expresión cinética de la presión.

Puede ser bastante ilustrativo simular nuestra caja con una pared superior móvil con una cierta masa y que tienda a caer por efecto de la gravedad intercambiando momento con las esferas duras (moléculas del gas). De nuevo, con la simulación nos libramos del tedioso cálculo integral y podemos obtener resultados cualitativos que reflejen perfectamente gran parte de la fenomenología.

Veamos a continuación un ejemplo completo y sencillo que simule este caso:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from visual import *
from random import uniform
scene=display()
scene.title='Estadistica_de_velocidades'
scene.autoscale=0
scene.center=(3.5,0,0)
# Paredes
thk = 0.3
side = 4.0
s2 = 2*side + thk
s3 = 2*side - thk
wallR = box (pos=vector( side, 0, 0), length=thk, height=s2,
             width=s3, color = color.green)
wallL = box (pos=vector(-side, 0, 0), length=thk, height=s2,
             width=s3, color = color.green)
wallB = box (pos=vector(0, -side, 0), length=s3, height=thk,
             width=s3, color = color.green)
wallBK = box(pos=vector(0, 0, -side), length=s2, height=s2,
             width=thk, color = (0.7,0.7,0.7))
# La tapa superior es diferente
```

```
top=side*3/4
wallT = box (pos=vector(0, top, 0), length=s3, height=thk,
            width=s3, color = color.yellow)
```

Las constantes las ajustaremos de manera que se muestre una simulación cualitativa armoniosa aunque no sea cuantitativamente realista.

Para hacer algo cuantitativamente realista, el número de partículas debería dispararse salvajemente ( $\approx > 10^{23}$ ) para lo que necesitaríamos una potencia de cálculo considerable y unos algoritmos optimizados. No obstante, veremos que podemos sacar buenas conclusiones con esta aproximación.

```
no_particles=40.
ball_radius=0.5
maxpos=side-.5*thk-ball_radius
maxv=1.0
ball_list=[]
velocidad_list=[]
velcolt=0.
mball=1.
mwall=5.
grav=0.5
```

Lo natural en Python es tratar a cada una de las partículas como elementos de una lista:

```
for i in arange(no_particles):
    ball=sphere(color=color.green, radius=ball_radius)
    ball.pos=maxpos*vector(uniform(-1,1), uniform(-1,1), uniform
        (-1,1))
    ball.velocity=maxv*vector(uniform(-1,1), uniform(-1,1),
        uniform(-1,1))
    ball_list.append(ball)
    velocidad=arrow(pos=(9,2,0), shaftwidth=0.1)
    velocidad.axis=ball.velocity*2
    velocidad_list.append(velocidad)
    velcol=ball.velocity.x**2+ball.velocity.y**2+ball.velocity.
        z**2
    velcolt+=velcol
```

Asimismo introduciremos una leyenda que muestre algunos datos representativos.

```
leyenda=label(opacity=0, box=0, color=color.green, pos=(9, -5, 0),
            text='Velocidad_media=')
velcolt=(velcolt**(.5))/no_particles
fps=20
timestep = 1./fps
topp=-grav*timestep
```

```
tiempo=0
colisiones=0
colisionesw=0
colisionesp=0
colisiones=0
while tiempo < 600:
    rate(fps)
    tiempo+=timestep
    nparts=len(ball_list)
    i=0
    velcolt=0
    while i < nparts :
        ball = ball_list[i]
        velocidad = velocidad_list[i]
        velocidad.axis=ball.velocity*2
```

Movemos las partículas

```
ball.pos = ball.pos + ball.velocity*timestep
velcol=ball.velocity.x**2+ball.velocity.y**2+ball.
    velocity.z**2
velcolt+=velcol
```

Pintamos las partículas de modo que su color nos de una idea de su velocidad:

```
ball.color=(velcol/maxv,0,1-velcol/maxv)
velocidad.color=ball.color
```

Detectamos colisiones con las paredes:

```
if ball.x > maxpos:
    ball.velocity.x = -ball.velocity.x
    ball.x=2*maxpos-ball.x
    colisionesw+=1
if ball.x < -maxpos:
    ball.velocity.x = -ball.velocity.x
    ball.x=-2*maxpos-ball.x
    colisionesw+=1
if ball.y > (top-.5*thk-ball_radius):
    ball.velocity.y = -ball.velocity.y
    ball.y=2*(top-.5*thk-ball_radius)-ball.y
    colisionesw+=1
topp=topp - ball.velocity.y*mball/mwall
```

Esta es la pared que cae libremente y por eso metemos aquí el intercambio de momento.



```

if ball.y < -maxpos:
    ball.velocity.y = -ball.velocity.y
    ball.y=-2*maxpos-ball.y
    colisionesw+=1
if ball.z > maxpos:
    ball.velocity.z = -ball.velocity.z
    ball.z=2*maxpos-ball.z
    colisionesw+=1
if ball.z < -maxpos:
    ball.velocity.z = -ball.velocity.z
    ball.z=-2*maxpos-ball.z
    colisionesw+=1
i+=1

```

Las colisiones entre partículas serán un poco más complicadas de computar:

```

for i in range(no_particles):
    for j in range(i+1,no_particles):
        distance=mag(ball_list[i].pos-ball_list[j].pos)
        if distance < 2*ball_radius :
            colisionesp+=1
            direction=norm(ball_list[j].pos-ball_list[i].
                pos)
            vi=dot(ball_list[i].velocity , direction)
            vj=dot(ball_list[j].velocity , direction)
            exchange=vj-vi
            ball_list[i].velocity=ball_list[i].velocity +
                exchange*direction
            ball_list[j].velocity=ball_list[j].velocity -
                exchange*direction
            overlap=2*ball_radius-distance
            ball_list[i].pos=ball_list[i].pos - overlap*
                direction
            ball_list[j].pos=ball_list[j].pos + overlap*
                direction

```

Ahora computamos el número de colisiones y otras magnitudes:

```

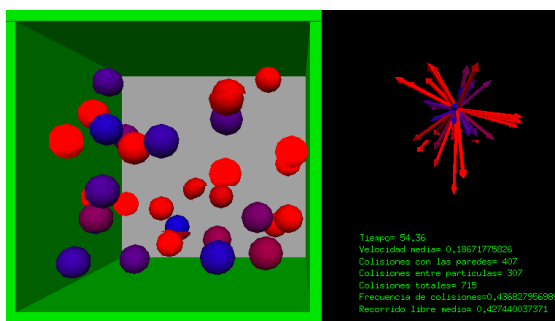
colisiones=(colisionesp+colisionesw)
frecuencia=colisiones/(tiempo*no_particles)
recorrido=0
volumen=(top+side)*4*(side**2)
temperatura=mball*velcolt**2/(3)
presion=mball*no_particles*(velcolt**2)/(3*volumen)
if not frecuencia == 0:

```

