Ciências ULisboa

Faculdade de Ciências da Universidade de Lisboa

Departamento de Engenharia Geográfica, Geofísica e Energia

Laboratório Numérico (em python)

Pedro M A Miranda

VERSÃO 2018/03/01

Indice

1 Introdução ... 3
1.1 Software e bibliografia ... 3
1.2 Contactos ... 4
2 Números, Operações e Erros ... 5
2.1 Objetos ... 5
2.2 Números inteiros (e lógicos) ... 5
2.3 Números de vírgula flutuante ... 6
2.4 Erros e estabilidade ... 7
2.5 Aplicações ... 9
2.6 Números, listas, vetores e matrizes ... 12
2.7 Revisão ... 13
3 Ajustamento de dados experimentais por regressão linear ... 15
3.1 Problema ... 15
3.2 Formulação matemática ... 15
3.3 Métodos numéricos ... 16
3.4 Exemplos ... 16
3.5 Revisão ... 19
4 Interpolação ... 21
4.1 Interpolação Polinomial. ... 21
4.2 Interpolação por funções spline. ... 24
4.3 Revisão ... 25
5 Raízes de equações não lineares ... 26
5.1 Equilíbrio térmico de um painel solar ... 26
5.2 Formulação matemática: ... 26
5.3 Métodos numéricos ... 27
5.3.1 Método da bissecção ... 27
5.3.2 Método de Newton-Raphson ... 30
5.3.3 Solução genérica ... 31
6 Processamento de dados multidimensionais ... 32
6.1 Funções simples de leitura e escrita de tabelas de dados ... 32
6.2 Gráficos bidimensionais ... 34
7 Sistemas de equações lineares ... 39
7.1 Identificação do problema e formulação matemática ... 39
7.2 Métodos numéricos ... 39
7.3 Eliminação de Gauss ... 40
7.3.1 Algoritmo sem pivot ... 40
7.4 Soluções recorrendo a funções pré-existentes ... 42
8 Integração numérica ... 44
8.1 Métodos numéricos ... 44
8.1.1 Cálculo de integrais por interpolação entre pontos regularmente espaçados ... 44
8.1.2 Método de Monte Carlo ... 45

8.2 Aplicações....................46
9 Solução de equações diferenciais ordinárias com condições fronteira num ponto ...53
9.1 Equações diferenciais ordinárias....................53
9.1.1 Lei de Newton do arrefecimento....................53
9.2 Integração das equações da Mecânica....................53
9.3 Métodos numéricos (Método de Euler)....................55
9.3.1 Comentários....................56
9.4 Aplicações....................56
10 Equações às derivadas parciais independentes do tempo....................66
10.1 Identificação do Problema e Formulação Matemática....................66
10.1.1 Equação de Poisson do potencial de uma distribuição contínua de carga....................66
10.1.2 Equação de Laplace aplicada ao equilíbrio térmico de uma placa....................66
10.2 Formulação geral....................66
10.3 Métodos numéricos (Relaxação)....................67
10.4 Aplicações....................69
10.5 Comentários e revisão....................73

1 Introdução

Assume-se neste curso que os estudantes têm um conhecimento básico de utilização de computadores, incluindo nomeadamente as aplicações de *office* geralmente disponíveis em ambiente Windows ou Linux e possuem as competências genéricas necessárias (e.g. manejamento de ficheiros, edição, etc.).

O curso proposto baseia-se na ***solução de problemas*** relevantes para estudantes de Engenharia, Física Aplicada e Ciências da Terra. Os problemas pretendem ser compreensíveis para estudantes do primeiro ano destas áreas, ainda que alguns se refiram a matérias só cobertas em detalhe em disciplinas do 2º ou mesmo do 3º ano. De facto parece-nos importante introduzir logo no início do curso metodologias de Laboratório Numérico, tornando-as disponíveis para um leque alargado de problemas, mesmo para problemas que só serão completamente compreendidos em semestres posteriores.

Para a resolução numérica de cada problema existem sempre inúmeras metodologias disponíveis, com vantagens e inconvenientes. Neste curso não se fará uma discussão detalhada dessas opções, nem se tentará estabelecer as propriedades das soluções propostas, assunto de uma disciplina formal de Análise Numérica. Em geral, recorre-se a metodologias simples mas eficazes.

Os problemas propostos podem ser resolvidos satisfatoriamente com diferentes metodologias, no contexto de diferentes linguagens de programação. No âmbito deste curso optou-se por recorrer a uma aplicação interativa com crescente relevância no ensino e na investigação em todos os domínios: o PYTHON.

