

# Manual de Python

Francisco Silva

João Gomes

Miguel Zilhão

Tiago Dinis

3 de Setembro de 2009



# Capítulo 1

## Introdução

Este manual foi escrito com o intuito de apoiar os projectos por nós dirigidos na Escola de Verão de Física, visto que estes passam por aprender o básico sobre Python - uma linguagem de programação simples e fácil de aprender.

Mas, porquê escrever mais um manual de Python? É verdade que na internet se encontram vários manuais de Python, no entanto estes manuais ora estão em inglês ora são demasiado avançados ora estão centrados em aspectos que não nos interessam... Por isso decidimos escrever um pequeno manual onde explicamos aquilo que necessitamos e da forma que nós queremos. No fim da escola este manual será actualizado e os alunos poderão utilizá-lo como referência caso queiram brincar um pouco mais com o Python.

### 1.1 O que é simulação

Suponhamos que estamos interessados em construir um carro que seja seguro. Precisamos de testar o comportamento do carro em vários tipos de acidentes. O modo mais directo de testar a sua segurança seria realizar uma colisão real, ou seja, metemo-nos no carro aceleramos e chocamos contra um obstáculo. Este método é incómodo pois só podemos fazer a experiência uma vez e nem podemos analisar os resultados.

E se no lugar de pessoas usássemos bonecos, os quais simulariam o condutor e os passageiros? É certamente melhor para a nossa integridade, e é uma das técnicas que usam os fabricantes de automóveis.

Mesmo assim temos um problema: para cada simulação precisamos dum carro, o que fica extremamente caro. Outra opção seria simular a situação dentro dum computador, poupando carros e bonecos. A estratégia passa por calcular todas as forças, velocidades e posições do carro e dos passageiros ao longo do acidente usando um computador, estimando no fim o dano causado nos vários passageiros. Podemos assim simular acidentes de forma muito mais barata e rápida e em inúmeras situações.

Claro que a simulação computacional é, em geral, menos fidedigna que o embate dum carro num obstáculo. Por isso usa-se as duas técnicas complementarmente: começa-se por simular no computador e numa fase mais avançada usa-se um protótipo.

Note-se que há imensas situações as quais só se podem estudar por simulação computacional: por exemplo a simulação da origem do universo ou nano-engrenagem<sup>1</sup>, até agora impossí-

---

<sup>1</sup>Podem ver no sítio <http://static.howstuffworks.com/flash/nanotechnology.swf> uma animação de uma engrenagem nanoscópica, composta por poucos átomos.

veis de fabricar.

## 1.2 Linguagens de programação

Claro que para dar as ordens ao computador precisamos de saber comunicar com ele, numa linguagem que ele compreenda. Desde os primórdios da informática (>1940s) que se tem vindo a criar inúmeras destas linguagens, chamadas linguagens de programação, cada uma com o seu propósito e sua estrutura. Dado isto, a escolha de uma linguagem de programação para uma dada tarefa não deve ser aleatória - imaginem que têm à vossa disposição um martelo e uma chave de fendas - é possível colocar um parafuso à martelada, mas é muito mais fácil com a chave.

Neste manual não vamos falar de várias linguagens, nem dos seus propósitos ou as diferenças entre elas - embora fosse interessante. Para os mais curiosos, recomendamos:

- Introdução às Famílias de Linguagens de Programação: Primeiro capítulo do livro *Modern Programming Languages: A Practical Introduction* (<http://www.fbeedle.com/mpl/76-7-02.pdf>);
- Uma lista e tabela de comparação do propósito usual de várias linguagens [http://en.wikipedia.org/wiki/Comparison\\_of\\_programming\\_languages](http://en.wikipedia.org/wiki/Comparison_of_programming_languages);
- Estatísticas sobre a popularidade de cada linguagem no mundo: <http://langpop.com/>.

No nosso caso optamos por uma linguagem chamada Python.

## 1.3 Porquê Python

Vejamos os nossos requerimentos para as nossas simulações:

- Precisamos de uma linguagem que nos permita fazer cálculos e manipular os resultados facilmente;
- Esta deve permitir-nos expor os resultados de forma conveniente - números, gráficos, animações, etc;
- Deve ser de acesso fácil e gratuito e funcione em qualquer computador ou sistema operativo;
- Precisamos de uma linguagem com uma sintaxe simples cujos elementos básicos possam ser ensinados em menos de 1 dia.

Embora existam várias linguagens que de certo modo preencham os 4 requerimentos, existe uma que achamos especialmente adequada: Python. Esta distingue-se especialmente por ter uma sintaxe clara e ter elementos básicos simples. Esta tem ainda a vantagem de possuir vastas bibliotecas, igualmente simples de usar, que nos permitem fazer gráficos, animações e interagir com o utilizador.

Dado isto, passemos à acção.

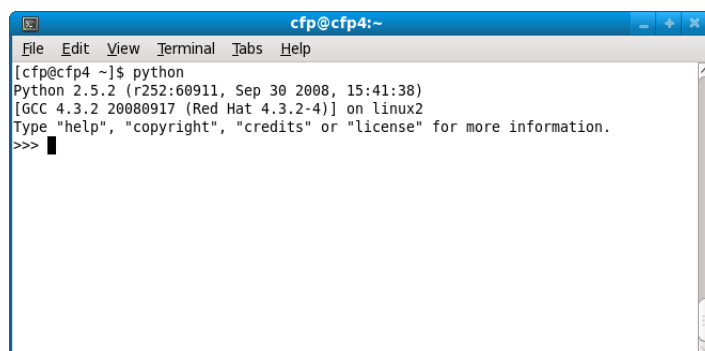
## Capítulo 2

# Rudimentos de Python

Neste capítulo vamos aprender a usar o básico do Python, com ele o aluno será capaz de fazer vários programas, simples mas poderosos. Supomos que o aluno usa Linux, as diferenças para outros sistemas operativos (Windows, MacOSX...) não são significativas.

### 2.1 Interpretador e Operações Básicas

O método mais simples de trabalhar com Python é através do interpretador. Este pode ser usado por vários caminhos: na linha de comandos, pela internet (<http://try-python.mired.org/>, <http://codepad.org/>), ou até dentro doutros programas. Na Escola de Física será usado na linha de comandos. Para abrir o interpretador, introduz-se *python* na linha de comandos.

A screenshot of a terminal window titled 'cfp@cfp4:~'. The window has a menu bar with 'File', 'Edit', 'View', 'Terminal', 'Tabs', and 'Help'. The terminal output shows the command '[cfp@cfp4 ~]\$ python' being executed. The response is 'Python 2.5.2 (r252:60911, Sep 30 2008, 15:41:38) [GCC 4.3.2 20080917 (Red Hat 4.3.2-4)] on linux2'. Below this, it says 'Type "help", "copyright", "credits" or "license" for more information.' and the prompt '>>>' is shown with a cursor.

O conceito do interpretador é simples, dá-se uma ordem e ele executa-a. Para sair do interpretador escreve-se *exit()* ou prime-se Control-d.

Começemos por usar o interpretador como uma calculadora. Temos as operações elementares da álgebra:

Operação	Símbolo
Adição	+
Subtração	-
Multiplicação	*
Divisão	/
Exponenciação	**
Resto	%

**Exercício 2.1.** Calcule  $4.5 \times 3.14 + 3.4^{2.1}$ . Reparem que o Python usa um ponto como separador decimal.

**Exercício 2.2.** Quanto vale  $3/2$ ? E  $3.01/2$ ?

**Exercício 2.3.** Quanto falta a 2354785 para ser um múltiplo de 7?

## 2.2 Variáveis

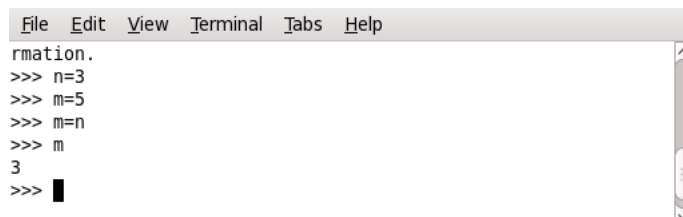
### 2.2.1 Variáveis numéricas

Em Python existem três tipos principais de variáveis numéricas:

- Os números inteiros, por exemplo  $n = 3$ ;
- Os números reais, por exemplo  $x = -3.14$ ;
- E os números complexos, como por exemplo  $z = 3.2 + 1j$ . Notem que o Python usa  $j$ , e não  $i$  como é habitual.

Caso não haja separador decimal o Python assume que o número é inteiro. Isto pode trazer problemas nas divisões, por exemplo  $1/2 = 0$ , mas  $1./2. = 0.5$ .

Quando se escreve  $n = 3$  está-se a guardar o valor 3 em  $n$ . Vejamos o exemplo seguinte:



```

File Edit View Terminal Tabs Help
rmation.
>>> n=3
>>> m=5
>>> m=n
>>> m
3
>>>

```

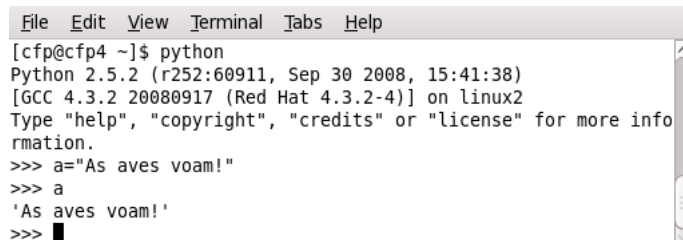
O que acontece:

1. Na primeira linha guardamos o valor 3 na variável  $n$ ;
2. Na segunda guardamos o valor 5 em  $m$ ;
3. Na terceira linha estamos a copiar o valor da variável  $n$  (que neste caso é 3) para a variável  $m$ ;
4. Por fim perguntamos o valor de  $m$  ao computador.

**Exercício 2.4.** Se num quinto passo atribuirmos o valor 7 a  $n$ , quanto vale agora  $m$ ?

### 2.2.2 Cadeias de caracteres

Mas nem todas as variáveis são números, podemos também definir cadeias de caracteres. As cadeias de caracteres são delimitadas por “” ou por ‘’, vejamos um exemplo:



```
File Edit View Terminal Tabs Help
[cfp@cfp4 ~]$ python
Python 2.5.2 (r252:60911, Sep 30 2008, 15:41:38)
[GCC 4.3.2 20080917 (Red Hat 4.3.2-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> a="As aves voam!"
>>> a
'As aves voam!'
>>> █
```

Infelizmente, nem todos os caracteres se podem usar em cadeias de caracteres, por vezes é preciso usar certos atalhos, por exemplo uma nova linha escreve-se “\n”, as aspas “\”...”

As cadeias de caracteres são estruturas rígidas, ou seja, não se podem alterar. Mas em contrapartida pode-se aceder a apenas parte da cadeia:

- `a[i]` acede ao i-ésimo carácter, por exemplo `a[4]` é igual a “v”. Note-se que a numeração dos caracteres começa pelo 0.
- `a[:j]` acede a todos os caracteres até à posição `j`, `j` não incluído, por exemplo `a[:8]` é igual “As aves”.
- `a[i:]` é igual ao anterior mas acedendo a todos os números após `i`, `i` incluído.
- `a[i:j]` acede aos caracteres entre as posições `i` e `j`, por exemplo `a[3:6]` é igual a “ave”.

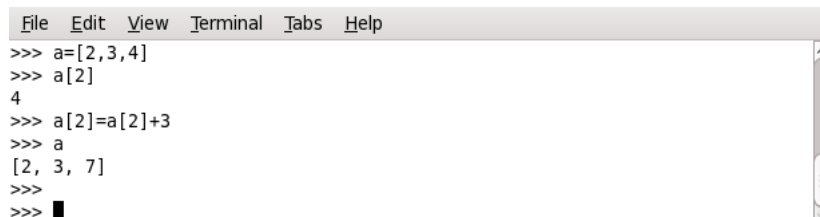
Pode-se também juntar várias cadeias, usando um sinal de +, por exemplo “Eu tenho uma”+“mesa” é igual a “Eu tenho uma mesa”. Ou multiplicar uma cadeia por um número natural: `2*“po”` é igual a “popo”.

**Exercício 2.5.** Crie uma variável com o valor “Das uvas se faz o vinho!”.

**Exercício 2.6.** Modifique a variável para “Das macas se faz a cidra!”. É possível pôr cedilhas e acentos nas palavras, mas nós vamos evitá-los.

### 2.2.3 Listas

Uma lista é um conjunto ordenado de objectos, estes podem ser números, cadeias de caracteres ou mesmo outras listas. Em Python uma lista é delimitada por parentesis rectos, por exemplo:

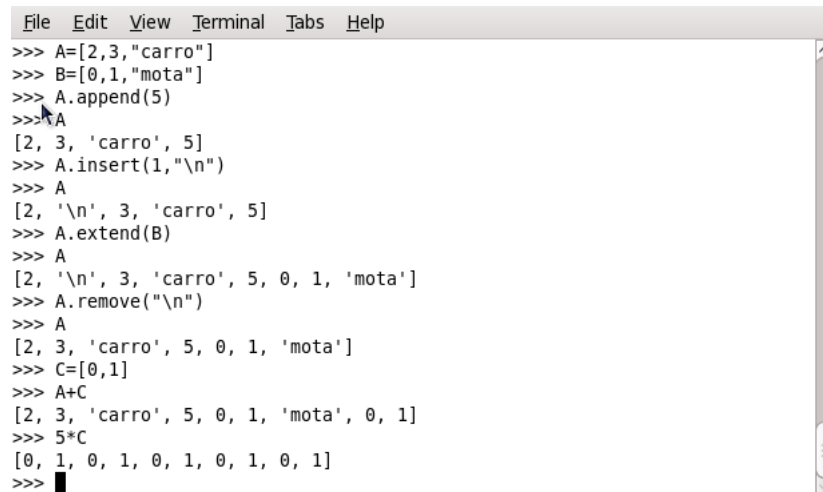


```
File Edit View Terminal Tabs Help
>>> a=[2,3,4]
>>> a[2]
4
>>> a[2]=a[2]+3
>>> a
[2, 3, 7]
>>> █
```

Como podem ver as listas são mais flexíveis que as cadeias. Vejamos algumas das operações que podemos realizar com elas:

- `A.append(X)` acrescenta o elemento `X` ao fim da lista `A`;
- `A.insert(I,X)` acrescenta o elemento `X` na posição `I` da lista `A`;
- `A.extend(B)` une a lista `A` com a lista `B`. Com a `B` após a `A`;
- `A.index(X)` devolve a posição da primeira aparição do elemento `X` na lista `A`;
- `A.remove(X)` apaga o primeiro elemento `X` da lista `A`;
- `len(A)` calcula o comprimento da lista `A`, ou seja, o número de elementos que a lista `A` contém;
- `A+B` devolve a união das listas `A` e `B`;
- `n*A` devolve a lista `A` repetida `n` vezes.