```

    recorrido=velcolt/frecuencia
    texto='_Tiempo=_'+str(tiempo)+'\n_Velocidad_media=_'+str(
    velcolt)+'\n_Colisiones_con_las_paredes=_'+str(
    colisionesw)+'\n_Colisiones_entre_particulas=_'+str(
    colisionesp)+'\n_Colisiones_totales=_'+str(colisiones)+'
    \n_Frecuencia_de_colisiones=_'+str(frecuencia)+'\n
    Recorrido_libre_medio=_'+str(recorrido)+'\n_Volumen=_'+
    str(volumen)+'\n_Temperatura=_'+str(temperatura)+'_K\n
    Presion=_'+str(presion)+' '
    leyenda.text=texto
    top+=topp*timestep
    topp+=-grav*timestep
    wallT.pos.y=top

```



**Figura 4.2:** Simulación de un gas por teoría cinética

El programa, aunque algo más extenso que los anteriores, es bastante sencillo. Está escrito con la idea de que todo resulte lo más lógico y natural posible para los no expertos sacrificando la optimización del código.

#### 4.1.7. Reversibilidad e irreversibilidad

Un concepto difícil de asimilar en los cursos de termodinámica, puede verse de un modo intuitivo con esta interpretación cinética.

Imaginemos que la tapa de nuestra caja no se mueve ahora libremente sino que asciende y desciende a una velocidad fijada por nosotros.

- Al descender la tapa el gas se calienta y al ascender se enfría.
- A bajas velocidades el proceso es cuasiestático y reversible, con lo que al volver el pistón a su posición inicial las constantes del sistema vuelven a sus valores iniciales.
- A altas velocidades se pierde la reversibilidad del proceso y el gas se va calentando paulatinamente.

- Esta pérdida de reversibilidad se puede ver en una sencilla gráfica de volumen frente a temperatura.
- Se puede relacionar esto con el número de colisiones de partículas con la tapa a lo largo del proceso de subida y de bajada.

En este siguiente ejemplo vamos a mover manualmente la tapa superior de modo que intercambie momento con las partículas de la caja calentándolas o enfriándolas.

El ejemplo está un poco más optimizado[7] y programado de una forma un poco más elegante. Además, se incluye una ventana con una gráfica que refleja (cualitativamente) la variación de temperatura con el volumen y un pequeño selector de velocidades utilizando tkinter.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from visual import *
from random import uniform
from visual.graph import *
from Tkinter import *

scene.autoscale=0
scene.center=(3.5,0,0)

class Caja:
    "La_caja_donde_meteremos_las_bolas"
    def __init__(self, caja):
        "Funcion_inicial"
        self.graph1 = gdisplay(x=0, y=0, width=300, height=150,
            title='T_vs._V', xtitle='V', ytitle='T',
            xmax=1100., xmin=450., ymax=10000, ymin=0,
            foreground=color.black, background=color.white)
        self.plotea = gcurve(gdisplay=self.graph1, color=color.
            blue)

        self.dialogo()
        self.variables()
        self.paredes()
        self.crearbolas()
        self.paso()
        self.coltop()
        self.colisiones()
        self.redibuja()
        self.idlecallback()
```

```

def dialogo ( self ) :
    "Dialogo_tkinter"
    self.frame=Frame(root)
    self.frame.pack()
    self.tapavel = Scale(self.frame, from_=0, to=50,
        resolution=1, label="Velocidad_Piston", command=self
        .changevtapa, orient=HORIZONTAL)
    self.tapavel.set(0)
    self.tapavel.pack()
    self.stbutton = Button(self.frame, text="Start",
        command=self.start)
    self.leyenda=label(opacity=0,box=0,color=color.green,
        pos=(10,-5,0),text='')
    self.stbutton.pack(side=BOTTOM)
    self.colsm=0
    self.colsp=0
def variables ( self ) :
    self.temperatura=1000
    self.paused=1
    self.vtapa=0
    self.timeout=10
    self.no_particles=20
    self.maxV=400
    self.side = 5.0
    self.tapatop=self.side
    self.tapabottom=0
    self.ballradius=0.4
    self.thk = 0.3
    self.dt = self.ballradius/self.maxV /2.
    self.fps=50
    self.acceleration=vector(0,-98,0)
    self.molecularmass=1 #kg
    self.Pressure=2.5 #kgm-2
    self.ltriang=fromfunction(lambda i,j: less_equal(j,i),[
        self.no_particles ,self.no_particles ])
def paredes ( self ) :
    "Creamos_las_paredes_y_las_dibujamos"
    self.maxpos=self.side-.5*self.thk-self.ballradius
    self.D2=(2*self.ballradius)**2
    self.s2 = 2*self.side + self.thk
    self.s3 = 2*self.side - self.thk
    wallR = box (pos=( self.side , self.side/2 , 0), length=
        self.thk , height=self.s2+2*self.side , width=self.s3 ,
        color = color.red)

```

```

wallL = box (pos=(-self.side , self.side/2, 0), length=
            self.thk, height=self.s2+2*self.side, width=self.s3,
            color = color.red)
wallB = box (pos=(0, -self.side, 0), length=self.s3,
            height=self.thk, width=self.s3, color = color.blue)
self.wallT = box (pos=(0, self.side+0.1 , 0), length=
            self.s3, height=self.thk, width=self.s3, color =
            color.blue)
wallBK = box(pos=(0, self.side/2, -self.side), length=
            self.s3+2*self.thk, height=self.s2+2*self.side,
            width=self.thk, color = (0.7,0.7,0.7))
self.wallT.mass=2 #self.Pressure/(self.s2**2)
self.wallT.velocity=0

```

```

def crearbolas(self):
    "Listas_de_bolas_en_posiciones_aleatorias_y_velocidades
     _en_un_rango"
    self.balls=[]
    self.plist=[]
    self.poslist=[]
    self.hlist=[]
    self.flechas=[]
    for i in arange(self.no_particles):
        ball = sphere(color = color.green, radius=self.
            ballradius)
        flecha = arrow(pos=(10,3,0), shaftwidth=0.1, color=
            color.white)
        p=[self.maxV*uniform(-1,1), self.maxV*uniform(-1,1),
            self.maxV*uniform(-1,1)]
        self.plist.append(p)
        position=[self.maxpos*uniform(-1,1), self.maxpos*
            uniform(-1,1), self.maxpos*uniform(-1,1)]
        self.poslist.append(position)
        ball.pos=vector(position)
        self.balls.append(ball)
        self.flechas.append(flecha)
        self.varray=array(self.plist)
        self.posarray=array(self.poslist)

def paso(self):
    self.posarray=self.posarray+self.varray*self.dt
    self.wallT.y=self.wallT.y+self.wallT.velocity*self.dt
    self.maxheight=self.wallT.y-0.5*self.thk-self.
        ballradius