O PYTHON é um produto de distribuição livre (*open-source*) extremamente popular no ensino da Engenharia e de diversas áreas das Ciências, em todo o mundo, permitindo resolver eficientemente problemas muito variados, incluindo as componentes de cálculo numérico, cálculo simbólico e representação gráfica. De facto, muitos problemas razoavelmente complexos resolvem-se em PYTHON com muito poucos comandos. Apesar da sua grande abrangência e complexidade, o PYTHON é no entanto uma linguagem genérica, disponível gratuitamente em todas as plataformas relevantes de computação.

O PYTHON apresenta, no entanto, uma importante limitação:

> No caso de problemas verdadeiramente complicados, o tempo de cálculo pode ser proibitivo. O PYTHON é uma linguagem "interpretada", sendo os comandos executados imediatamente. Pelo contrário, em C ou em Fortran os programas são previamente processados por um compilador e são sujeitos a processos de otimização automática que aumentam imensamente a sua velocidade no momento da execução.

Por estas e outras razões pode ser conveniente ser capaz de recorrer também a uma linguagem genérica de alto nível, como o C ou o Fortran90, em especial em aplicações muito complexas como Mecânica de Fluidos, Meteorologia, Geofísica, Oceanografia, etc. Felizmente, muitos dos conceitos aqui introduzidos são também aplicáveis em linguagens compiladas.

1.1 Software e bibliografia

O PYTHON é uma linguagem muito rica, caracterizada pela existência de inúmeras funções desenvolvidas para a solução de problemas diferenciados. Neste texto, não se tentará fazer uma descrição sistemática da linguagem, e muito menos da extensa biblioteca de funções. Tratando-se de

um curso de resolução de problemas, ir-se-ão introduzindo instruções e características da linguagem à medida que vão fazendo falta.

Infelizmente, existem 2 versões de PYTHON não totalmente inter-compatíveis, mas que (felizmente) podem ser instaladas em paralelo no mesmo computador, sem conflito. Genericamente, existem vantagens na versão 3, aquela que será aqui utilizada, mas existem diversas funções disponíveis na internet que só funcionam na versão 2. Por essa razão, pode ser necessário recorrer pontualmente à versão mais antiga.

A utilização do PYTHON é facilitada se se recorrer a instalações que incluem não só o interpretador base (python) mas o interpretador interativo (IPython), um visualizador de variáveis em tempo real e um editor especializado, integrados numa interface gráfica completa em que se podem visualizar saídas numéricas e gráficas. A distribuição aqui utilizada é a ***anaconda***, disponível gratuitamente para todos os sistemas de operação, que inclui a interface gráfica ***spyder***.

Dada a grande relevância do python, é facílimo encontrar ajuda para a resolução de qualquer problema por simples pesquisa em motor de busca. Existem também inúmeros livros disponíveis na internet.

1.2 Contactos

São muito bem-vindas correções a estas notas e sugestões de modificação.

pmmiranda@fc.ul.pt

2 Números, Operações e Erros

2.1 Objetos

O PYTHON é uma linguagem de processamento de ***objetos***, sendo estes estruturas muito genéricas de dados, que podem incluir conjuntos de números (de diversos tipos), textos e até pedaços de código. Esta metodologia de processamento permite frequentemente o desenvolvimento de códigos muito compactos e elegantes, mas a sua utilização plena sai do âmbito do presente curso. Só ocasionalmente utilizaremos este tipo de conceitos. Muitos dos objetos, ou parte desses objetos, são, no entanto, constituídos por números, tal como acontece em todas as linguagens de programação, e o processamento desses números é central nas aplicações científicas.

2.2 Números inteiros (e lógicos)

Todas as instruções executadas por um computador têm de ser traduzidas em sequências de operações elementares compreendidas pela Unidade de Processamento Central (CPU, i.e., pelo(s) Microprocessador(es)). O leque de instruções elementares disponíveis varia muito de máquina para máquina, dependendo da sua arquitetura e da existência ou não de processadores especializados opcionais. Para o utilizador normal aquilo que se passa a esse nível é, num certo sentido, e felizmente, pouco relevante, dependendo a sua utilização do computador fundamentalmente dos recursos fornecidos pelo *software*, isto é, pela linguagem de alto nível ou por outra aplicação utilizada.

No entanto, é claro que os recursos oferecidos pelo *software* são fundamentalmente condicionados pelo modo de operação do computador a um nível mais baixo. Os computadores (digitais) são construídos para obter um elevado grau de eficiência na realização de operações matemáticas sobre números com um número bem determinado de dígitos (na base 2) e segundo regras bem definidas. Nesse contexto só podem ser representados exatamente um subconjunto dos números usuais (inteiros, reais, complexos, etc.) e são de esperar limitações e, eventualmente, erros, nas operações a efetuar.