Vejamos um exemplo:



```

File Edit View Terminal Tabs Help
>>> A=[2,3,"carro"]
>>> B=[0,1,"mota"]
>>> A.append(5)
>>> A
[2, 3, 'carro', 5]
>>> A.insert(1,"\n")
>>> A
[2, '\n', 3, 'carro', 5]
>>> A.extend(B)
>>> A
[2, '\n', 3, 'carro', 5, 0, 1, 'mota']
>>> A.remove("\n")
>>> A
[2, 3, 'carro', 5, 0, 1, 'mota']
>>> C=[0,1]
>>> A+C
[2, 3, 'carro', 5, 0, 1, 'mota', 0, 1]
>>> 5*C
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
>>>

```

**Exercício 2.7.** Crie duas listas, digamos `A` e `B`, e calcule `A.append(B)`, o que é que acontece?

**Exercício 2.8.** Crie uma lista com todos os números inteiros ordenados entre 1 e 50. Dica experimente o que acontece se fizer `range(10)`.

## 2.3 Ficheiros Python

Imaginem que estamos a fazer um cálculo e ao final de 30 min reparamos que nos enganamos. Ora temos que repetir quase tudo. Mas existe uma solução que é escrever num ficheiro as ordens, a que nós chamamos normalmente de código, que nós queremos que o computador siga.

Por isso, começamos por abrir um editor de texto qualquer, e escrevemos:

---

```

#coding=utf8

frase="Este é o meu primeiro programa!" #Isto é um comentário
print frase

```

---



Gravamos o ficheiro, por exemplo com o nome `primeiro.py` (o `.py` serve para identificar os ficheiros de Python), e por fim corremos o Python com o ficheiro usando o comando `python primeiro.py`.

Vamos analisar o programa:

- `#coding=utf8` permite usar caracteres acentuados e o ç;
- `frase="Este é o meu primeiro programa."` cria uma cadeia de caracteres;
- `#` indica um comentário, o Python ignora a partir deste símbolo até à mudança de linha;
- `print frase` é o comando que escreve (ou imprime) o valor da variável `frase` na linha de comandos.

O código deve ser limpo, bem organizado e muito bem comentado. Senão, ninguém será capaz de perceber o que vocês estão a fazer nem mesmo vocês daqui a uma semana.

No caso especial do Python, o código tem que ser devidamente indentado (que consiste na adição de tabulações no início de cada linha), explicaremos este conceito à medida que precisarmos dele.

## 2.4 Funções

Imaginemos que queremos calcular o seno de 0.3, imprimir<sup>1</sup> uma frase ou colocar uma lista em ordem alfabética. Para efectuar estas tarefas vamos precisar de usar funções.

### 2.4.1 Print

De nada serve fazer um programa se ele não comunicar com o utilizador. A função `print` imprime aquilo que nós quisermos para a linha de comandos, como viram no exemplo anterior. Mas podemos fazer coisas mais complicadas:

---

```
#coding=utf8
n=5
print "Este é o meu primeiro cálculo:", "\n", n, "ao quadrado é", n**2
```

---

O resultado é simples de interpretar:

- Começamos por imprimir a frase: "Este é o meu primeiro cálculo:";
- Criamos uma nova linha com o `"\n"`;
- Imprimimos o valor de `n`, lembrem-se que `n` vale 5;
- Imprimimos "ao quadrado é";
- E por fim, o programa devolve o valor de  $n^2$ .

Outra forma de escrever este exemplo:

---

```
#coding=utf8
n=5
print "Este é o meu primeiro cálculo:\n %i ao quadrado é %i" % (n, n**2)
```

---

<sup>1</sup>Imprimir significa que o nosso programa nos devolve algo, por exemplo na linha de comandos, num ficheiro pdf ou em papel.

Notem que utilizamos o operador `%i` para introduzir uma variável inteira. Para variáveis reais usa-se `%f` e para cadeias usa-se `%s`.

**Exercício 2.9.** Crie um ficheiro que imprima a frase “Se  $n = 3$  e  $m = 2$ , então  $n \times m = 6$ ”, e onde seja fácil mudar os valores de  $n$  e  $m$ .

### 2.4.2 Input

Vamos ser audaciosos e construir um programa que peça valores ao utilizador:

---

```
#coding=utf8

n=input("Escreve um número:")
print "O número que escreveu foi",n
```

---

A função `input` imprime um texto e fica à espera que o utilizador devolva um número, uma lista, uma cadeia de caracteres... escritos como se fosse no ficheiro, ou seja, no caso duma cadeia de caracteres é preciso colocar as aspas.

Existe uma outra versão que é

---

```
#coding=utf8

s=raw_input("Como é que te chamas?")
print "O teu nome é",s
```

---

Esta função faz exactamente a mesma coisa que `input` mas grava tudo o que utilizador escrever em forma de cadeia de caracteres.<sup>2</sup>

**Exercício 2.10.** Escreve um programa que dê uma lista com todos os números inteiros entre dois valores definidos pelo utilizador.

### 2.4.3 Transformar cadeias de caracteres em números e vice-versa

Para o Python “45” e 45 são coisas diferentes, sendo, respectivamente, uma cadeia de caracteres e um número. Mas podemos convertê-los facilmente:

- A função `str()` transforma um número ou uma lista numa cadeia de caracteres;
- Enquanto que a função `eval()` faz a transformação inversa.

**Exercício 2.11.** Explique, passo por passo, o que acontece ao escrevermos:

---

```
#coding=utf8

a=56
b=str(a)
c=2*b
d=eval(c)
print 2*d
```

---



---

<sup>2</sup>Para inserir uma cadeia de caracteres não é preciso colocar aspas.

### 2.4.4 Definir funções

Por vezes queremos usar funções que ainda não estão definidas, nesse caso temos que ser nós a defini-las. Vejamos um exemplo duma definição:

---

```
#coding=utf8
def soma(x, y):
    u=x+y
    return 2*u

g=3
h=2
print soma(g, h)
```

---

Neste exemplo existem vários promenores importantes:

- Uma função precisa dum nome, por exemplo, “soma”, os nomes devem ser simples e explícitos;
- Para definir uma função usa-se “def função(variáveis):”;
- Pode haver quantas variáveis nós queiramos, de notar que os nomes que usamos aqui só servem para a definição;
- O Python é uma linguagem de indentação, ou seja, ele sabe que determinadas ordens pertencem à função porque estão indentadas;
- Dentro da função podemos fazer o que quisermos;
- “return” devolve-nos o resultado da função, note-se que não é obrigatório haver um resultado, a função pode imprimir algo na linha de comandos;
- A função acaba quando o texto deixar de ser indentado.

**Exercício 2.12.** Escreva um programa que calcule  $n^2$ .

**Exercício 2.13.** Escreva um programa que nos devolva  $n!$ . Use o facto de se  $n = 1$ ,  $n! = 1$ , escrito na linha: “If  $n==1$ : return 1”<sup>3</sup>

## 2.5 Controlo de fluxo

Às vezes queremos que uma certa ordem seja realizada apenas se uma certa condição é respeitada ou então queremos repetir uma acção umas tantas vezes. A isso chamamos de controlo de fluxo.

### 2.5.1 Condições

Em linguagem corrente uma condição exprime-se “Se chover eu não sairei de casa.”, ora em inglês “se” escreve-se “if”, vejamos um exemplo:

---

<sup>3</sup>Se  $n$  for igual a 1 devolve 1.

---

```
#coding=utf8
n=input("Escreva um número: ")
if n==0:
    print "O número é nem positivo nem negativo"
    print "Ou seja, é o zero."
elif n>0:
    print "O número é positivo"
else:
    print "O número é negativo"
```

---

Existe três comandos a ter em conta:

- “If” que significa “se”, neste caso “se  $n=0$ ” ele cumpre as ordens que aparecem em baixo indentadas;
- “elif” significa “senão se”, ou seja, se as condições que a precedem não forem respeitadas ela aplica mais uma condição;
- “else” significa “senão” e significa que se nenhuma das condições anteriores for respeitada ela cumpre as ordens que aparecem indentadas em baixo.

As condições podem ser escritas usando os símbolos apresentados na tabela:

símbolo	significado
==	igual
!=	diferente
>	maior
<	menor
>=	maior ou igual
<=	menor ou igual

Podemos ainda combinar várias condições usando “and”, “or” e “not”<sup>4</sup>, por exemplo:

---

```
#coding=utf8
n=input("Escreva um número: ")
m=input("E um segundo número: ")

if n>0 and m>0:
    print "São ambos positivos."

if (n>0 or m>0) and not (n>0 and m>0):
    print "Um e só um é positivo."
```

---

No primeiro exemplo temos que “ $n > 0$  e  $m > 0$ ”, enquanto que o segundo exemplo lê-se “um ou outro” e não “um e outro”.

**Exercício 2.14.** Crie um programa que verifique se certo número é par ou ímpar.

**Exercício 2.15.** Dado um determinado par de números queremos que um e apenas um seja positivo. Já damos um exemplo de como fazer isso, dê outros exemplos.

---

<sup>4</sup>Em português “e”, “ou” e “não”.

### 2.5.2 Repetições: para ... em ...

Temos uma operação e queremos-la repetir mil vezes. Ou escrevemos a ordem mil vezes, ou usamos um ciclo for (para em inglês):

---

```
#coding=utf8
n=input("Até que limite? ")

for i in range(n):
    print 2*i
```

---

Este programa imprime o dobro de todos os inteiros entre 0 e n-1. Vejamos passo por passo:

- “for i in range(n)”, significa que para todo o elemento, ao qual damos o nome de “i”, que está na lista “range(n)” efectuamos as ordens seguintes;
- As ordens aparecem nas linhas em baixo e aparecem indentadas, neste caso imprimimos o dobro de cada elemento.

Por vezes, queremos parar um ciclo “for” abruptamente, para isso usamos o comando “break”:

---

```
#coding=utf8
n=input("Até que número? ")

for i in range(n):
    print 2*i
    if i>=100:
        print "Atingiu o limite"
        break
```

---

Note que:

- As operações dentro do ciclo “if” estão indentadas em relação ao “if”;
- O “break” quebra o ciclo “for” imediatamente antes do “if”.

**Exercício 2.16.** Crie uma lista de naipes (paus, ouros, copas e espadas) e uma lista de valores de cartas (Ás, 2, 3, 4, 5, 6, 7, Valete, Dama e Rei) e imprima todas as cartas do baralho português.

**Exercício 2.17.** Verifique se determinado número é primo. Use a função “break”.

**Exercício 2.18.** Faça um programa que gere uma lista de números primos.

### 2.5.3 Repetições: enquanto ...

Podemos também repetir uma ordem enquanto (while em inglês) algo aconteça, por exemplo:

---

```
#coding=utf8
n=0
while n<=0:
    n=input("Escreva um número positivo: ")
print n
```

---

O ciclo vai ser repetido enquanto  $n$  for negativo.

Opçionalmente o ciclo pode ser acompanhado por um “else” que será realizado assim que a condição deixe de ser cumprida.

À imagem do ciclo “for” podemos quebrar um ciclo usando o comando “break”. E nesse caso o “else” não vai ser realizado.

**Exercício 2.19.** *Queremos calcular a raiz quadrada de  $n$ . Um método rudimentar é de pedir estimativas ao utilizador até que ele se aproxime o suficiente.*

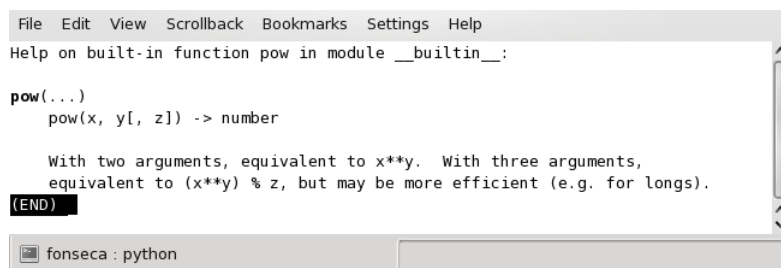
**Exercício 2.20.** *Um outro método é usando o algoritmo: seja  $r$  uma estimativa da raiz quadrada, sabemos que  $\frac{1}{2}(r + \frac{n}{r})$  vai estar mais próximo da raiz. Calcule a raiz quadrada com uma boa precisão.*

## Capítulo 3

# Python mais avançado

### 3.1 Ajuda

Mesmo depois de usar Python durante anos, não é suposto saber tudo sobre ele. Para isso existem manuais, a internet e a ajuda do Python. É sobre a última que nos debruçaremos a seguir. Num interpretador de Python escreva “help(pow)”:



```
File Edit View Scrollback Bookmarks Settings Help
Help on built-in function pow in module __builtin__:

pow(...)
    pow(x, y[, z]) -> number

    With two arguments, equivalent to x**y. With three arguments,
    equivalent to (x**y) % z, but may be more efficient (e.g. for longs).
(END)
```

fonseca : python

Como podem ver, o Python explica-nos como funciona a função “pow”. Ela pega em dois ou três argumentos e devolve-nos um número.

### 3.2 Bibliotecas

Experimente escrever “cos(2.1)” no interpretador. Em princípio o Python não reconhece a função. Mas o Python pode ser estendido graças a bibliotecas que contém a definição de algumas funções:

- A biblioteca padrão adiciona várias funcionalidades ditas padrão ao Python. Dentro desta biblioteca existem vários módulos:
  - O módulo math onde estão definidas várias funções como o seno, a exponencial, o logaritmo, a raiz quadrada entre outras (ver a ajuda do math) e também duas constantes o  $\pi$  e a constante de euler  $e$ . Veremos na secção 3.2.1 como utilizar este módulo. Só para números reais;
  - O módulo cmath permite funcionalidade parecidas com o módulo math mas definidas para números complexos;

- O módulo random permite-nos usar números pseudo-aleatórios;
- O módulo time permite várias operações com o tempo;
- numpy estende as potencialidades do Python, definindo vectores, matrizes multi-dimensionais bem como funções que operam nesses objectos. Existe ainda o scipy que o complementa;
- A biblioteca matplotlib que contém o módulo pylab que define várias rotinas para desenhar gráficos, ver secção 3.2.3;
- A biblioteca VPython ou Visual Python permite-nos criar, de forma simples, gráficos e animações a 3d interactivos.