```

```

self.maxarray=array([self.maxpos, self.maxheight, self.
    maxpos])
self.varray=self.varray+self.acceleration*self.dt
if self.wallT.y > self.tapatop:
    self.graph1.ymax=self.temperatura
    self.wallT.velocity=-self.vtapa
elif self.wallT.y < self.tapabottom:
    self.wallT.velocity=self.vtapa

def coltop(self):
    "Computa_colisiones_con_la_tapa"
    self.hit=greater(self.posarray[:,1], self.maxheight)
    self.hitlist=sort(nonzero(self.hit))
    for i in self.hitlist:
        self.varray[i,1]=self.varray[i,1]-2*self.wallT.
            velocity*self.wallT.mass/self.molecularmass

def colisiones(self):
    "Colisiones"
    putmask(self.varray, greater(self.posarray, self.maxarray
        ),-self.varray)
    putmask(self.posarray, greater(self.posarray, self.
        maxarray), 2*self.maxarray-self.posarray)
    putmask(self.varray, less_equal(self.posarray,-self.
        maxpos),-self.varray)
    putmask(self.posarray, less_equal(self.posarray,-self.
        maxpos),-2*self.maxpos-self.posarray)

    separation=self.posarray-self.posarray[:,NewAxis]
    sepmag2=add.reduce(separation*separation,-1)
    putmask(sepmag2, self.ltriang, 4*self.D2)
    hit=less_equal(sepmag2, self.D2)
    hitlist=sort(nonzero(hit.flat))

    for ij in hitlist:
        i, j = divmod(ij, self.no_particles)
        sepmag=sqrt(sepmag2[i, j])
        direction=separation[i, j]/sepmag
        pi=dot(self.varray[i], direction)
        pj=dot(self.varray[j], direction)
        exchange=pj-pi
        self.varray[i]=self.varray[i]+exchange*direction

```

```

self.varray[j]=self.varray[j]-exchange*direction

overlap=2*self.ballradius-sepmag
self.posarray[i]=self.posarray[i]-overlap*direction
self.posarray[j]=self.posarray[j]+overlap*direction

def redibuja(self):
    for i in arange(len(self.balls)):
        self.balls[i].pos=self.posarray[i]
        self.flechas[i].axis=vector(self.varray[i])/self.
            maxV
    #colisiones=(colisionesp+colisionesw)
    #frecuencia=colisiones/(tiempo*no_particles)
    #recorrido=0
    volumen=(self.wallT.pos.y+self.side)*4*(self.side**2)
    velmean=sum(mag2(self.varray))/self.no_particles
    self.temperatura=self.molecularmass*velmean**2*10**(-8)
        /(3)
    presion=0#mball*no_particles*(velcolt**2)/(3*volumen)
    #if not frecuencia == 0:
        #recorrido=0#velcolt/frecuencia
    texto=' _Volumen=_ ' +str(round(volumen,4))+'\n_
        Temperatura=_ ' +str(round(self.temperatura,4))+', '
    self.leyenda.text=texto
    self.plotea.plot(pos=(volumen, self.temperatura))

def idlecallback(self):
    "Esto_hace_el_loop"
    if self.paused == 0:
        self.paso()
        self.coltop()
        self.colisiones()
        self.redibuja()
    self.stbutton.after(self.timeout, self.idlecallback)

def changevtapa(self, event):
    self.tapa="piston"
    self.vtapa=self.tapavel.get()
    print self.vtapa
def start(self):
    if self.paused == 0:
        self.paused=1
    else:
        self.paused=0

```

```

root = Tk()
app = Caja(root)
root.mainloop()

```

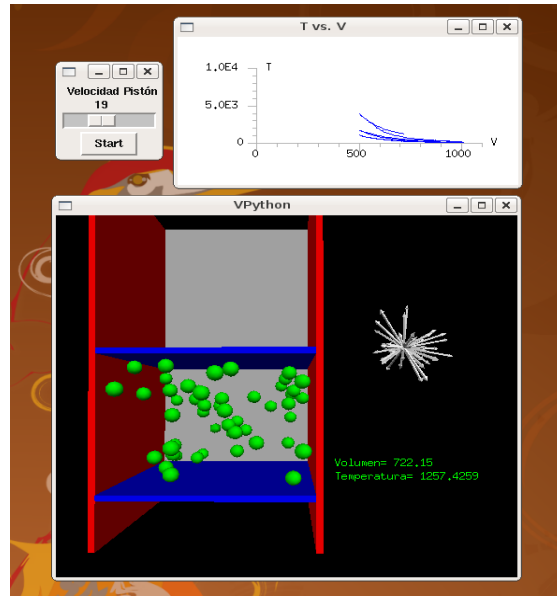


Figura 4.3: Simulación reversibilidad e irreversibilidad

#### 4.1.8. Conclusiones

El paso del mundo macroscópico al microscópico es un pilar fundamental desde los primeros años de aprendizaje de la física.

Si bien la física estadística corresponde a etapas superiores del aprendizaje, construir un modelo de esferas rígidas y choques elásticos para simular cualitativamente algunas fenomenologías macroscópicas de la termodinámica puede ser muy instructivo.

Una vez construido un modelo simple de detección de colisiones, se puede pedir al alumno que analice las variables del sistema, dibuje el espectro de velocidades o modifique las condiciones de la caja integrando, como en este ejemplo, paredes móviles.

## 4.2. Breve introducción a los métodos Montecarlo

Los métodos Montecarlo son métodos de resolución de problemas físicos mediante la simulación de variables aleatorias. Su nombre les viene por la analogía que tienen con el funcionamiento de un casino, de los que el más famoso es el de Montecarlo.

Debido a que hay muchos problemas que no pueden atacarse analítica ni numéricamente, tienen una indiscutible relevancia en la física actual. La creciente potencia y



velocidad de cálculo de los ordenadores los ha convertido en los últimos años en una de las herramientas básicas de simulación e investigación.

La simulación de crecimiento de dendritas mediante el método de agregación limitada por difusión puede ser un primer contacto con este tipo de métodos computacionales.

#### 4.2.1. Simulación de crecimiento de dendritas

Muchas estructuras naturales tienen forma de dendritas. Por ejemplo, el crecimiento de cristales bajo determinadas condiciones (copo de nieve), los relámpagos, las grietas, algunos líquenes, la hiedra creciendo por una pared...

Hay muchos métodos para simular estos fenómenos, de los que veremos uno de los más simples.

##### Agregación limitada por difusión

El método de “ALD” simula cómo una sustancia en disolución va precipitándose formando una cierta estructura cristalina siguiendo básicamente las siguientes reglas:

- Se dejan caer partículas una a una desde el margen superior con movimiento aleatorio (browniano).
- Las partículas rebotan en el borde superior y en los laterales.
- Cuando las partículas tocan el borde inferior se quedan pegadas.
- Cuando una partícula toca a otra se queda pegada.

Las reglas las hemos planteado para una precipitación sobre una superficie, con lo que obtendremos algo que nos recuerda a algas creciendo, pero podríamos plantearlas con simetría esférica (con una semilla) con lo que el resultado podría parecerse más a un copo de nieve.

Es fascinante lo naturales que nos resultan los patrones de crecimiento obtenidos.

##### Movimiento Browniano

Imaginemos (podríamos implementarlo fácilmente) la trayectoria de una de las partículas del gas en la simulación anterior ignorando la existencia del resto de partículas. Si el medio es lo suficientemente denso, su trayectoria nos parecerá completamente aleatoria. Si la mirásemos de cerca la veríamos como formada por pequeños tramos rectilíneos y bruscos cambios de dirección (choques). Estos tramos rectilíneos tendrían una cierta longitud promedio característica (recorrido libre medio).

Este tipo de movimiento fue estudiado por Jan Ingenhousz en 1785 y posteriormente por Robert Brown (a quien debe el nombre) en 1827 observando el movimiento de partículas microscópicas en el seno de un fluido. En 1905 Albert Einstein le da una descripción matemática demostrando definitivamente la teoría atómica y abriendo las puertas a la física estadística.

Aunque la matemática tras este movimiento es muchísimo más compleja, vamos a describirlo en una primera aproximación como un proceso puramente aleatorio de paso constante:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
from visual import *
from random import *
scene=display()
scene.title="Dendrite_growing"
Dimension=2
```

Vamos a hacer el programa capaz de simular el proceso en dos o tres dimensiones cambiando sólo una variable (*Dimension*). Por otro lado, la longitud  $d$  de los tramos rectilíneos está relacionada con el recorrido libre medio.

```
d = 1
ncm=100
nc=0
```

El valor *ncm* marca un límite en la cantidad de tramos de 1000 puntos simulados. *Visual Python* redibuja las curvas definidas con más de 1000 puntos eliminando puntos intermedios<sup>1</sup>. Como eso no nos interesa, dividiremos nuestra trayectoria en tramos de 1000 puntos. Así,  $ncm = 100$  implica que simularemos 100000 pasos aleatorios.

```
part=sphere(color=color.red)
ray=curve(color=color.white)
part.pos = [0,0,0]
if Dimension == 2:
    part.pos.z=0
ray.pos=part.pos
while nc < ncm :
    if Dimension == 3:
        dd = [randint(-1,1),randint(-1,1),randint(-1,1)]
    elif Dimension == 2:
        dd = [randint(-1,1),randint(-1,1),0]
    part.pos += dd
```

Como vemos, los pasos son saltos entre posiciones vecinas en una red cuadrada o cúbica.

```
ray.append(part.pos)
if len(ray.pos) > 1000:
    nc=nc+1
    curve(pos=ray.pos,color=color.white)
    ray.pos=part.pos
```

<sup>1</sup>No ya en las últimas versiones en desarrollo, con lo que ese apaño dejará de ser necesario.

En este último condicional es donde hacemos ese pequeño truco para evitar que *Visual Python* elimine puntos intermedios.

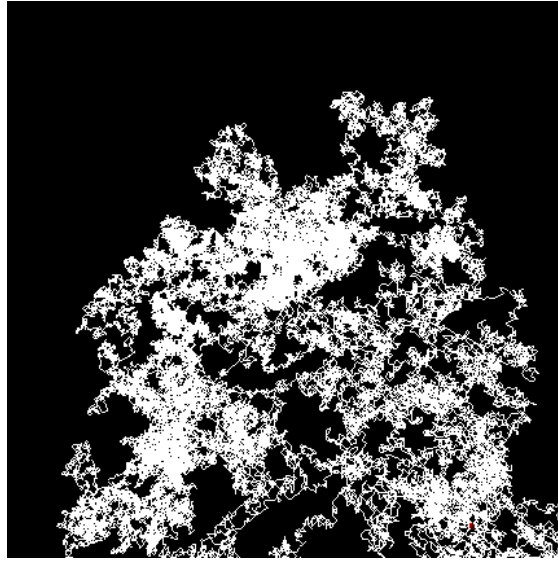


Figura 4.4: Simulación de movimiento browniano

#### Implementación del método ALD:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
from visual import *
from random import *
scene=display()
scene.title="Dendrite_growing"
Dimension=2
ignorefrom=500
```

Esa cifra *ignorefrom* la vamos a utilizar para acelerar un poco los cálculos. Para la detección de colisiones, el sistema tendrá en cuenta sólo las últimas 500 partículas depositadas de modo que las capas que se han ido quedando más profundas son ignoradas. Hasta qué punto es buena esta aproximación dependerá del tamaño de la escena que queramos simular.

```
maxparts = 10000
d = 1
alto=25
ancho = alto
parts = 0
altura=1-ancho
```

El parámetro altura es otra optimización del código. Si disparamos la partícula siempre desde el tope de la caja tardará un cierto tiempo en llegar a la zona con partículas. Como estamos tratando aleatoriamente la posición inicial, es perfectamente válido soltar las partículas desde una posición cercana al suelo e incluso hacerlas rebotar en un techo inferior.

```
scene.range=(ancho+2,alto+2,ancho+2)
part=sphere(color=color.red)
ray=curve(color=color.white)
part.visible=1
dentro=1
```

Simularemos hasta que se alcance el número máximo de partículas determinado o el cluster llegue al límite superior de la caja.

```
while ( part.pos.y < alto-1 ) or ( parts < maxparts ) :
    parts = parts + 1
    part.pos = [randint(-ancho, ancho), altura, randint(-ancho,
        ancho)]
    if Dimension == 2:
        part.pos.z=0
    ray.pos=part.pos
```