Para aceder a um módulo usamos o comando:

---

```
#Primeiro importamos o módulo
import modulo
```

```
#Para usar uma função presente no módulo, escrevemos:
modulo.funcao(x)
```

---

ou então

---

```
#Importamos uma função presente no módulo
from modulo import funcao
```

```
#Para usar essa função presente no módulo, escrevemos:
funcao(x)
```

---

A primeira importa todo o módulo ao início enquanto que a segunda apenas importa uma função presente no módulo. A escolha de um dos métodos depende da nossa intenção.

### 3.2.1 O módulo math

Um dos módulos que nós mais vamos usar é o math, dele fazem parte as seguintes funções

Função	Comando
Coseno	cos()
Seno	sin()
Tangente	tan()
Arco-coseno	acos()
Arco-seno	asin()
Arco-tangente	atan()
Exponencial	exp()
Logaritmo	log()
Raiz Quadrada	sqrt()
Potência ( $x^y$ )	pow(x,y)
Valor absoluto	fabs()
Arredondamento por cima	ceil()
Arredondamento por baixo	floor()
Constante de Euler	e
$\pi$	pi



Duas notas. O seno, coseno e tangente usam radianos. No caso do logaritmo não neperiano usa-se  $\log(\text{base})$

Vamos ver alguns exemplos de utilização:

---

```
#coding=utf8
import math

print "O coseno de pi sobre 3 é:",math.cos(math.pi/3)
```

---

ou então

---

```
#coding=utf8
from math import pow, pi

print "A raiz cúbica de pi é:",pow(pi,1./3.)
```

---

**Exercício 3.1.** *Imprima uma tabela com o coseno entre 0 e 1.*

### 3.2.2 O módulo numpy

Vamos usar o módulo numpy para criar e manipular vectores e matrizes. Um pequeno exemplo:

---

```
#coding=utf8
import numpy

# Cria um vector tridimensional de valores reais
a=numpy.array([3],Float)

# Damos um valor às componentes do vector
a[0]=1
a[1]=3.2
a[2]=0.1

# Cria uma matriz 3x3 de valores inteiros inicializada a zero
M=numpy.zeros([3,3],Int)

# Altera o valor das entradas da matriz
M[0][2]=1
M[1][1]=-1
M[2][0]=1

# Imprime a matriz
print M
```

---

### 3.2.3 O módulo pylab

O nosso próximo passo vai ser aprender a fazer gráficos.

Vejamos um exemplo de utilização:

---

```
#coding=utf8
import pylab
```

```

#Criamos duas listas com os valores de x e de y
lista_X=[1,2,3,4,5,6,7,8,9,10]
lista_Y=[1,3,7,10,5,7,2,0,-2,0]

#Criamos um gráfico com as duas listas
pylab.plot(lista_X,lista_Y,"--g")

#Damos-lhe o título de "O meu primeiro grafico!"
pylab.title("O meu primeiro grafico!")

#Gravamos o gráfico no ficheiro grafico.png
pylab.savefig("grafico.png")

#Desenhamos o gráfico no ecrã
pylab.show()

```

---

O módulo pylab contém muitas funções já pré-definidas, dentro das quais encontramos:

- `plot(X,Y,opções)`, desenha um gráfico com as abcissas X, as ordenadas Y. Nas opções podemos definir o estilo, a cor, a espessura, o nome entre outras propriedades da linha (ver a ajuda de `pylab.plot`). De notar que ele desenha o gráfico na memória mas não o mostra nem grava nenhum ficheiro. Podemos desenhar várias linhas.;
- `show()`, comando que usaremos quase sempre no fim do programa. Ele mostra os gráficos que nós desenhamos;
- `savefig("figura")`, grava o ficheiro em formato png com o nome `figura.png`;
- `title("Titulo giro")`, dá ao gráfico o título: "Titulo giro";
- `xlabel("x_nome")`, dá um nome ao eixo das abcissas;
- `ylabel("y_nome")`, o equivalente para o eixo das ordenadas;
- `axis([x_min,x_max,y_min,y_max])`, define o tamanho dos eixos, permite outras opções como "auto" que permite ao Python de escolher os intervalos;
- `xlim()`, o mesmo que `axis()` mas só para o eixo de x;
- `ylim()`, o mesmo que `axis()` mas só para o eixo de y;
- `legend()`, cria uma legenda. Atenção que o nome que aparece é definido no `plot()` (ou no `subplot()`);
- `subplot()`, cria uma grelha de gráficos, útil para quando se quer desenhar vários gráficos lado a lado;
- `hist()`, faz um histograma;
- `figure()`, cria um outro gráfico. Útil quando se quer fazer vários gráficos, mas em janelas diferentes. Quando se faz `show()` as várias figuras vão aparecer. Pode ser escrito na forma `fig=figure()`, agora `fig` representa esta figura;
- `close()`, serve para recomençar um gráfico. Pode ser usado em conjugação com o `figure()`, por exemplo, depois de fazer 5 figuras (`fig1`, `fig2`, `fig3`, `fig4` e `fig5`) e de as gravar todas em ficheiros, queremos ver todas menos a 4<sup>a</sup>. Nesse caso fazemos `close(fig4)`, e todas ficam activas excepto a 4<sup>a</sup>.

**Exercício 3.2.** Desenhe o gráfico do seno de 0 a  $\pi$ . Faça com 5, 100 e 1000 pontos.

**Exercício 3.3.** Embeleze o gráfico com título, legenda, cores...

### 3.2.4 Visual Python

Vejamos agora um exemplo simples da utilização do Visual Python:

---

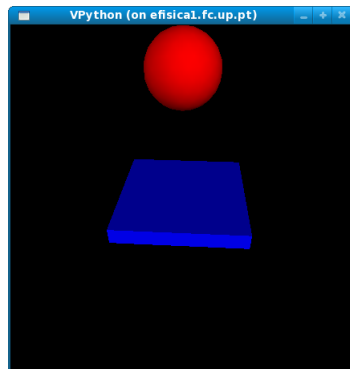
```
from visual import *

# Cria uma caixa na posição (0,0,0) de comprimento 4, altura 0.5 e espessura 4 de cor azul
caixa = box(pos=(0,0,0), length=4, height=0.5, width=4, color=color.blue)

# Cria uma esfera na posição (0,4,0) de raio 1 e de cor vermelha
bola = sphere(pos=(0,4,0), radius=1, color=color.red)
```

---

O resultado pode ser visto na seguinte figura:



É possível rodar a figura mantendo o botão direito do rato premido sobre a janela e movendo-o. Procedendo da mesma maneira com o botão do meio permite-nos ampliar ou reduzir.

Para além de caixas e bolas é possível desenhar outros objectos:

Objecto	Comando	Algumas opções
Caixa	box	pos=<p>,length=<x>,height=<x>,width=<x>
Cilindro	cylinder	pos=<p>,axis=<v>,radius=<x>
Esfera	sphere	pos=<p>,radius=<x>
Cone	cone	pos=<p>,axis=<v>,radius=<x>
Pirâmide	pyramid	pos=<p>,size=<v>
Seta	arrow	pos=<p>,axis=<v>
Elipsoide	ellipsoid	pos=<p>,length=<x>,height=<x>,width=<n>
Curva	curve	pos=[<p <sub>12</sub>
Anel	ring	pos=<p>,axis=<v>,radius=<x>,thickness=<x>
Hélice	helix	pos=<p>,axis=<v>,radius=<x>
Etiqueta	label	pos=<p>,text="Texto"

Em que <p> é um ponto, por exemplo (1, 1, 1), <v> é um vector, por exemplo (2, 0, -1), e <x> é um número real.

Com o Visual Python podemos animar facilmente objectos, basta para isso alterar as propriedades dos objectos num ciclo. Vejamos uma bola a orbitar em torno dum cilindro:

---

```
#coding=utf8
from visual import *
from math import *

N=1000    #Número de iterações
x=sin(0)  #Posição inicial
y=cos(0)  #Posição inicial

# Definimos um cilindro e uma esfera
eixo=cylinder(pos=(0,0,-1),axis=(0,0,2),radius=0.2)
bola=sphere(pos=(x,y,0),radius=0.3)

# Actualizamos a posição da esfera
for i in range(N):
    rate(100) #Limita o número de actualizações por segundo a 100

    x=sin(i*2*pi/N)
    y=cos(i*2*pi/N)

# Actualiza a posição da bola
bola.pos=(x,y,0)
```

---

Podem encontrar no site <http://vpython.org/contents/doc.html> informações mais detalhadas sobre o Visual Python. Um bom tutorial será [http://vpython.org/contents/docs/visual/VPython\\_Intro.pdf](http://vpython.org/contents/docs/visual/VPython_Intro.pdf).

**Exercício 3.4.** *Anime uma bola, fazendo-a aumentar e diminuir de volume sinusoidalmente.*

### 3.3 Trabalhar com ficheiros

Frequentemente, necessitamos de guardar os resultados dos nossos programas em ficheiros de texto. Ou então ir buscar dados a um ficheiro... Para abrir um ficheiro (quer ele exista ou não) usa-se o comando:

---

```
#coding=utf8
# Abrimos um ficheiro chamado dados.txt
fic=open("dados.txt","w")

# Escrevemos uma frase no ficheiro
fic.write("Galileu Galilei morreu no ano:\n")

# Para escrever um número temos primeiro que convertê-lo em cadeia de caracteres
s=str(1642)

# E finalmente escrevê-lo no ficheiro
fic.write(s)

# Fechamos o ficheiro
fic.close()
```

---

Agora sempre que quisermos chamar o ficheiro usamos “fic”. Devem ter reparado no “w” que vem do inglês write, isto quer dizer que abrimos um ficheiro para escrever nele (caso já

exista um ficheiro com esse nome o Python apaga todo o seu conteúdo). No lugar do “w”, poderíamos usar um “r” do inglês read e nesse caso só poderíamos ler o ficheiro. Ou “a” de append, logo qualquer coisa que escrevamos vai imediatamente para o fim do ficheiro. E “r+” que nos permite ler e escrever.

Para ler o ficheiro podemos usar:

---

```
#coding=utf8
# Abrimos o ficheiro dados.txt em modo de leitura
fic=open("dados.txt","r")

# Lemos a primeira e depois a segunda linha
print fic.readline()
print fic.readline()

# Fechamos o ficheiro
fic.close()
```

---

O comando `fic.readline()` lê uma linha e passa para a seguinte. O comando `fic.read()` lê o ficheiro completo e retorna-o numa cadeia de caracteres. O `fic.readlines()` lê também o ficheiro completo mas devolve uma lista cujos elementos são as linhas.

**Exercício 3.5.** *Crie uma tabela com os pontos do gráfico do seno entre 0 e  $\pi$  com mil pontos. Imprimi-la num ficheiro.*



# Capítulo 4

## Soluções

### 4.1 Soluções do capítulo 2

1. Basta escrever no interpretador  $4.5*3.14+3.4*2.1$  o que resulta em 27.194889045094623.
2. O primeiro vale 1 e o segundo vale 1.505. O que acontece é que no primeiro caso ele está a dividir dois números inteiros e então o resultado também vai ser um inteiro.
3. A resposta é  $7-2354785\%7$  que é igual a 1.
4. m não muda, logo obtém-se 3.
5. frase="Das uvas se faz vinho!".
6. frase=frase[:4]+"macas"+frase[8:16]+"cidra!".
7. Se criarmos a lista A=[1,2,3] e B=[4,5,6], A.append(B) vai-nos dar A=[1,2,3,4,5,6], ou seja, o quarto elemento da nova lista A vai ser [4,5,6].
8. Por exemplo X=range(1,51).
9. Código:

---

```
#coding=utf8
n=3
m=2

print "Se n=",n,"e m=",m," , então nxm=",n*m
```

---

10. Código:

---

```
#coding=utf8
# Começamos por pedir os dois valores ao utilizador
x_min=input("Escreve o primeiro número: ")
x_max=input("Escreve o último número: ")

# Imprimimos o resultado
print range(x_min,x_max+1)
```

---

11. Passo por passo:

- (a) Criamos uma variável numérica com valor 56;
- (b) Transformamo-la numa cadeia que vale "56";
- (c) Duplicamos a cadeia: "5656";
- (d) Transformamos a cadeia no número 5656;
- (e) Duplicamos o número para 11312 e imprimimos.

12. Código:

---

```
#coding=utf8
# Criamos a função quadrado
def quadrado(x)
    return x**2

# Pedimos um número ao utilizador
m=input("Escreva um número: ")

# Devolvemos o seu factorial
print m,"ao quadrado é igual a ",quadrado(m)
```

---

13. Código:

---

```
#coding=utf8
# Criamos a função factorial
def fac(n)
    if n==1:
        return 1
    return n*fac(n-1)

# Pedimos um número ao utilizador
m=input("Escreva um número: ")

# Devolvemos o seu factorial
print m,"factorial vale",fac(m)
```

---

14. Código:

---

```
#coding=utf8
# Pedimos um número
n=input("Escreva um número inteiro: ")

# Verificamos se ele é par ou impar
if n%2==0:
    print n,"é par."
else:
    print n,"é impar."
```

---

15. Código:



---

```
#coding=utf8
# Pedimos ambos os números
n=input("Escreva um número: ")
m=input("E um segundo número: ")

if (n>0 or m>0) and n*m<=0:
    print "Um e só um é positivo."
```

---

## 16. Código:

---

```
#coding=utf8
# Criamos os naipes e os valores das cartas:
naipe=["paus","ouros","copas","espadas"]
valor=["Ás","Duque","Terno","Quadra","Quina","Sena","Bisca","Valete","Dama","Rei"]

# Imprimimos as cartas todas
for r in valor:
    for s in naipe:
        print r,"de",s
```

---

## 17. Código:

---

```
#coding=utf8
# Pedimos um número
n=input("Escreva o número natural supostamente primo: ")

# p é uma variável auxiliar que valerá um se n não for primo.
p=0

# Verificamos se n é maior que 1 e inteiro
if n>1 and n%1==0:

    # Para todos os i entre 2 e n/2 testamos se é divisível
    for i in range(2,n/2+1):
        if n%i==0:
            print "O número",n,"é divisível por",i
            p=1
            break

    # Se p continuar nulo é porque n é primo
    if p==0:
        print "O número",n,"é primo."

    # Tratamos o caso da unidade à parte
    elif n==1:
        print "A unidade não é um número primo."

    # Só nos resta os número não inteiros positivos
    else:
        print "O número não é inteiro positivo."
```

---

Como podem verificar este método é lento para números muito grandes, pois ele tem que construir uma lista do tamanho de  $n$  mesmo que o número seja par. Um ciclo while (que aparece no capítulo a seguir) seria mais adequado:

---

```
#coding=utf8
# Pedimos um número
n=input("Escreva o número natural supostamente primo: ")

def primo(n):
    i=2
    while(i<=n/2):
        if n%i==0:
            print n,"é divisível por",i
            return 0
        i=i+1
    return 1

# Verificamos se n é maior que 1 e inteiro
if n>1 and n%1==0:
    if primo(n)==1:
        print n,"é primo."
    elif n==1:
        print "A unidade não é um número primo."
    else:
        print "O número não é inteiro positivo."
```

---

#### 18. Código:

---

```
#coding=utf8
# Vamos procurar primos até a N_max
N_max=2000

# Verifica se n é primo. Devolve 1 se n for primo e 0 senão
def primo(n):
    for i in range(2,n/2+1):
        if n%i==0:
            return 0
    return 1

# O principal do programa
for n in range(2,N_max+1):
    if primo(n)==1:
        print n
```

---

#### 19. Código:

---

```
#coding=utf8
# n é o número a calcular a raiz quadrada, e é uma espécie de precisão
n=5
e=0.0005
m=1
print "Queremos calcular a raiz quadrada de",n
```

---

```

# Enquanto a aproximação for má, ele continua
while m**2>e**2:
    p=input("Escreva um número: ")
    m=p**2-n
    print "A sua previsão falhou por:"m

# Imprime o resultado
print "A raiz quadrada é",p

```

---

## 20. Código:

```

#coding=utf8
# n é o número a calcular a raiz quadrada, e é uma espécie de precisão
n=5
e=0.000005
r=1
print "Queremos calcular a raiz quadrada de",n

# Enquanto a aproximação for má, ele continua
while (r**2-n)**2>e**2:
    r=0.5*(r+n/r)

# Imprime o resultado
print "A raiz quadrada é",r

```

---

## 4.2 Soluções do capítulo 3

## 1. Código:

```

#coding=utf8
# Vamos buscar o coseno e o pi
from math import cos, pi

# Definimos o número de pontos
p=20

# Imprimimos o resultado
for i in range(p):
    print cos(i*pi/p)

```

---

## 2. Código:

```

#coding=utf8
# Vamos buscar o coseno e o pi e carregamos o módulo pylab
from math import cos, pi
import pylab

# Definimos o número de pontos
p=5

# Criamos X e Y

```

```

Y=[]
X=[]
for i in range(p):
    Y.append(cos(i*pi/p))
    X.append(i*pi/p)

# Desenhamos o gráfico
pylab.plot(X,Y)
pylab.show()

```

---

3. Aquilo a que chamamos embelezar o gráfico é extremamente subjectivo. Aqui vão algumas dicas:
- 

```

#coding=utf8
from math import cos, pi
import pylab
p=1000
Y=[]
X=[]
for i in range(p):
    Y.append(cos(i*pi/p))
    X.append(i*pi/p)

# Cria uma linha de nome cos, com espessura 2, tracejada e vermelha
pylab.plot(X,Y,"--r",label="cos",linewidth=2)

# Dá um título
pylab.title("Gráfico do coseno de x")

# Nomeamos o eixo das abcissas de x
pylab.xlabel("x")

# Nomeamos o eixo das ordenadas de y e rodamos o nome
pylab.ylabel("y",rotation="horizontal")

# Criamos uma legenda
pylab.legend()
pylab.show()

```

---

4. Código:
- 

```

#coding=utf8
from visual import *
from math import *
import time

n=10          #Número de actualizações por segundo
t=200         #Tempo que dura a animação
r=1+sin(0)    #Raio inicial

# Definimos uma esfera
bola=sphere(pos=(0,0,0),radius=r,color=color.red)

```

```
# Fixamos o tamanho da janela
scene.range=(3,3,3)

# Actualizamos o raio da esfera
for i in range(n*t):
    time.sleep(1./n) #Faz uma pausa de 1/n segundos
    r=1+sin(i*0.2*pi/n)
    bola.radius=r
```

---

## 5. Código:

---

```
#coding=utf8
# Vamos buscar o coseno e o pi
from math import cos, pi

# Criamos um ficheiro
fic=open("dados.txt","w")

# Definimos o número de pontos
p=1000

# Imprimimos o resultado
for i in range(p):
    s=str(i*pi/p)+" "+str(cos(i*pi/p))+ "\n"
    fic.write(s)

# Fechamos o ficheiro
fic.close()
```

---



## Capítulo 5

# Passeio Aleatório

Neste capítulo vai ser apresentado um modelo estatístico adequado a explicar o movimento Browniano. Vamos começar por estudar o problema do passeio aleatório, o qual é a base do nosso modelo, e depois aplicá-lo ao estudo do movimento Browniano.

### 5.1 Experiência

Começemos por uma experiência conceptual. Imaginem alguém que ingeriu uma grande quantidade de bebidas alcoólicas, ao ponto de perder completamente o seu sentido de orientação. Imaginemos que este, devido à sua falta de equilíbrio, em cada instante de tempo dá um passo numa direcção aleatória. A nossa pergunta é: ao fim de um dado tempo, e a que distância estará este da sua posição inicial e qual a sua posição média? (neste caso vai-nos interessar o quadrado desta distância, também chamado de desvio quadrático. Vai ser claro porquê mais à frente.)

Para responder a esta pergunta, podemos utilizar os nossos conhecimentos de python e fazer um programa que simule esta situação improvável de observar na vida real.

Começemos por definir o nosso problema de um modo mais formal: queremos calcular uma trajectória com  $N$  passos, onde em cada um deles se dá um passo numa direcção aleatória. Queremos também calcular o desvio quadrático e a sua posição média ao longo dos  $N$  passos. Para simplificar vamos assumir que só existem 4 direcções possíveis: cima, baixo, esquerda e direita. Como implementar isto?

- Qual a maneira mais indicada para guardar os dados da trajectória e desvio quadrático? Necessitamos de guardar  $N$  valores de  $x$ ,  $y$ ,  $x$  médio,  $y$  médio e  $r^2$ . A maneira mais imediata de fazer isto é usar uma lista.
- O facto de haverem  $N$  passos implica que temos de fazer um ciclo, que corre  $N$  vezes, adicionando um ponto à trajectória de cada vez.
- É necessário escolher direcções aleatórias - logo será necessário gerar números aleatórios, e fazer uma decisão consoante o resultado.

Uma implementação possível:

---

```
#coding=utf8

import pylab
import random
```

```

# Cria as listas que representam a nossa trajectória x,y
# inicialmente estamos no ponto (0,0)
x = 0
y = 0
trajectoria_x = [0]
trajectoria_y = [0]

#Cria a lista do R^2, x médio e y médio
R2      = [0]
soma_x  = 0
x_medio = [0]
soma_y  = 0
y_medio = [0]

N = 1000
# Adiciona um ponto novo à trajectória N vezes
for i in xrange(N):

    # Adiciona valores as listas com os dados de x e y da nossa trajectória
    # segundo um passeio aleatorio
    p = random.random()*4
    if(p < 1):
        x = x + 1
    elif(p < 2):
        x = x - 1
    elif(p < 3):
        y = y + 1
    else:
        y = y - 1

    trajectoria_x.append(x)
    trajectoria_y.append(y)

    R2.append(x**2 + y**2)

    soma_x += x
    x_medio.append(soma_x/(i+1))
    soma_y += y
    y_medio.append(soma_y/(i+1))

#força o gráfico a ter um aspecto rectangular 14x4
pylab.figure(0,figsize=(14,4))

# Desenha um linha com a trajectória
pylab.subplot(131)
pylab.title('Trajectoria')
pylab.plot(trajectoria_x,trajectoria_y)

#Desenha o gráfico do x médio e y médio
pylab.subplot(132)
pylab.title('X e Y medios em funcao do tempo')
pylab.plot(x_medio)

```



```

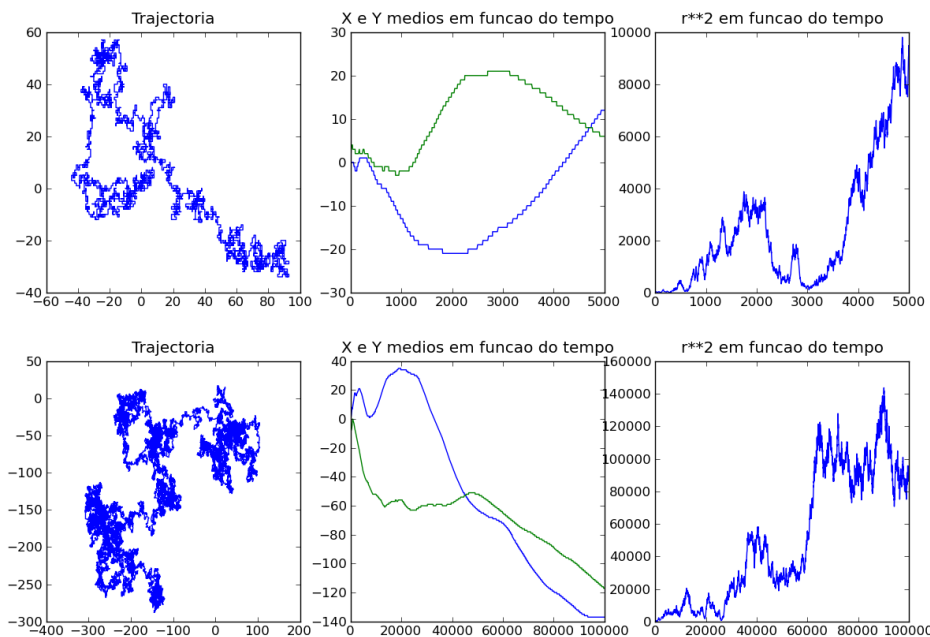
pylab.plot(y_medio)

#Desenha o gráfico do R^2
pylab.subplot(133)
pylab.title('r**2 em funcao do tempo')
pylab.plot(R2)

pylab.show()

```

Usando este programa, que resultados obtemos? Dois resultados possíveis, com  $N=5000$  e  $N=100000$ :



A trajetória parece caótica, sem nenhum padrão óbvio. De facto, executando o programa várias vezes e observando várias trajetórias diferentes não há muitas conclusões que se possam tirar. Mesmo os gráficos do  $x$  e  $y$  médios são *geralmente* diferentes do que esperaríamos intuitivamente: se há a mesma probabilidade de andar para um lado que para o outro, seria de esperar que em média o deslocamento fosse 0.

Vemos assim que não é muito vantajoso analisarmos o problema em termos de uma só partícula. Na verdade é apenas quando se faz uma estatística sob a trajetória de várias partículas que o problema fica interessante.

## 5.2 Vários Caminhantes

Imaginemos o mesmo problema, mas em vez de termos apenas um bêbado temos uma comitiva de  $M$  bêbados. Queremos estudar os seus deslocamentos médios em  $x$  e  $y$  (num dado instante de tempo, não ao longo do tempo como no exemplo anterior) e a média dos seus desvios quadráticos. Usemos outra vez o python para resolver o problema. Agora em vez de haver apenas um caminhante, existem  $M$  caminhantes, logo  $M$  posições  $x, y$ . Como não vamos fazer os gráficos das trajetórias, apenas necessitamos de guardar a posição corrente dos caminhantes, ao

contrário do exemplo anterior que guardávamos a trajectória ao longo do tempo. Assim, podemos utilizar uma lista para guardar todos os valores de  $x$  num dado instante de tempo (e outra para  $y$ ).

Uma implementação possível:

---

```
#coding=utf8

import pylab
import random

# Número de Caminhantes
M = 50

# Cria as listas que vão conter os pontos x_i,y_i no último instante de tempo
trajectorias_x = []
trajectorias_y = []
# Adiciona M pontos iniciais às trajectórias
for j in range(M):
    trajectorias_x.append(0)
    trajectorias_y.append(0)

#Cria a lista do R^2, x médio e y médio
R2 = [0]
x_medio = [0]
y_medio = [0]

N = 1000
# Processa pontos novos N vezes
for i in xrange(N):
    # Adiciona valores as listas com os dados de x e y das nossas trajectórias
    # segundo um passeio aleatorio
    for j in range(M):
        p = random.random()*4
        if(p < 1):
            trajectorias_x[j] = trajectorias_x[j] + 1
        elif(p < 2):
            trajectorias_x[j] = trajectorias_x[j] - 1
        elif(p < 3):
            trajectorias_y[j] = trajectorias_y[j] + 1
        else:
            trajectorias_y[j] = trajectorias_y[j] - 1

    # Calcula a média dos desvios quadráticos
    soma_R2 = 0.
    for k in range(M):
        soma_R2 = soma_R2 + trajectorias_x[k]**2+trajectorias_y[k]**2

    R2.append(soma_R2/M)

    # Calcula a posição média entre as partículas
    soma_x = 0.
    soma_y = 0.
    for l in range(M):
```

```

    soma_x = soma_x + trajetorias_x[l]
    soma_y = soma_y + trajetorias_y[l]
    x_medio.append(soma_x/M)
    y_medio.append(soma_y/M)

#força o gráfico a ter um aspecto rectangular 10x4
pylab.figure(0,figsize=(10,4))

#Desenha o gráfico do x médio e y médio
pylab.subplot(121)
pylab.plot(x_medio)
pylab.plot(y_medio)
pylab.title('X e Y medios em funcao do tempo')
#força a escala em y de -10 a 10
pylab.ylim((-10,10))

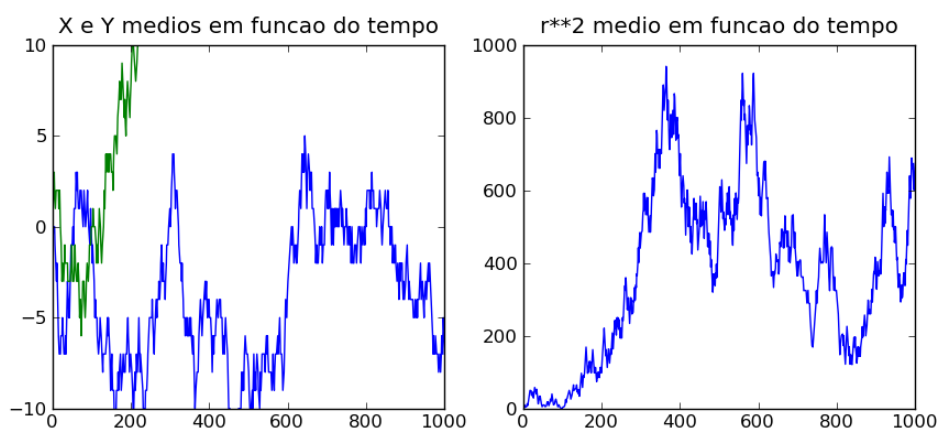
#Desenha o gráfico do R^2
pylab.subplot(122)
pylab.plot(R2)
pylab.title('r**2 medio em funcao do tempo')

pylab.show()

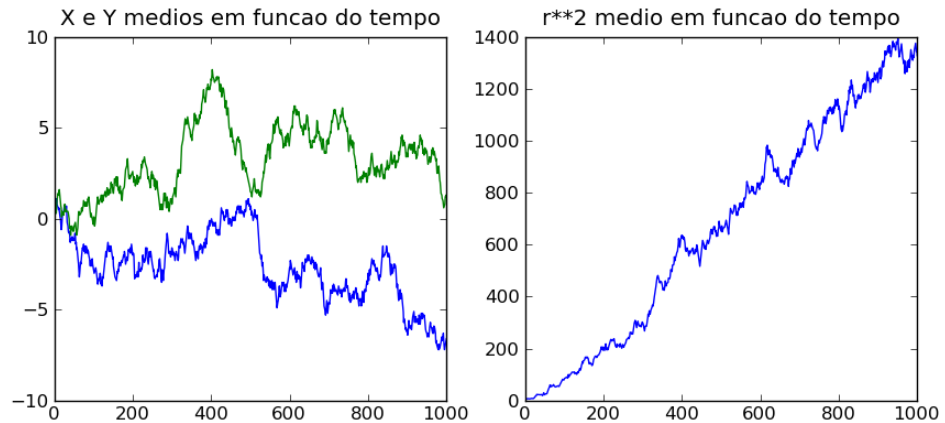
```