Mientras la partícula no ocupe la posición de otra o toque el suelo, se moverá aleatoriamente rebotando en las paredes:

```
while ( part.pos.x, part.pos.y, part.pos.z ) not in listaclust
    and part.pos.y >= -alto :
    if part.pos.x > ancho :
        dx=randint(-1,0)*d
        part.pos.x += dx
    elif part.pos.x < -ancho :
        dx=randint(0,1)*d
        part.pos.x += dx
    elif 1 :
        dx=randint(-1,1)*d
        part.pos.x += dx
    if Dimension == 3:
        if part.pos.z > ancho :
            dz=randint(-1,0)*d
            part.pos.z+=dz
        elif part.pos.z < -ancho :
            dz=randint(0,1)*d
            part.pos.z+=dz
```

```

    elif 1 :
        dz=randint(-1,1)*d
        part.pos.z+=dz
elif 1 :
    dz=0
if part.pos.y > altura :
    dy=randint(-1,0)*d-0.5
    part.pos.y+=dy
elif 1 :
    dy=randint(-1,1)*d
    part.pos.y+=dy
ray.append(part.pos)
dd=(dx,dy,dz)

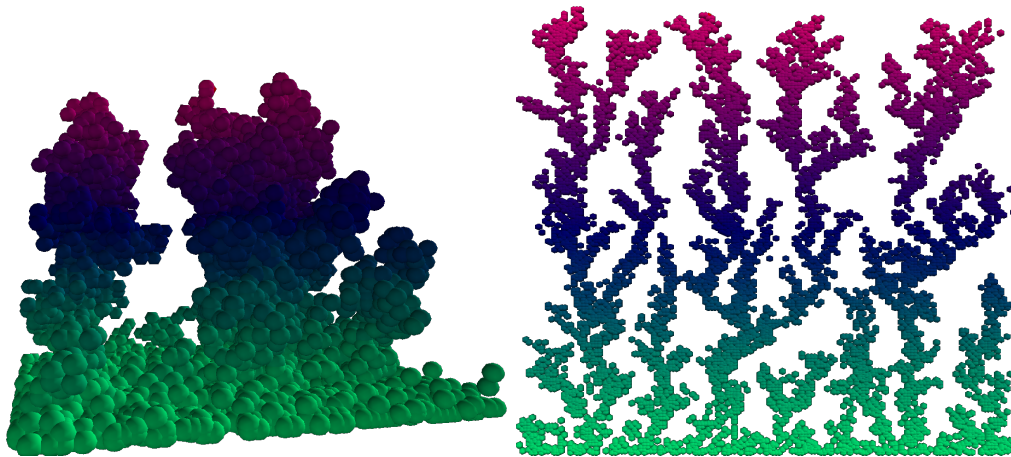
```

Cuando el número de partículas supera el que habíamos fijado para empezar a ignorar, eliminaremos la primera que se agregó a la lista.

```

if parts > ignorefrom :
    listaclust.remove(listaclust[0])
    listaclust.append((part.pos.x - dd[0] , part.pos.y - dd[1]
        , part.pos.z - dd[2]))
    sphere(pos=part.pos+dd, color=(part.pos.y/alto, -part.pos.y/
        alto, 0.5), radius=d*1.3)

```



**Figura 4.5:** Dendrita simulada en 2 y 3 dimensiones

Los diagramas obtenidos son fractales de dimensión 1.71



## 5 Jugando con Visual Python

### 5.1. Visual Pong

Quizá el videojuego más sencillo y adictivo que pueda escribirse sea el pong. No fue el primer videojuego de la historia pero sí el pionero en popularizar los videojuegos. Basado originalmente en el tenis de mesa (ping-pong), consta de sencillamente un recinto cuadrado con dos cursores (jugadores) a cada lado que hacen rebotar sobre ellos una pelota evitando que golpee el lateral que cada jugador defiende.

En nuestro caso, vamos a hacer una variante tridimensional para un sólo jugador. Tendremos un recinto cúbico y un cursor cuadrado (controlado por el mouse) para defender la cara inferior de la caída de la bola. La bola rebotará elásticamente en las paredes y nos ayudaremos de una línea que dibuje la trayectoria y de una guía vertical para visualizar el entorno tridimensional.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
from visual import *
scene=display()
scene.title='Visual_Pong'
scene.autoscale=0
db=0.1
ddb=0.02
level=0
lx=5
err=1
pared1=box(size=(0.1,10,10),pos=(-5,0,0),color=color.blue)
pared2=box(size=(0.1,10,10),pos=(5,0,0),color=color.blue)
pared3=box(size=(10,0.1,10),pos=(0,5,0),color=color.blue)
pareddelamuerte=box(size=(10,0.1,10),pos=(0,-5.4,0),color=color
    .red)
pared4=box(size=(10,10,0.1),pos=(0,0,-5),color=color.blue)

cursor=box(size=(2,0.1,2),pos=(0,-5,0),color=color.green)
ball=sphere(pos=(0,0,0),radius=1,color=color.white)
by=cylinder(pos=(0,-5,0),radius=0.1,color=color.blue,axis
    =(0,10,0))
trail=curve(color=color.yellow)
ballvx=0.87
```

```
ballvy=0.5
ballvz=0.35

dentro=1
while dentro :
```

La interacción del usuario la realizaremos con el ratón mediante la proyección del mismo sobre la cara inferior del cubo.

```
    if demo == 0:
        temp = scene.mouse.project(normal=(0,1,0), point
            =(0,-5,0))
        if temp:
            cursor.pos = temp
```

Las colisiones con el cursor son las que delimitan si hemos acertado o perdemos.

```
    if ball.pos.y < -4:
        if cursor.pos.x < ball.pos.x+err and cursor.pos.x >
            ball.pos.x-err and cursor.pos.z < ball.pos.z+err and
            cursor.pos.z > ball.pos.z-err:
            ballvy=ballvy*(-1)
            ball.pos.y=-4
```

El control de niveles acelerará la bola para que el juego aumente en dificultad.

```
        if level < 15:
            db=db+ddb
            level=level+1
            trail.pos=ball.pos
        elif ball.pos.y < -4.1:
            label(pos=(0,0),text='Game Over\nYou have reached
                level'+str(level)+'\nPress Esc to exit.',
                border=10,color=color.red)
            ball.color=color.red
            dentro=0
```

Las colisiones con las paredes las hacemos como en el ejemplo de la simulación del gas.