---

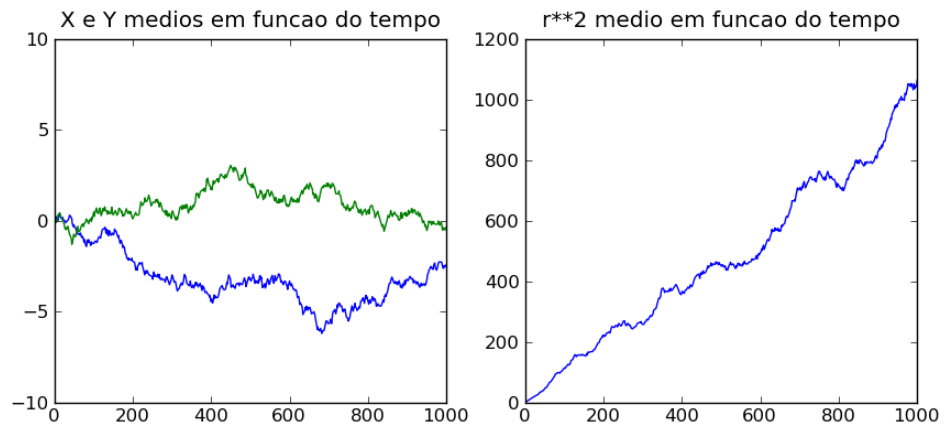
Vejamos resultados possíveis. Para  $N=1000$  (pontos da trajectória),  $M=1$  (número de caminhantes):



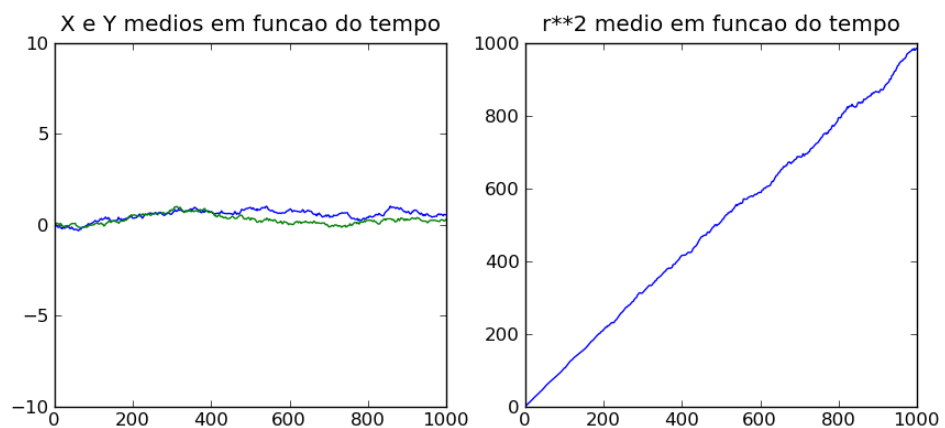
Apenas um caminhante. O resultado é semelhante ao do exemplo anterior. Para  $N = 1000$ ,  $M=10$ :



Para  $N = 1000$ ,  $M = 50$ :



Para  $N = 1000$ ,  $M = 500$ :



Ao aumentarmos o número de caminhantes, vemos duas tendências claras: a média dos deslocamentos das partículas mantém-se cada vez mais próxima de zero durante as trajetórias e o desvio quadrático médio torna-se cada vez mais próximo de uma recta. Estes resultados implicam que, em média, as partículas se afastam da posição inicial, de maneira que a média

quadrado da distância a esta posição inicial cresça linearmente com o tempo. Este resultado é concordante com o que Einstein derivou em 1905 na sua publicação: 'Investigações na teoria do movimento browniano', a qual explicou o movimento errático de partículas de polen suspensas em água com argumentos baseados em moléculas e átomos, conceitos bastante disputados na altura.

Assim, pode-se dizer que este modelo simples do passeio aleatório e a teoria desenvolvida por Einstein dão uma explicação para o comportamento dos grãos de polen de brown, de maneira a reforçar a credibilidade da ideia que a matéria é composta por átomos. Este foi um passo importante para a ciência no início do século XX.

## 5.3 Previsão

Não fiquemos por aqui. Usemos o nosso modelo teórico para prever o que acontece noutras situações. Por exemplo, imaginem grãos de polen suspensos em água. Estes estão sujeitos a uma grande quantidade de choques por parte das moléculas de água, logo idealmente vão descrever movimento browniano. Mas o que acontece quando se encontram numa superfície inclinada, ou quando existe uma pequena corrente na água para um dos lados? Naturalmente as suas trajectórias serão mais direccionadas para uma das direcções.

Como podemos inserir isto no nosso modelo? Uma maneira será atribuir uma maior probabilidade de o caminhante se deslocar para um dos lados do que para os outros. Isto pode ser feito substituindo, no exemplo anterior, a linha:

---

```
p = random.random()*4
```

---

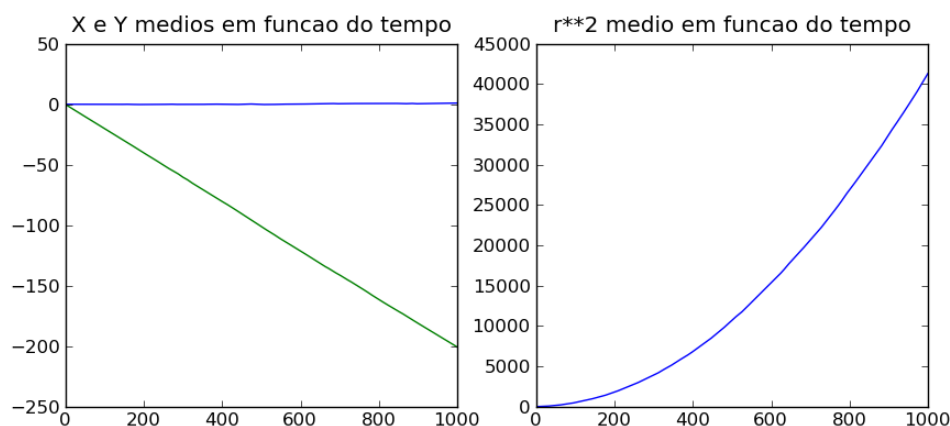
Por:

---

```
p = random.random()*5 #ou um número maior
```

---

Assim, a probabilidade do caminhante se deslocar para baixo será o dobro das outras direcções. Porque? Vejamos os resultados possíveis: se sair um número entre 0 e 1, vai para a direita, entre 1 e 2, vai para a esquerda, 2 e 3, cima, 3 e 5, baixo. Como é duas vezes mais provável ir para baixo do que ir para uma das outras direcções, o efeito é equivalente à presença de inclinação ou fluxo de água. Um resultado possível, com  $N=1000$  e  $M=500$  será:



Vemos que o  $y$  médio decresceu quase linearmente ao longo do tempo, tal como num movimento rectilíneo. Não surpreendentemente, a média de  $r^2$  aumenta quadraticamente, tal como

no movimento rectilíneo (se a posição é  $x = 0, y = mt$ , o desvio quadrático será  $r^2 = x^2 + y^2 = m^2 t^2$ , quadrático no tempo). Assim, se fizermos uma experiência seguindo as trajectórias de partículas em água, e calculando o desvio quadrático médio, esperamos um gráfico com uma recta caso a amostra esteja nivelada e sem correntes ou uma parábola caso contrário.

## Capítulo 6

# Da ordem ao caos

Segundo a física clássica o nosso mundo é determinístico. Ou seja, se nós repetirmos a mesma experiência nas mesmas condições vamos obter exactamente os mesmos resultados. Porém não somos capazes de prever tudo. Porquê? Esta dificuldade em prever certos fenómenos surge, de entre outras causas, do caos.

Para perceber donde surge o caos, propomos aqui um exemplo simples.

### 6.1 O problema da pirâmide

Imaginemos um pequeno jogo.

Temos uma pirâmide de tubos, tal como mostra a figura 6.1, abertos em baixo e em cima. Deixamos cair uma bola por entre os tubos e deixamo-la cair por entre os tubos até que por fim ela “se decida” a parar numa certa caixa. Agora queremos adivinhar que caixa é essa.

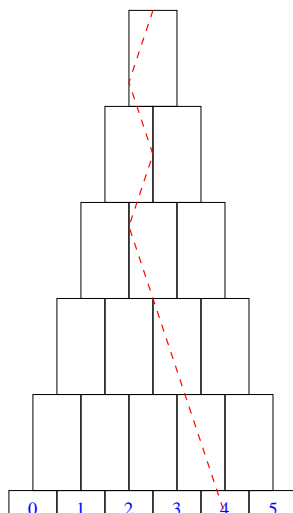


Figura 6.1: Neste exemplo, a bola cai no primeiro tubo e vai batendo nas várias paredes até chegar à caixa 4.

Infelizmente o problema é ainda muito complicado para ser tratado com todo o pormenor

em apenas uma semana. Por isso, em 2008, decidimos simplificar o modelo desprezando a gravidade.

### 6.1.1 Caixa final

Para calcular a que caixa a bola chega utilizamos, em 2008, o seguinte programa:

---

```
#coding=utf8

x=0.59864 #posição de entrada
b=5.12653 #comprimento do tubo
w=.301 #tangente do ângulo inicial
n=100 #nº de níveis
#a espessura dos tubos é 1

num=0 #indicador do número da caixa

# saida(x,w) é uma função auxiliar que
# calcula a posição e o ângulo de saída para cada tubo
# sabendo a posição e o ângulo de entrada neste

def saida(x,w):
    y=w*b+x #posição final se o tubo não tivesse paredes
    n=int(y) #devolve a parte inteira de y
    d=y-n #
    if y>=0:
        if n%2==0:
            wf=w
            xf=d
        else:
            wf=-w
            xf=1-d
    else:
        if n%2==0:
            wf=-w
            xf=-d
        else:
            wf=w
            xf=1+d
    return [xf,wf]

# Calcula fila por fila onde está a bola
for j in range(n):
    r= saida(x,w)
    if r[0]>0.5:
        x=r[0]-0.5
        num=num+1
    else:
        x=0.5+r[0]
        w=r[1]

#resultado final
print num
```

---



### 6.1.2 Trajectória da bola

Para desenhar a trajectória bastou-nos registar a posição a cada nível e no fim fazer um gráfico:

---

```
#coding=utf8
import pylab
import math

x=0.59864 #posição de entrada
b=5.12653 #comprimento
w=.301 #tangente do ângulo inicial
n=100 #nº de níveis

#a espessura dos tubos é 1

titulo="Tangente do angulo =" +str(w) #titulo do gráfico
nome="traj_"+str(int(1000*w)) #nome do ficheiro

num=0 #indicador do número da caixa

Pos_x=[] #abscissa = número da caixa
Pos_y=[] #ordenada = nível

#e=0.01

#calcula a posição e o ângulo de saída para cada tubo
#sabendo a posição e o ângulo de entrada
def saida(x,w):
    y=w*b+x
    #int() devolve a parte inteira de y
    n=int(y)
    d=y-n
    if y>=0:
        if n%2==0:
            wf=w
            xf=d
        else:
            wf=-w
            xf=1-d
    else:
        if n%2==0:
            wf=-w
            xf=-d
        else:
            wf=w
            xf=1+d
    return [xf,wf]

#calcula fila por fila onde está a bola
for j in range(n):
    r= saida(x,w)
    #b=b+e #varia o tamanho do cilindro, aumenta a imprevisibilidade
    if r[0]>0.5:
        x=r[0]-0.5
```

```

    num=num+1
    else:
        x=0.5+r[0]
        w=r[1]
        Pos_y.append(n-j) #ordenadas da trajetória
        Pos_x.append(num+(n-j)/2.) #abscissas da trajetória

#desenha a trajetória
pylab.plot(Pos_x,Pos_y)
pylab.axis([0,n,0,n])
pylab.title(titulo)
pylab.savefig(nome)
#pylab.show()

```

Podemos ver na figura 6.2 dois exemplos de trajetórias.

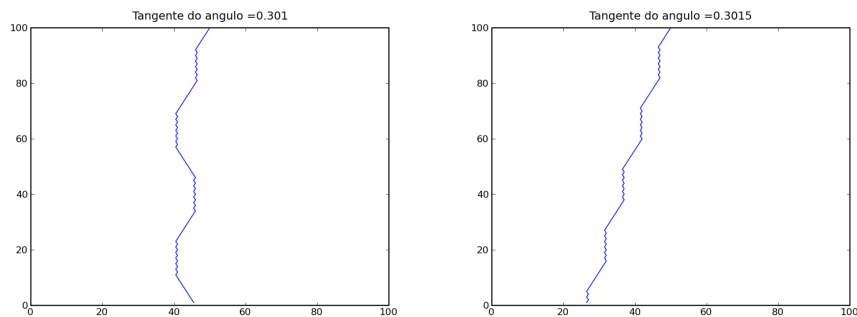


Figura 6.2: Trajetórias para duas configurações quase idênticas.

### 6.1.3 Variando o ângulo

Para melhor testar a dependência do modelo nas condições iniciais, corremos o programa variando o ângulo inicial. Nesse intuito alteramos o nosso programa:

```

#coding=utf8
import pylab
import math

x=0.59864 #posição de entrada
B=5.12653 #comprimento
w_min=-1.343 #tangente do ângulo inicial mínimo
w_max=1.124 #tangente do ângulo inicial máximo
n=100 #nº de níveis
P=3000 #número de lançamentos

#a espessura dos tubos é 1

Pos_x=[] #abscissa = tangente inicial
Pos_y=[] #ordenada = número da caixa

```

```

#e=0.01

#calculo a posição e o ângulo de saída para cada tubo
#sabendo a posição e o ângulo de entrada
def saida(x,w):
    y=w*b+x
    #int() devolve a parte inteira de y
    n=int(y)
    d=y-n
    if y>=0:
        if n%2==0:
            wf=w
            xf=d
        else:
            wf=-w
            xf=1-d
    else:
        if n%2==0:
            wf=-w
            xf=-d
        else:
            wf=w
            xf=1+d
    return [xf,wf]

#percorre os vários ângulos
for k in range(P):
    w=w_min+(w_max-w_min)/P*k
    Pos_x.append(w) #tangente do ângulo de partida
    num=0
    b=B
#calcula fila por fila onde está a bola
    for j in range(n):
        r= saida(x,w)
        #b=b+e #varia o tamanho do cilindro, aumenta a impresibilidade
        if r[0]>0.5:
            x=r[0]-0.5
            num=num+1
        else:
            x=0.5+r[0]
            w=r[1]

    Pos_y.append(num) #número da caixa

#desenha o gráfico
pylab.plot(Pos_x,Pos_y,".")
pylab.xlabel("tangente do angulo inicial")
pylab.ylabel("numero da caixa final")
pylab.title("O numero da caixa final em funcao da tangente\ncom a altura da caixa dependente do n")
pylab.axis([-1.4,1.2,0,100])
#pylab.savefig("caos")
pylab.show()

```

---

Isto resulta no gráfico .



Figura 6.3: Gráfico da variação do número da caixa final em função da tangente do ângulo inicial.

### 6.1.4 Perguntas em aberto

1. O que acontece se em vez de variarmos o ângulo inicial variarmos a posição inicial?
2. Será que podemos considerar a gravidade? Vai aumentar ou diminuir a imprevisibilidade?
3. Existe alguma forma de calcular a imprevisibilidade? Calcula duas trajetórias começando em condições semelhantes (mas não exactamente iguais) e veja como divergem. Repete a experiência várias vezes.
4. E se o tamanho da caixa fosse aleatório, mas pré-definido?
5. Qual é a probabilidade de a bola cair numa determinada posição?
6. Imaginemos a mesma experiência, mas na qual a bola “escolhe” ir para o tubo à esquerda ou à direita aleatoriamente. Como se comparam os resultados de ambas as experiências?

## 6.2 Outros modelos

Na preparação deste projecto foram investigados vários modelos com características semelhantes. O modelo da pirâmide foi escolhido por ser o mais adequado às pretensões da Escola de Verão, mas existem outros igualmente interessantes. Nesta secção fazemos uma pequena digressão sobre esses modelos.

### 6.2.1 Sistemas dinâmicos e coelhos

Numa certa ilha deserta existe uma população de coelhos, a densidade de coelhos no momento  $n$  é dada por  $u_n$ . É normal considerar que a taxa de natalidade dos coelhos seja proporcional à quantidade de coelhos existentes ( $\sim u_n$ ). Por outro lado, esperamos que a quantidade de coelhos dependa da quantidade de comida disponível. Por sua vez a quantidade de comida

depende da concentração de coelhos ( $\sim (1 - u_n)$ ). Podemos então dizer que a concentração na segunda geração de coelhos é descrita pela equação:

$$u_{n+1} = A u_n (1 - u_n)$$

Apesar de parecer tão simples, esta equação esconde propriedades bastante interessantes. Para começar tem dois pontos fixos:  $u = 0$ , se não há coelhos no início não há procriação;  $u = 1 - A^{-1}$ . Para  $A < 3$  (valor aproximado) o sistema converge sempre, depois tem um intervalo em que o sistema alterna entre dois pontos, três pontos... assim por diante, até que a partir de 3.6 ele começa a visitar todos os pontos entre 0 e 1. Como se pode ver no gráfico 6.4.

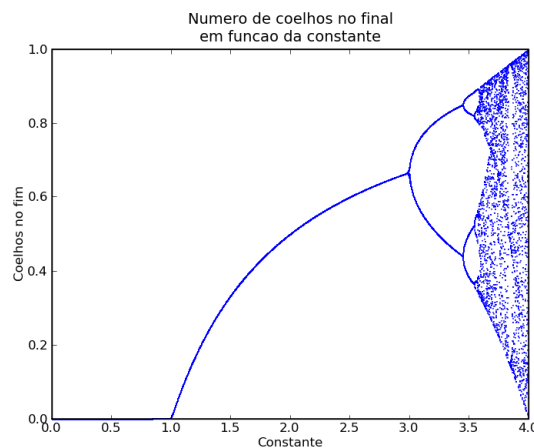


Figura 6.4: Quando se varia a constante  $A$ , obtêm-se resultados completamente diferentes, desde regiões convergentes até outras que são verdadeiramente caóticas.

O código usado foi:

```
#coding=utf8

#           Este programa simula uma população de coelhos
#
# A taxa de natalidade dos coelhos numa sociedade é proporcional
# ao número de coelhos e à quantidade de comida, esta é
# proporcional também ao número de coelhos, ficamos assim com uma
# uma equação do tipo  $u \rightarrow A u (1-u)$ 

import pylab

num=200 #quantidade de interações
Q=4 #1-Experiência Simples; 2-Variar número de coelhos;
    #3-Variar constante; 4-Varia ambos

N_A=2000 #número de passos (variando a constante)
N_x=20 #número de passos (variando num de coelhos)

A_min=0. #constante mínima para 3
A_max=4. #constante máxima para 3
```

```

A=[1.0,2.0,3.0] #constante para 1 e 2

n_min=0.1 #número de coelhos mínimo para 2
n_max=0.9 #número de coelhos máximo para 2
n=[0.7] #número de coelhos para 1 e 3

#cálculo da interação seguinte
def inter(a,u):
    return a*u*(1-u)

# Calcula o número de coelhos
# Desenha o gráfico da variação desse número em função do tempo

def Simples():
    X=range(num)
    Y_num=[]
    g=0
    for a in A:
        for x in n:
            Y_num.append([])
            s="numero="+str(x)+" e A="+str(a)
            for j in range(num):
                x=inter(a,x)
                Y_num[g].append(x)
            pylab.plot(X,Y_num[g],"",label=s)
            g+=1
    pylab.title("Numero de coelhos por interacao.")
    pylab.xlabel("Interacao")
    pylab.ylabel("Numero de coelhos")
    pylab.legend()

# Calcula o número de coelhos depois de num interações
# Para diferentes quantidades iniciais de coelhos

def Quantidade():
    for a in A:
        X_in=[]
        Y_fin=[]
        for i in range(N_x):
            x=n_min+i*1.*(n_max-n_min)/N_x
            X_in.append(x)
            for j in range(num):
                x=inter(a,x)
                Y_fin.append(x)
            s="A="+str(a)
            pylab.plot(X_in,Y_fin,"",label=s)
    pylab.title("Numero de coelhos no final\ndependendo do numero inicial")
    pylab.xlabel("Coelhos no inicio")
    pylab.ylabel("Coelhos no fim")
    pylab.legend()

# Calcula o número de coelhos depois de num interações
# Para diferentes constantes

```

```

def Constante():
    for x in n:
        X_a=[]
        Y_fin=[]
        s="numero inicial = "+str(x)
        for i in range(N_A):
            a=A_min+i*1.*(A_max-A_min)/N_A
            X_a.append(a)
            u=x
            for j in range(num):
                u=inter(a,u)
            Y_fin.append(u)
        pylab.plot(X_a,Y_fin,"",label=s)
    pylab.title("Numero de coelhos no final\nem funcao da constante")
    pylab.xlabel("Constante")
    pylab.ylabel("Coelhos no fim")
    pylab.legend()

#      Calcula o número de coelhos depois de uma interacções
#      Para diferentes constantes
#      É repetida para diferentes quantidades iniciais

def Ambos():
    X_a=[]
    Y_fin=[]
    for i in range(N_A):
        a=A_min+(A_max-A_min)*i*1./N_A
        for k in range(N_x):
            x=n_min+k*1.*(n_max-n_min)/N_x
            X_a.append(a)
            for j in range(num):
                x=inter(a,x)
            Y_fin.append(x)
    pylab.plot(X_a,Y_fin,"")
    pylab.title("Numero de coelhos no final\nem funcao da constante")
    pylab.xlabel("Constante")
    pylab.ylabel("Coelhos no fim")

#escolhe que rotina seguir
if Q==1:
    Simples()
elif Q==2:
    Quantidade()
elif Q==3:
    Constante()
else:
    Ambos()

#desenha o gráfico
pylab.savefig("coelhos")
pylab.show()

```

---

### 6.2.2 Mapa a 2 dimensões

Um mapa a 2 dimensões consiste num conjunto de duas variáveis. Em cada passo as duas variáveis são actualizadas em simultâneo, vejamos alguns exemplos que apresentam caos:

1. Mapa de Duffing:

$$\begin{cases} x_{n+1} &= y_n \\ y_{n+1} &= -b x_n + a y_n - y_n^3 \end{cases}$$

2. Mapa de Hénon:

$$\begin{cases} x_{n+1} &= y_n + 1 - a x_n^2 \\ y_{n+1} &= b x_n \end{cases}$$

3. Mapa de Tinkerbell:

$$\begin{cases} x_{n+1} &= x_n^2 - y_n^2 + a x_n + b y_n \\ y_{n+1} &= 2 x_n y_n + c x_n + d y_n \end{cases}$$

Fica a cargo dos alunos mais interessados em programar estes sistemas e escolher as informações relevantes a retirar daqui.

### 6.2.3 Funções por ramos

Uma função por ramos é uma função que tem dois ou mais comportamentos distintos. Quando iteradas várias vezes, algumas delas ganham um carácter imprevisível.

Vejamos alguns exemplos:

1. Mapa tenda:

$$x_{n+1} = \begin{cases} \mu x_n & \text{se } x_n < \frac{1}{2} \\ \mu(1 - x_n) & \text{se } x_n > \frac{1}{2} \end{cases}$$

2. Mapa diático:

$$x_{n+1} = \begin{cases} 2 x_n & \text{se } x_n < \frac{1}{2} \\ 2 x_n - 1 & \text{se } x_n > \frac{1}{2} \end{cases}$$

A programação destes modelos é idêntica às anteriores. Sendo assim, fica como exercício aos eventuais leitores que chegaram até aqui.

### 6.2.4 Geradores de números aleatórios

Os computadores actuais são incapazes de produzir números aleatórios, uma vez que um computador só utiliza fenómenos determinísticos. No entanto, é possível produzir números aparentemente aleatórios, que são chamados números pseudo-aleatórios. Não é de estranhar que os algoritmos usados sejam caóticos. Vejamos alguns exemplos:

1. Método de Van Neumann:

- (a) Seja  $v_n$  um número de  $n$  dígitos;
- (b) Calculamos  $m^2$ ;
- (c) E isolamos os  $n$  dígitos centrais, temos assim o novo número  $v_{n+1}$ .



2. Método de Fibonacci, seja  $m$  um primo grande:

$$x_{n+1} = (x_n + x_{n-k}) \% m$$

em que  $k$  é um número natural.

3. Método de Lehmnner, sejam  $a$ ,  $b$  e  $c$  números grandes e primos entre si:

$$u_{n+1} = (a u_n + b) \% c$$

O aluno interessado não terá dificuldade em programar estes geradores.



## Capítulo 7

# Cálculo de Órbitas

Neste trabalho calculamos a órbita de uma sonda em torno de Júpiter. Para isso, simulamos no computador o movimento da sonda e de Júpiter. Obviamente que o resultado pode ser facilmente adaptado a outros problemas tal como a órbita da lua.