```
    elif ball.pos.y > 4:
        ballvy=ballvy*(-1)
        ball.pos.y= 4
    elif ball.pos.z > 4:
        ballvz=ballvz*(-1)
        ball.pos.z= 4
    elif ball.pos.z < -4:
        ballvz=ballvz*(-1)
```



```

    ball.pos.z= -4
elif ball.pos.x > 4:
    ball.vx=ball.vx*(-1)
    ball.pos.x= 4
elif ball.pos.x < -4:
    ball.pos.x= -4
    ball.vx=ball.vx*(-1)

ball.pos=ball.pos+(ball.vx*db, ball.vy*db, ball.vz*db)
by.pos=(ball.pos.x, -5, ball.pos.z)
trail.append(pos=ball.pos)

```

De nuevo para agregar un poco de dificultad al juego, se eliminan las ayudas para visualizar la posición tridimensional de la bola cada 5 niveles.

```

if level == lx:
    by.visible=0
    ball.color=color.orange
if level < 15 and level == lx+1:
    by.visible=1
    ball.color=color.white
    lx=lx+6
rate(50)

```

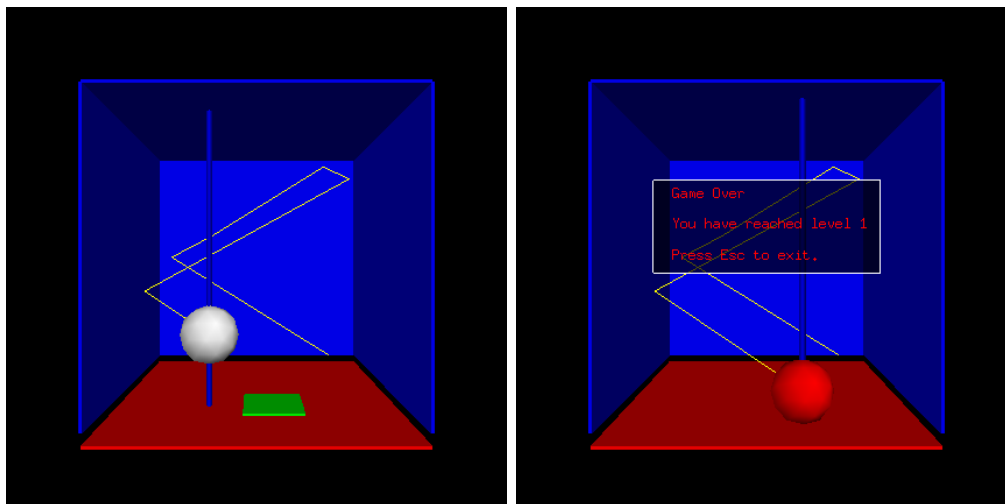


Figura 5.1: Visual Pong

## 5.2. Space Rebounder Racing

Vamos a llevar una jugabilidad similar a la anterior a un nuevo plano de originalidad en este nuevo juego.

El entorno consistirá esta vez en una esfera cuya coordenada x controlamos con el mouse para que se mueva rebotando entre losetas situadas en dos planos superior e inferior. Esas losetas irán apareciendo de manera aleatoria y moviéndose en dirección a la cámara de modo que el jugador tiene una cierta sensación de movimiento. Esa situación de movimiento la enfatizaremos con una serie de juegos de cámara y objetos a modo de decorado.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from visual import *
from random import *
scene.autoscale=0
scene.fov=pi/3.
scene.fullscreen=0
coche=frame(pos=(0,0,0))
suelo1=[]
suelo2=[]
cuerpo=sphere(color=color.green,frame=coche,radius=0.5,pos
              =(0,0,0))
n=0
side=20
origen=-60
final=20
o=0
thick=7.6
while o < final-origen-side:
    r=randint(-1,1)
    o=n*20
    losa1=box(size=(thick,0.1,side),pos=(r*4,-5,origen+n*side),
        color=(0.1*randint(0,10),0.1*randint(0,8)+0.2,0.1*
        randint(0,10)))
    losa2=box(size=(thick,0.1,side),pos=(r*4,5,origen+n*side),
        color=(0.1*randint(0,10),0.1*randint(0,8)+0.2,0.1*
        randint(0,10)))
    suelo1.append(losa1)
    suelo2.append(losa2)
    n+=1
paisaje=[]
planeta1=sphere(color=color.blue,pos=(-20,20,-50),radius=20)
```



Subimos niveles haciendo más veloz a la bola y más estrechas las losas para aumentar la dificultad.

```

if t > levelazo and demo == 0 :
    levelazo+=15
    level+=1
    thick=thick-dthick
    cochevel+=cochevel/10
    i=0
    while i < len(suelo1):
        suelo1[i].size=(thick,0.1,side)
        suelo2[i].size=(thick,0.1,side)
        i+=1
i=0
while i < len(suelo1):
    losa1=suelo1[i]
    losa2=suelo2[i]
    losa1.pos.z+=v*dt
    losa2.pos.z+=v*dt
    if losa1.pos.z > 20:
        r=randint(-1,1)
        losa1.color=(0.1*randint(0,10),0.1*randint(0,10)
            +0.2,0.1*randint(0,10))
        losa1.pos.x=r*4
        losa1.pos.z=origen
        losa1.size=(thick,0.1,side)
        r=randint(-1,1)
        losa2.color=(0.1*randint(0,10),0.1*randint(0,8)
            +0.2,0.1*randint(0,10))
        losa2.pos.x=r*4
        losa2.size=(thick,0.1,side)
        losa2.pos.z=origen
    i+=1
if coche.pos.y < -3.9 :
    if coche.pos.y < -4.5 :
        vidas-=1
        cochevel=abs(cochevel)
        if vidas == 2:
            colorin=color.yellow
        elif vidas == 1:
            colorin=color.red
        for i in coche.objects:
            i.color=colorin
    if vidas == 0:

```

```

gover.text='Game_over ... You_Reached_level_'+
str(level)+'\n_Press_s_to_start_bouncing_
again'
gover.color=color.red
while scene.fov > pi/3. :
    scene.fov-=dt/3
    rate(fps)
for i in coche.objects:
    i.color=color.green
demo=1
t=0
level=0
levelazo=10
thick=7.6
coche.pos=(0,0,0)
cochevel=10
dv=0.5
vidas=3

```

Rebotes en las losas y pérdida de vidas:

```

for i in suelo1 :
    if coche.pos.z > i.pos.z-side/2.-0.5 and coche.pos.
z < i.pos.z+side/2.+0.5 :
        if coche.pos.x > i.pos.x-thick/2.-0.5 and coche
.pos.x < i.pos.x+thick/2.+0.5 :
            cochevel=abs(cochevel)

elif coche.pos.y > 4 :
    if coche.pos.y > 4.5:
        vidas-=1
        cochevel=-abs(cochevel)
        if vidas == 2:
            colorin=color.yellow
        elif vidas == 1:
            colorin=color.red
        for i in coche.objects :
            i.color=colorin
    if vidas == 0:
        gover.text='Game_over ... You_Reached_level_'+
str(level)+'\n_Press_s_to_start_bouncing_
again.'
        gover.color=color.red
        while scene.fov > pi/3. :

```

```

        scene.fov==dt/3
        rate(fps)
        for i in coche.objects:
            i.color=color.green
        coche.pos=(0,0,0)
        demo=1
        t=0
        level=0
        levelazo=10
        thick=7.6
        cochevel=10
        dv=0.5
        vidas=3
    for i in suelo2 :
        if coche.pos.z > i.pos.z-side/2.-0.5 and coche.pos.
            z < i.pos.z+side/2.+0.5 :
            if coche.pos.x > i.pos.x-thick/2.-0.5 and coche
                .pos.x < i.pos.x+thick/2.+0.5 :
                cochevel=-abs(cochevel)

    if demo == 0 :
        coche.pos.y=coche.pos.y+cochevel*dt
        gover.text='Lives='+str(vidas)+'  _ _ _ level='+str(level)+'
        ,
    elif demo == 1 :
        coche.pos.y=coche.pos.y
        if scene.kb.keys:
            temp=scene.kb.getkey()
            if temp == 's':
                demo=0
                gover.color=color.white

```

Control del mouse

```

    if demo == 0:
        temp = scene.mouse.project(normal=(0,0,1), point
            =(0,-5,0))
        if temp:
            coche.pos.x = temp.x

```

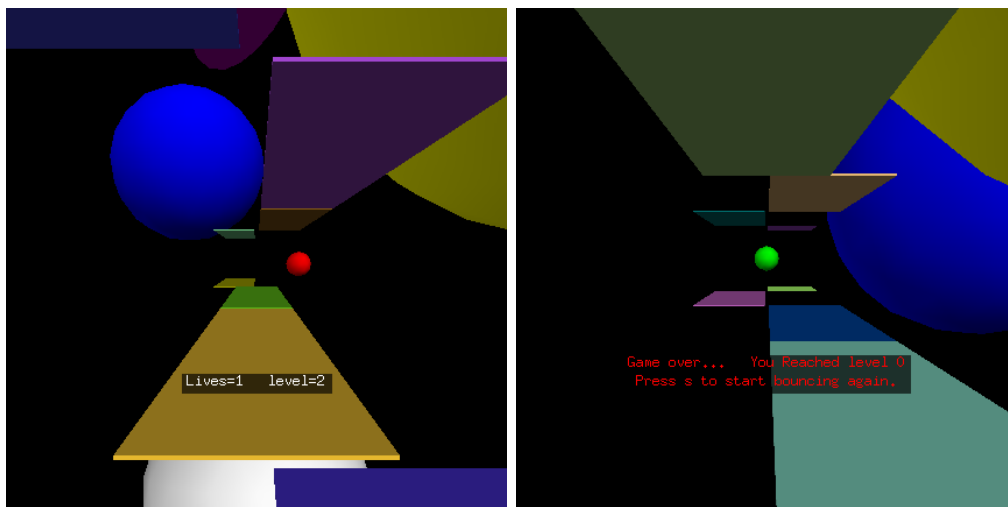


Figura 5.2: Space Rebounder Racing





## Bibliografía

- [1] Allen Downey. *Think Python: An introduction to software design*. Green Tea Press, 2008.
- [2] Chabay, R. W. and Sherwood, B. A. Bringing atoms into first-year physics. *American Journal of Physics*, (67):1045–1050, 1999.
- [3] Chabay, R. W. and Sherwood, B. A. *Research-based Reform of University Physics*, chapter Matter and Interactions. PER-Central, 2007. [1.1](#)
- [4] Chabay, R. W. and Sherwood, B. A. Computational physics in the introductory calculus-based course. *American Journal of Physics*, (76):307–313, 2008.
- [5] Peter Bowyer. *An investigation into teaching introductory programming to physics undergraduates*. 2003. [1.2](#)
- [6] Ruth Chabay, David Scherer, and Bruce Sherwood. *The Visual Module of VPython*. <http://www.vpython.org/webdoc/visual/index.html>. [3.1.2](#)
- [7] Sally Lloyd and Stephen Roberts. Using vpython to simulate a gas. <http://comptlsci.anu.edu.au/Tutorial-Gas/tute-gas.html>. [4.1.7](#)
- [8] Iu. Iú. Tarasiévich. *Simulación matemática y computacional*. URSS, 2004.

## *BIBLIOGRAFÍA*

# Historia

## 0.1 - 17 de octubre de 2008

- Se imparte el primer curso durante la VIII Semana de la Ciencia de Madrid.
- Ve la luz la primera versión para alqua de este documento.

Las siguientes tareas merecen atención, a juicio de los editores y autores:

- Mejorar y añadir comentarios al código
- Limpiar y corregir código
- Ampliar introducción a Python
- Desarrollar más las introducciones físicas
- Ampliar la colección de simulaciones
- Corregir ortografía



# Creative Commons Deed

## Reconocimiento - NoComercial - CompartirIgual 2.5

Usted es libre de:



copiar, distribuir y comunicar públicamente la obra



hacer obras derivadas

Bajo las condiciones siguientes:



Reconocimiento. Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciador (pero no de una manera que sugiera que tiene su apoyo o apoyan el uso que hace de su obra).



No comercial. No puede utilizar esta obra para fines comerciales.



Compartir bajo la misma licencia. Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

- Al reutilizar o distribuir la obra, tiene que dejar bien claros los términos de la licencia de esta obra.
- Alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor.
- Nada en esta licencia menoscaba o restringe los derechos morales del autor.

**Los derechos derivados de usos legítimos u otras limitaciones reconocidas por la ley no se ven afectados por lo anterior.**

Esto es un resumen legible por humanos del texto legal (la licencia completa) disponible en:

<http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es>

Aprenda cómo distribuir su obra utilizando esta licencia en:

<http://creativecommons.org/learn/licenses>



# El proyecto libros abiertos de Alqua

El texto que sigue es una explicación de qué es y cómo se utiliza un libro abierto y contiene algunas recomendaciones sobre cómo crear un libro abierto a partir de un documento de Alqua. Si estás leyendo estas páginas como anexo a otro documento, éste es casi con seguridad un *documento libre* de Alqua; libre en el sentido descrito en el [manifiesto de Alqua](#) y las [directrices para documentos libres de Alqua](#). Si has obtenido dicho documento en un centro público, como una biblioteca, entonces es además un *libro abierto* de Alqua.