### 7.1 Forças

Existem na natureza quatro forças fundamentais: a forte, a fraca, a electromagnética e a gravítica. Mas no espaço, apenas estamos sujeitos à força gravítica. Vejamos a forma da força gravítica entre dois corpos de massa  $m_1$  e  $m_2$ , e posição  $\vec{r}_1$  e  $\vec{r}_2$  respectivamente:

$$\vec{F}_{21} = -G \frac{m_1 m_2}{|\vec{r}_{12}|^3} \vec{r}_{12}, \quad (7.1)$$

onde  $G$  é a constante de gravitação,  $|\vec{r}_{12}| = |\vec{r}_1 - \vec{r}_2|$  é a distância entre as massas  $m_1$  e  $m_2$ . É, portanto, uma força atractiva, proporcional a ambas as massas e inversamente proporcional ao quadrado da distância entre os dois corpos. Por outro, a segunda lei de Newton diz-nos que

$$\vec{F} = m\vec{a}, \quad (7.2)$$

pelo que podemos calcular a aceleração sofrida pela massa  $m_2$  devido à massa  $m_1$ :

$$\vec{F}_{21} = -G \frac{m_1 m_2}{|\vec{r}_{12}|^3} \vec{r}_{12} = m_2 \vec{a}_2 \Leftrightarrow \vec{a}_2 = -G \frac{m_1}{|\vec{r}_{12}|^3} \vec{r}_{12}. \quad (7.3)$$

### 7.2 Problema de três corpos

Olhemos melhor para o nosso problema. Temos três corpos: o Sol (com massa  $m_0$  e posição  $\vec{r}_0$ ), Júpiter (com massa  $m_1$  e posição  $\vec{r}_1$ ), e uma sonda espacial (com massa  $m_2$  e posição  $\vec{r}_2$ ). O Sol é bastante pesado quando comparado com Júpiter ou a sonda. Consideramos então que o Sol está fixo, na origem do nosso referencial. Poderíamos também desprezar a influência da sonda em Júpiter, no entanto, preferimos não o fazer para podermos adaptar facilmente o nosso código à órbita de uma lua em torno de um planeta. Se inicialmente considerarmos que a sonda está no plano da órbita de Júpiter e que a sua velocidade inicial também está nesse plano, verificamos que a sonda nunca sai desse plano. Visto isso, desprezamos desde o início

a direcção perpendicular a esse plano. Usamos a equação (7.1) para calcular a aceleração da sonda devido ao Sol e a Júpiter:

$$\begin{aligned}\vec{F}_{20} + \vec{F}_{21} &= -G \frac{m_1 m_2}{|\vec{r}_{12}|^3} \vec{r}_{12} - G \frac{m_0 m_2}{|\vec{r}_{02}|^3} \vec{r}_{02} = m_2 \vec{a}_2 \\ \Leftrightarrow \vec{a}_2 &= -G \frac{m_1}{|\vec{r}_{12}|^3} \vec{r}_{12} - G \frac{m_0}{|\vec{r}_2|^3} \vec{r}_2.\end{aligned}\quad (7.4)$$

De igual modo, a aceleração de Júpiter é dado pela fórmula:

$$\vec{a}_1 = -G \frac{m_2}{|\vec{r}_{21}|^3} \vec{r}_{21} - G \frac{m_0}{|\vec{r}_1|^3} \vec{r}_1. \quad (7.5)$$

Em que a norma de um vector ( $\vec{r} = (x, y)$ ) é dado pela fórmula usual:

$$|\vec{r}| = \sqrt{r_x^2 + r_y^2}, \quad (7.6)$$

### 7.3 Método de Euler

Estamos interessados em calcular a trajectória da sonda e de Júpiter. Sabemos que a variação da posição é dada pela velocidade:

$$\frac{\Delta \vec{r}_1}{\Delta t} = \vec{v}_1 \quad (7.7)$$

$$\frac{\Delta \vec{r}_2}{\Delta t} = \vec{v}_2 \quad (7.8)$$

e por sua vez, a variação da velocidade é dada pela aceleração que depende da posição, como vimos em cima:

$$\frac{\Delta \vec{v}_1}{\Delta t} = \vec{a}_1 \quad (7.9)$$

$$\frac{\Delta \vec{v}_2}{\Delta t} = \vec{a}_2. \quad (7.10)$$

Notem que estas equações são aproximadas. Essa aproximação será tanto melhor quanto menor for o intervalo de tempo  $\Delta t$  considerado. Vejamos em detalhe uma das equações:

$$\frac{\Delta r_x}{\Delta t} = v_x \quad (7.11)$$

$$\Delta r_x = v_x \Delta t, \quad (7.12)$$

ou seja,

$$r_x(t + \Delta t) = r_x(t) + v_x \Delta t \quad (7.13)$$

O método de Euler é baseado neste resultado. Sabendo as posições e as velocidades iniciais, calculamos o valor destas um instante  $\Delta t$  depois. Temos agora novas posições e velocidades, e podemos calcular o próximo passo. E assim, sucessivamente.

## 7.4 O código em Python

---

```

# coding: utf8

from __future__ import division
from math import *
import pylab
from visual import *

UD=7.78412027E+7                                # distância de referência

r2=7.78412027E+8/UD                             # raio (médio) da órbita de Júpiter em km
raio_jup=6.9173e4/UD                           # raio de Júpiter em km
raio_sol=6.95500e5/UD                          # raio do Sol em km
massa_jup=1.8988E+27                           # massa de Júpiter
massa_sol=1.988435E+30                         # massa do Sol
massa=2.e3                                     # massa da sonda em kg
G=(6.67E-20)*3600**2/(UD**3)                  # constante de gravitação universal em kg^-1 UD^3 h^-2

# condições iniciais,
# o vector 'u' contém as posições e velocidades iniciais dos dois corpos:

u=8*[0.]

u[0]=7.57e8/UD                                # coordenada x inicial da sonda
u[1]=4.493e6/UD                              # coordenada y inicial da sonda
vel_i=50650./UD                              # velocidade inicial da sonda
theta_i=70.                                  # ângulo inicial
u[2]=vel_i*cos(theta_i*pi/180.)              # componente x da velocidade da sonda
u[3]=vel_i*sin(theta_i*pi/180.)              # componente y da velocidade da sonda
omega=sqrt(G*massa_sol)/pow(r2,1.5)          # velocidade angular de Jupiter em rad/hora
u[4]=r2                                       # coordenada x inicial de Júpiter
u[5]=0.                                       # coordenada y inicial de Júpiter
u[6]=0.                                       # componente x da velocidade de Júpiter
u[7]=1.*omega * r2                          # componente y da velocidade de Júpiter

# Calcula a variação das velocidades e das posições de Júpiter e do satélite
def variacao(u):

    v=8*[0]

    # Posição da sonda
    v[0]=u[2]      v[1]=u[3]
    # Velocidade da sonda
    v[2]=-G*massa_sol*u[0]/pow(u[0]**2+u[1]**2,1.5)\
        -G*massa_jup*(u[0]-u[4])/pow(u[0]**2+u[1]**2+\
        u[4]**2+u[5]**2-2*(u[0]*u[4]+u[1]*u[5]),1.5)
    v[3]=-G*massa_sol*u[1]/pow(u[0]**2+u[1]**2,1.5)\
        -G*massa_jup*(u[1]-u[5])/pow(u[0]**2+u[1]**2+\
        u[4]**2+u[5]**2-2*(u[0]*u[4]+u[1]*u[5]),1.5)

    # Posição de Júpiter
    v[4]=u[6]

```

```

v[5]=u[7]

    # Velocidade de Júpiter
v[6]=-G*massa_sol*u[4]/pow(u[4]**2+u[5]**2,1.5)\
    +G*massa*(u[0]-u[4])/pow(u[0]**2+u[1]**2+\
    u[4]**2+u[5]**2-2*(u[0]*u[4]+u[1]*u[5]),1.5)
v[7]=-G*massa_sol*u[5]/pow(u[4]**2+u[5]**2,1.5)\
    +G*massa*(u[1]-u[5])/pow(u[0]**2+u[1]**2+\
    u[4]**2+u[5]**2-2*(u[0]*u[4]+u[1]*u[5]),1.5)

    return v

# Método de Euler

def euler(u,dt):
    v=variacao(u)[:]
    # v é a variação de u
    for i in range(8):
        u[i]+=v[i]*dt
    return u

# Escreve as trajectórias para uma lista traj
def trajectoria():
    traj[0].append(u[0])
    traj[1].append(u[1])
    traj[2].append(u[4])
    traj[3].append(u[5])

# Animação:
def anim(t):
    # No início temos que criar os objectos
    if t==0:
        global jupiter, jupiter2          # usamos "global" para que possamos usar
        global sonda, sonda2              # estes objectos das próximas vezes que
        global ref_sol, ref_jup            # correremos esta função

        lado=400
        d0 = r2

    # O display cria uma janela que chamaremos ref_sol. A largura e a altura são igual a lado.
    # O centro é (0,0,0). 'range' define a escala da janela. Desligamos o 'autoscale'
    # e 'exit = 0' evita que o programa pare ao fechar manualmente a janela.
    ref_sol=display(title = 'Orbita no referencial do Sol',
                    x=0, y=0, width=lado, height=lado,center=(0,0,0),
                    range=(1.2*d0,1.2*d0,1.2*d0),autoscale=0, exit=0)

    # definimos 'jupiter' como uma esfera na janela 'ref_sol'.
    jupiter=sphere(display=ref_sol, pos=(u[4],u[5],0),
                    radius=raio_jup, color=color.orange)

    # para ver o rasto (trail) de 'jupiter' usamos:
    jupiter.trail=curve(color=jupiter.color)

```

```

# definimos uma curva chamada 'sonda':
sonda=curve(display=ref_sol, color=color.green)

# definimos uma outra esfera chamada 'sun':
sun=sphere(display=ref_sol, pos=(0,0,0),
            radius=0.05*r2, color=color.red)

# criamos uma segunda janela, para uma animação no referencial de Júpiter:
ref_jup=display(title = 'Orbita no referencial de Jupiter',
                x=lado, y=lado, height=lado, width=lado, uniform=1,
                autoscale=1, autocenter=1, exitr=0)

# criamos uma esfera de nome sonda2 com o respectivo traço
sonda2=sphere(display=ref_jup, pos=(u[0]-u[4],u[1]-u[5],0),
               radius=0.1/UD,color=color.green)
sonda2.trail=curve(color=sonda2.color)

# criamos uma esfera chamada jupiter2 com uma etiqueta
jupiter2=sphere(display=ref_jup, pos=(0,0,0), radius=raio_jup,
                 color=color.orange)
label(pos=jupiter2.pos, text='Jupiter')

# se t maior que zero, actualizamos as posições dos objectos:
else:
    rate(10000) # limitamos a frequência de actualização a 10000 por segundo.
    jupiter.pos=(u[4],u[5],0) # actualizamos a posição de jupiter
    jupiter.trail.append(pos=jupiter.pos) # e também o seu rasto
    sonda.append(pos=(u[0],u[1],0)) # actualizamos a posição de sonda
    jupiter2.pos=(0,0,0) # actualizamos a posição de jupiter2
    sonda2.pos=(u[0]-u[4],u[1]-u[5],0) # actualizamos a posição de sonda2
    sonda2.trail.append(pos=sonda2.pos) # e o seu traço

# inicio programa
# decide se apresenta ou não as animações
choice = raw_input("Queres ver as animações [S/n]?")

if choice=='n' or choice=='N':
    choice=0
else:
    choice=1

t=0.0 # tempo inicial
tmax=5000.0 # tempo de simulação
passo = 0.5 # passo de integração

delta = 100 # intervalo de tempo entre cada registo de dados
traj=[[],[],[],[]] # lista com a trajectória da sonda e de Jupiter
# a ordem é: [x_sonda, y_sonda, x_jupiter, y_jupiter]

trajectoria() # primeiro ponto da trajectória

if choice == 1:

```

```
    anim(t)
t_traj=delta                                     # tempo do próximo ponto da trajectória

while t<tmax:
    u=euler(u,passo)
    t+=passo

    # a animação:
    if choice == 1:
        anim(t)

    if t>=t_traj:                                # escreve um ponto da trajectória quando atinge t_traj
        print 't=',t,'/',tmax
        trajectoria()
        t_traj+=delta

    if u[0]**2+u[1]**2<raio_sol**2:              # testa se caiu ao sol
        print "A sonda caiu no Sol"
        break

    if (u[0]-u[4])**2+(u[1]-u[5])**2<raio_jup**2: # testa se caiu a Júpiter
        print "A sonda caiu em Júpiter"
        break

# depois do ciclo escreve um ultimo ponto
print 't = ',t,'/',tmax trajectoria()

# Desenha a orbita
pylab.title("Orbitas!")
pylab.xlabel("x")
pylab.ylabel("y")
pylab.plot(traj[0],traj[1], label = 'Sonda')
pylab.plot(traj[2],traj[3], label = 'Jupiter')
pylab.legend()
# pylab.savefig("orbita.png")
pylab.show()
```

---



# Apêndice A

## Bibliotecas úteis e exemplos

No capítulo 3 já foram usadas bibliotecas para poder usar funções matemáticas ou fazer gráficos. Neste capítulo vamos mostrar vários exemplos de código que usam bibliotecas que podem ser úteis em várias situações.

### 1) Grafico animado (pylab/matplotlib)

---

```
#coding=utf8

import pylab
import random

# Pedo ao pylab para entrar no modo interactivo (ion = interactive on)
# Assim podemos alterar os gráficos em tempo real, dentro do nosso programa
pylab.ion()

# Cria as listas que representam a nossa trajectória x,y
# inicialmente estamos no ponto (0,0)
x = [0]
y = [0]

# Desenha um linha com a trajectória x,y
# azul (b), com pontos (o) e linhas (-), semitransparente (alpha=0.5)
line, = pylab.plot(x,y,'bo-',alpha='0.5')

# Adiciona um ponto novo à trajectória e desenha-a novamente no ecrã 100 vezes
for i in xrange(100):

    # Adiciona valores as listas com os dados de x e y da nossa trajectória
    # segundo um passeio aleatorio
    p = random.random()*4
    if(p < 1):
        x.append(x[-1]+1)
        y.append(y[-1])
    elif(p < 2):
        x.append(x[-1]-1)
        y.append(y[-1])
```

```

elif(p < 3):
    x.append(x[-1])
    y.append(y[-1]-1)
else:
    x.append(x[-1])
    y.append(y[-1]+1)
    # Muda os valores de x e y da linha que já desenhemos
    # Notem que podem ter comprimentos diferentes do gráfico original
line.set_data(x,y)

# Define os limites do gráfico novamente.
# Se não fizessemos isto ele usaria os limites do primeiro gráfico
pylab.xlim(-20,20)
pylab.ylim(-20,20)

# Desenha o gráfico outra vez
pylab.draw()

```

---

## 2) Vários gráficos animados na mesma janela (pylab/matplotlib)

---

```
#coding=utf8
```

```

import pylab
import random
import numpy

```

*#Rotina que retorna dois vectores com um passeio aleatório de comprimento 'tamanho', que co*

```

def passeio_aleatorio(x0,y0,tamanho):
    #Cria 2 vectores, cada um com comprimento 'tamanho'
    x = numpy.zeros(tamanho)
    y = numpy.zeros(tamanho)
    #O ponto inicial é (x0,y0)
    x[0] = x0
    y[0] = y0

    for i in xrange(1,tamanho):
        p = random.random()*4
        if(p < 1):
            x[i] = x[i-1]+1
            y[i] = y[i-1]
        elif(p < 2):
            x[i] = x[i-1]-1
            y[i] = y[i-1]
        elif(p < 3):
            x[i] = x[i-1]
            y[i] = y[i-1]-1
        else:
            x[i] = x[i-1]
            y[i] = y[i-1]+1

    return x,y