## Qué son los libros abiertos

Los libros abiertos son ediciones impresas de los documentos libres de Alqua que se pueden obtener en las bibliotecas u otros centros públicos. La particularidad de los libros abiertos no reside en *qué contienen* (el contenido es el mismo que el de los libros descargados de la red) sino en *cómo pueden utilizarse*.

Al igual que los usuarios de Alqua a través de la red forman una comunidad de interés que aprende colectivamente leyendo los documentos, discutiendo sobre ellos y modificándolos para adaptarlos a propósitos muy variados, los lectores de una biblioteca constituyen también una comunidad. El ciclo de vida de un documento libre es de constante realimentación: las nuevas versiones son leídas, corregidas o quizá bifurcadas, lo que conduce a la publicación de nuevas versiones listas a su vez para un nuevo ciclo del proceso. ¿Por qué no abrir esa dinámica a la participación de comunidades que no se articulan en torno a la red?. No todos disponen del tiempo o los medios para participar efectivamente en el proceso de mejora de los documentos a través de la red, que es la aportación diferencial más importante de los libros libres respecto a los no libres. Por ello queremos poner a disposición de las bibliotecas *libros abiertos* que faciliten lo siguiente:

- El acceso de personas sin recursos informáticos al conocimiento que su estudio proporciona.
- La posibilidad de contribuir a la mejora de dichos documentos por parte de la amplísima comunidad de lectores de las bibliotecas, sin otro medio que un lápiz o una pluma.
- La formación de grupos de interés locales: compartir a través de un documento libre puede compartir su proceso de aprendizaje con personas interesadas por temas afines.

- La constitución, hasta en los centros que cuentan con una financiación más débil, de un fondo de documentos libres que cubra áreas del conocimiento que su presupuesto no permite afrontar.

## ¿Cómo puedo contribuir a los libros abiertos?

Sólo tienes que utilizarlos como si fuesen tuyos, pero recordando que compartes tu experiencia de aprendizaje con otras personas.

Por ejemplo, contrariamente a lo que harías con cualquier otro libro de la biblioteca puedes escribir en los márgenes de los libros abiertos tus propios comentarios: correcciones, aclaraciones, bibliografía relacionada... Intenta hacerlo ordenadamente, de modo que no interrumpa la lectura.

Si quieres compartir algún razonamiento más largo, puedes utilizar tus propias hojas e incorporarlas al final del documento, poniendo una nota donde corresponda. En este caso, no olvides firmar tu contribución con un nombre o seudónimo y, opcionalmente, una dirección de correo electrónico u otra forma de contacto.

Cualquiera que pueda participar a través de la red puede incorporar tus contribuciones a la versión que se distribuye en línea, con la ayuda de la comunidad de Alqua. De esta manera abrimos el mecanismo de colaboración a los lectores que no están acostumbrados al ordenador o prefieren no usarlo. La firma permite atribuir la autoría en el caso de que los cambios se incorporen y establecer contacto al respecto. Damos por hecho que al escribir tus aportaciones en un libro abierto estás de acuerdo con que sean libremente utilizadas (en el sentido descrito en las directrices para documentos libres ya mencionadas) y por lo tanto incorporadas a las sucesivas versiones digitales.

Los libros abiertos pueden ser editados de modo que se puedan separar sus hojas porque no hay inconveniente en que éstas sean fotocopiadas: no tenemos que usar la encuadernación como un modo de evitar la reproducción, puesto que no sólo no la prohibimos sino que animamos a ella. Por tanto, una vez que obtengas un ejemplar en préstamo puedes llevar contigo sólo la parte que estés utilizando.

Como lector, tu ayuda es necesaria no sólo para mejorar los documentos, sino para que existan: hace falta imprimir, encuadernar y donar a una biblioteca un documento libre de Alqua para que se convierta en un *libro abierto*.

Quienes tengan acceso a una impresora pueden ayudar a que los *libros abiertos* perduren en la biblioteca sustituyendo las partes deterioradas por el uso y actualizando periódicamente el documento impreso. Para facilitar la tarea a continuación proponemos un sistema de encuadernación modular.

## ¿Cómo puedo publicar un libro abierto?

Los pasos para publicar un libro abierto son los siguientes:

1. Imprimir la versión más actualizada del documento tal cual se distribuye en la página web de Alqua, <http://alqua.org>



2. Conseguir una encuadernación modular – sugerimos un archivador de anillas con una ventana o de portada transparente. Ello permite llevar consigo sólo la parte del libro que se está usando y añadir hojas con nuevas contribuciones.
3. Encuadernar el libro y situar el título, el autor y la clasificación decimal universal en su lomo y tapas.
4. Si puedes, adjuntar al archivador una copia del [CD-ROM de documentos libres de Alqua](#) .
5. Donarlo a la biblioteca y comunicar a Alqua la edición, escribiendo a [librosabiertos@alqua.org](mailto:librosabiertos@alqua.org) .

Se trata de un proceso sencillo al alcance tanto de particulares como de bibliotecas y otras instituciones, con un coste marginal que no se verá significativamente incrementado por la conservación y actualización puesto que se puede mantener la encuadernación y sustituir solamente las páginas impresas.

## En conclusión

El proyecto *libros abiertos*, consecuencia de los principios establecidos en el [manifiesto de Alqua](#) , persigue dotar a las bibliotecas de un fondo amplio y asequible de documentos libres y a la vez facilitar la participación de los usuarios en el proceso creativo del que son fruto.

Tu ayuda es esencial para que el proyecto alcance estos objetivos.

---

(C) Álvaro Tejero Cantero, 2003.



**Proyecto Physthones**  
*Simulaciones Físicas en Visual Python*  
Pablo M. García Corzo

descripción

Es un proyecto libre para construir una colección lo más amplia posible de simulaciones físicas tridimensionales en Visual Python insistiendo en el paradigma del software libre que iguala usuario y desarrollador. Pretende romper la barrera que separa el código del usuario final tendiendo un puente desde la física fundamental a la computacional asentando a su paso los pilares del álgebra y el cálculo diferencial.

requisitos

- Conocimientos básicos de física

<http://alqua.org/documents/physthones>

Aprende en comunidad - <http://alqua.org> <

**otros documentos libres**

Variedades, tensores y física - Óptica electromagnética - Ecuaciones diferenciales ordinarias - Introducción a la física cuántica, segunda parte - Redes y sistemas - Sistemas Operativos - Geometría simpléctica - Física del láser - Análisis funcional - Geografía general de España (en preparación).

<http://alqua.org/documents/>

alqua, **madeincommunity**