```

```

# # # Programa principal # #

#Dados do programa
num_iteracoes = 100
tamanho = 32

# Pedo ao pylab para entrar no modo interactivo (ion = interactive on)
# Assim podemos alterar os gráficos em tempo real, dentro do nosso programa
pylab.ion()

# Cria a janela com 9x9 polegadas, com cor branca (w=white)
pylab.figure(0,figsize=(9,9),facecolor='w')

# Escreve o título do gráfico pylab.suptitle('Demonstracao passeio aleatorio')

# Reserva espaco para o passeio aleatorio num vector
passeio_x = numpy.zeros((num_iteracoes*tamanho))
passeio_y = numpy.zeros((num_iteracoes*tamanho))
#preenche os primeiros 'tamanho' pontos dos vectores com um passeio aleatorio que começa em (0,0)
passeio_x[0:tamanho],passeio_y[0:tamanho] = passeio_aleatorio(0,0,tamanho)

#Grafico 1: 1 Passeio aleatório (o primeiro de uma matrix de 2x2)
passeio_subplot = pylab.subplot(221)
passeio_subplot.set_title('Passeio aleatorio (1 particula)')
passeio_line, = pylab.plot(passeio_x[0:tamanho],passeio_y[0:tamanho])
#Calcula o desvio quadrático da trajetoria
r_quadrado = passeio_x**2+passeio_y**2

#Grafico 2: Desvio Quadrático Médio para uma particula (o segundo de uma matrix de 2x2)
desvio_subplot = pylab.subplot(222)
desvio_subplot.set_title('Desvio Quadratico Medio (1 particula)')
desvio_line, = pylab.plot(r_quadrado)
desvio_subplot.set_xlim(0,num_iteracoes*tamanho)
desvio_subplot.set_ylim(0,10000)

# Reserva espaço para as trajetorias de 10 particulas em vectores (num_iteracoes*tamanho,10)
passeios_10_x = numpy.zeros((num_iteracoes*tamanho,10))
passeios_10_y = numpy.zeros((num_iteracoes*tamanho,10))

# Preenche os inicios dos vectores com vários passeios aleatórios
for i in xrange(10):
    passeios_10_x[0:tamanho,i],passeios_10_y[0:tamanho,i] = passeio_aleatorio(0,0,tamanho)

#Grafico 3: passeios aleatorios de 10 particulas (o terceiro de uma matrix de 2x2)
passeios_10_subplot = pylab.subplot(223)
passeios_10_subplot.set_title('Passeios Aleatorios (10 particulas)')
passeios_10_lines = pylab.plot(passeios_10_x[0:tamanho,:],passeios_10_y[0:tamanho,:])

# Calcula os desvios quadráticos médios, fazendo a soma dos quadrados das matrizes e
# somando ao longo do eixo das partículas. (eixo das trajetorias = 0, eixo das particulas = 1)
r_quadrado10 = numpy.sum(passeios_10_x**2+passeios_10_y**2,axis=1)

```

```

#Gráfico 4:Desvio quadratico medio para 10 particulas (o quarto de uma matrix de 2x2)
desvios10_subplot = pylab.subplot(224)
desvios10_subplot.set_title('Desvio Quadratico Medio (10 particulas)')
desvios10_line, = pylab.plot(r_quadrado10)
desvios10_subplot.set_xlim(0,num_iteracoes*tamanho)
desvios10_subplot.set_ylim(0,50000)

# Para cada iteração
for i in xrange(num_iteracoes-1):

    #Adiciona mais pontos ao vector do passeio aleatório
    x,y = passeio_aleatorio(passeio_x[(i+1)*tamanho-1],passeio_y[(i+1)*tamanho-1],tamanho)
    passeio_x[(i+1)*tamanho:(i+2)*tamanho] = x
    passeio_y[(i+1)*tamanho:(i+2)*tamanho] = y

    #Actualiza o grafico do passeio aleatorio
    passeio_line.set_data(passeio_x[(i+2)*tamanho:],passeio_y[(i+2)*tamanho:])
    passeio_subplot.set_xlim(-100,100)
    passeio_subplot.set_ylim(-100,100)

    #Recalcula o desvio quadratico médio de 1 partícula e actualiza os dados do grafico
    r_quadrado = passeio_x**2+passeio_y**2
    desvio_line.set_ydata(r_quadrado)

    #Adiciona mais pontos às várias trajetórias dos passeios aleatorios de 10 partículas e
    for j in xrange(10):
        x,y = passeio_aleatorio(passeios_10_x[(i+1)*tamanho-1,j],passeios_10_y[(i+1)*tamanho-1,j],tamanho)
        passeios_10_x[(i+1)*tamanho:(i+2)*tamanho,j] = x
        passeios_10_y[(i+1)*tamanho:(i+2)*tamanho,j] = y

        passeios_10_lines[j].set_data(passeios_10_x[(i+2)*tamanho:],passeios_10_y[(i+2)*tamanho:])
    passeios_10_subplot.set_xlim(-100,100)
    passeios_10_subplot.set_ylim(-100,100)

    #Recalcula o desvios quadratico médio dos passeios das 10 partículas
    r_quadrado10 = numpy.sum(passeios_10_x**2+passeios_10_y**2,axis=1)
    desvios10_line.set_ydata(r_quadrado10)

    # Desenha o gráfico outra vez
    pylab.draw()

```

---

### 3) Gráficos $z=f(x,y)$ e o conjunto de mandelbrot (pylab/matplotlib)

---

```

#coding=utf8

import pylab
import numpy

# Rotina que retorna uma matrix (w,h) com o conjunto de mandelbrot
# uma boa explicação do que é em http://en.wikipedia.org/wiki/Mandelbrot_set
# código retirado de http://www.scipy.org/Tentative_NumPy_Tutorial/Mandelbrot_Set_Example

```

```

y,x = numpy.ogrid[ (y0-dy):(y0+dy):w*1j , (x0-dx):x0+dx:h*1j]
c = x+y*1j
z = c
divtime = maxit + numpy.zeros(z.shape, dtype=int)

    for i in xrange(maxit):
        z = z**2 + c
        diverge = z*numpy.conj(z) > 2**2
# who is diverging
        div_now = diverge & (divtime==maxit) # who is diverging now
        divtime[div_now] = i                 # note when
        z[diverge] = 2                       # avoid diverging too much

    return divtime

#
#
# Programa principal
#
#
# Pedo ao pylab para entrar no modo interativo (ion = interactive on)
# Assim podemos alterar os gráficos em tempo real, dentro do nosso programa
pylab.ion()

# Título pylab.title('Conjunto de Mandelbrot')
# Ponto à volta do qual vamos fazer zoom
x0, y0 = -0.1528,1.0397

mandelbrot_image = pylab.imshow(mandelbrot(64,64,-0.5,0,2,2))

for i in xrange(100):

    #conjunto de mandelbrot
    dx = 2*numpy.exp(-i*0.1)
    dy = 2*numpy.exp(-i*0.1)
    mandelbrot_image.set_data(mandelbrot(64,64,x0,y0,dx,dy))

    # Desenha o gráfico outra vez
pylab.draw()

```

---

#### 4) Como desenhar imagens e animações no ecrã (pygame)

```

#coding=utf8

import random
import math
import pygame
from pygame.locals import * #para podermos usar o QUIT, KEYDOWN e K_ESCAPE

# Documentação do PyGame - http://pygame.org/docs/index.html

# Inicializa o PyGame

```

```

pygame.init()

# Cria uma variável 'screen' que representa a janela, na qual podemos desenhar
# Tem 800x600 pixels
screen = pygame.display.set_mode((800, 600))

# O título da janela
pygame.display.set_caption('Satélites')
# Faz load de uma imagem da terra, para a podermos desenhar
# um ficheiro adequado: http://www.windypundit.com/archives/2005/images/20050528-GoogleEarth
imagem_terra = pygame.image.load('terra.jpg')

# desenha a terra no centro da janela
screen.blit(imagem_terra, (400-imagem_terra.get_width()/2., 300-imagem_terra.get_height()/2.))

# Cria N satélites para orbitar a terra
# cria 4 listas: 2 para a posição (x e y) e 2 para a velocidade (vx e vy)
N = 10
satellite_x = [random.random()*800 for i in xrange(25)]
#http://www.network-theory.co.uk/docs/pytut/ListComprehensions.html
satellite_y = [random.random()*600 for i in xrange(25)]
satellite_vx = [random.random()*10-5 for i in xrange(25)]
satellite_vy = [random.random()*10-5 for i in xrange(25)]

# Cria uma lista de N cores aleatórias
# cada cor sao 3 valores entre 0 e 255, segundo o sistema RGB http://en.wikipedia.org/wiki/
satellite_cor = [(random.randrange(10,250), random.randrange(10,250), random.randrange(10,250)) for i in xrange(N)]

#Guarda o tempo em milissegundos, para podermos apagar o ecrã de 20 em 20 segundos
ultimo_instante = pygame.time.get_ticks()

# Corre em ciclo enquanto o utilizador não tentar sair
while 1:

    #vamos calcular a energia total em cada ciclo. é suposto ficar constante
    #inicia como zero
    energia_total = 0

    # Actualiza as posições dos satélites
    for i in xrange(N):

        #calcula a aceleração devido à força gravítica
        dx = float(satellite_x[i])-400
        dy = float(satellite_y[i])-300.
        r = math.sqrt(dx**2+dy**2) #distância ao centro da terra

        constante = 2500
        #valor arbitrário, neste programa não precisamos de valores realistas

        #aceleracao = 1./r**2, mas decompondo nas duas componentes ax,ay temos:
        ax = -constante*dx/r**3
        ay = -constante*dy/r**3
        dt = 0.25

    #valor arbitrário, neste programa não precisamos de valores realistas.

```

```

    # quanto mais pequeno este valor fôr, mais estáveis serão os resultados
    # (isto é, há menos satélites a sair disparados sem motivo quando se aproximam muito da terra)
    # integração utilizando o método de euler-cromer - http://wiki.vdrift.net/Numerical\_Integration
    satellite_vx[i] += dt*ax
    satellite_vy[i] += dt*ay
    satellite_x[i] += dt*satellite_vx[i]
    satellite_y[i] += dt*satellite_vy[i]

    # Se o satellite se afastar muito, volta a colocá-lo no ecrã com velocidade aleatoria
    if(r > 1000):
        satellite_x[i] = random.random()*800
        satellite_y[i] = random.random()*600
        satellite_vx[i] = random.random()*10-5
        satellite_vy[i] = random.random()*10-5
        print 'Um novo satélite foi gerado. A energia total será diferente'

    # Calcula a energia do satélite e adiciona à total
    massa = 1
    energia_satelite = 1./2.* massa * (satellite_vx[i]**2 + satellite_vy[i]**2) - constante*massa/r
    energia_total += energia_satelite

print 'Energia total = ',energia_total
# Escreve a energia total no título
#pygame.display.set_caption('Satélites - energia total = %f'%energia_total)

# desenha os N satélites como circunferências na janela screen
# na posição (satellite_x[i],satellite_y[i]), com raio 2 e espessura da linha 1
for i in xrange(N):
    pygame.draw.circle(screen, satellite_cor[i], (int(satellite_x[i]),int(satellite_y[i])), 2,1)

# Pergunta ao PyGame se o utilizador tentou interagir com a janela e percorre todos os eventos
for event in pygame.event.get():
    if event.type == QUIT: #Se ele tentou fechar a janela
        quit()
    elif event.type == KEYDOWN and event.key == K_ESCAPE:
        #Se carregou numa tecla e essa tecla era o Escape
        quit()

pygame.display.flip() #actualiza o conteúdo da janela

# Se ja passaram 20 segundos, limpa a janela
este_instante = pygame.time.get_ticks()
if(este_instante - ultimo_instante > 20000):
    #preenche a janela de preto
    screen.fill((0,0,0))
    #desenha a terra
    screen.blit(imagem_terra, (400-imagem_terra.get_width()/2.,300-imagem_terra.get_height()/2))

    # faz com que isto so aconteça outra vez daqui a 10 segs
    ultimo_instante = este_instante

```

---

## 5) Interface gráfico básico (Tkinter)

---

```
#coding=utf8

# Tkinter é uma biblioteca que nos permite fazer interfaces gráficas
import Tkinter
import math

#Rotinas

# Rotina a ser chamada quando se clica no botão 'sair'
def sair():
    quit()

# Rotina a ser chamada quando se clica no botão 'calcula'
def calcula():
    # Diz ao python que queremos acesso a estas variáveis globais, que foram definidas fora
    global texto_vx, texto_h, texto_t, texto_distancia
    # Obtém o texto que o utilizador escreveu no interface gráfico (texto_XX.get()) e
    vx = float(texto_vx.get())
    h = float(texto_h.get())
    # Faz os cálculos
    t = math.sqrt(2*h/9.8)
    distancia = vx*t
    # Escreve o texto no interface gráfico. Notem que é preciso convertê-lo para uma cadeia de
    texto_t["text"] = str(t)
    texto_distancia["text"] = str(distancia)

# Cria a janela
janela = Tkinter.Tk()
# Muda o título
janela.title('Projectil')

# Cria um contendor dos elementos do interface gráfico (invisível)
contentor = Tkinter.Frame(janela) # Adiciona o contentor
contentor.pack()

#Cria os vários elementos do interface gráfico: texto simples (Label) ou caixas de edição o

titulo_vx = Tkinter.Label(contentor, text='vx (m/s)') titulo_vx.pack()
texto_vx = Tkinter.Entry(contentor, justify=Tkinter.RIGHT)
texto_vx.insert(Tkinter.END, '1') #Escreve '1' na caixa de texto, inicialmente
texto_vx.pack()

titulo_h = Tkinter.Label(contentor, text='altura (m)') titulo_h.pack()

texto_h = Tkinter.Entry(contentor, justify=Tkinter.RIGHT)
texto_h.insert(Tkinter.END, '1')
texto_h.pack()

titulo_t = Tkinter.Label(contentor, text='t (s)') titulo_t.pack()
texto_t = Tkinter.Label(contentor, text='-')
texto_t.pack()
```



```
titulo_distancia= Tkinter.Label(contentor,text='Distancia (m)')
titulo_distancia.pack()
texto_distancia = Tkinter.Label(contentor,text='-') texto_distancia.pack()

# Adiciona dois botões, os quais chamam as funções calcula ou sair quando são clickados

botao1 = Tkinter.Button(contentor, text="Calcula", command=calcula)
botao1.pack()
botao2 = Tkinter.Button(contentor, text="Sair", command=sair) botao2.pack()

# Calcula os resultados com os valores iniciais
calcula()

# Corre o ciclo principal da janela. Isto quer dizer que a janela vai esperar
# que o utilizador interaja com ela para fazer alguma coisa. A execução vai ficar
# parada nesta linha até que a janela seja fechada.
janela.mainloop()
```

---