A afirmação anterior não implica que não se possa utilizar computadores para realizar operações exatas com números reais ou com qualquer precisão pretendida, mas, tão só, que tais operações serão necessariamente muito menos eficientes. Na verdade existem aplicações que permitem fazer isso mesmo, ou até realizar operações simbólicas sobre expressões matemáticas, nomeadamente no âmbito do PYTHON. No entanto, na forma mais eficiente, as linguagens de programação limitam-se a oferecer um leque limitado de possibilidades à representação de números e operações.

Internamente, os números são sempre representados em computadores como sequências de algarismos binários (0 ou 1), que correspondem, cada um deles, aos dois estados possíveis do elemento físico elementar em que se encontram armazenados. Quer isto dizer que os números armazenados num computador utilizam como base natural a ***base binária***. Cada algarismo binário, ou a correspondente *célula* de armazenamento, designa-se geralmente por ***bit***. Todas as operações realizadas por um computador são, por isso, em última análise, operações sobre esses bits e podem ser entendidas como sequências de operações ***lógicas*** (*0* ou *1* é o mesmo que *verdadeiro* ou *falso*), estudadas com base na *Álgebra de Boole*.

Por motivos práticos, é conveniente agrupar os bits em conjuntos. Assim o conjunto de 8 bits designa-se por ***byte***. Se se considerar que um byte constitui um número inteiro na base 2 é claro que um byte poderá representar qualquer número inteiro positivo entre 0 (00000000) e 255 ($11111111=1\times2^7+1\times2^6+1\times2^5+1\times2^4+1\times2^3+1\times2^2+1\times2^1+1\times2^0$). Os computadores atuais atuam

preferencialmente sobre conjuntos de bytes, sendo os respetivos microprocessadores designados em função do comprimento dos seus registos (16 bits, 32 bits ou 64 bits). As diferentes linguagens permitem definir variáveis com diversos comprimentos (geralmente de 2, 4 ou 8 bytes).

Do que foi dito anteriormente depreende-se que é muito fácil representar números ***inteiros*** e ***lógicos***. Assim, um número lógico só precisa de ocupar um bit (embora ocupe mais na prática, porque é mais eficiente manejar bytes ou mesmo grupos de bytes) e um número inteiro ocupará um número dado de bits na representação binária. Neste último caso, se se reservar um bit para o sinal (positivo ou negativo), uma sequência de n bits $x_1x_2...x_n$ será dado por:

$$\pm(x_2 \times 2^{2n-2} + x_3 \times 2^{2n-3} + \cdots + x_{n-1} \times 2^2 + x_n) \qquad (2\text{-}1)$$

Assim, é claro que a representação de números inteiros e lógicos é ***exata***. No entanto, nem todos os números inteiros podem ser representados dada a existência de um número finito de bits. Diz-se que a representação dos números inteiros tem um ***alcance*** finito. No caso de se utilizarem 4 bytes para representar um inteiro (positivo ou negativo) o alcance será então dado por:

$$-2^{31} < N < +2^{31} - 1 \qquad (2\text{-}2)$$

em que se reservou um bit para o sinal e se tomou em consideração a existência do 0 (daí a assimetria do alcance).

Uma outra questão é levantada pela aritmética, isto é pelo resultado das operações efetuadas sobre estes números. No caso das operações lógicas, é claro que elas não levantam qualquer problema. No caso das operações inteiras há, porém, um problema resultante do alcance finito da representação: quando de uma operação resultar um número fora do alcance, o resultado não pode ser obtido, está-se nesse caso numa situação de ***overflow***.

Por outro lado deve notar-se que a aritmética inteira é obrigada a produzir, a partir de qualquer operação, um resultado inteiro. No caso da divisão entre dois números inteiros, em que o dividendo não seja um múltiplo do divisor, o resultado é constituído pela parte inteira da razão e por um resto (inteiro). É preciso indicar explicitamente qual dos dois números inteiros referidos (a razão ou o resto) se pretende obter.

2.3 Números de vírgula flutuante

Considere-se agora o problema da representação de números reais não inteiros. Esses números são geralmente representados na forma, dita de **vírgula flutuante (*float*)**:

$$m \times 2^{e-E} \qquad (2\text{-}3)$$

em que *m* (***mantissa***) e *e* (***expoente***) são dois números inteiros, positivos ou negativos. Assim, procede-se à representação de números não inteiros sob a forma de pares de números inteiros. (Nota: a expressão anterior inclui, para cada precisão, uma constante aditiva *E* (***viés***), necessária, nomeadamente, para centrar o alcance).

Os números de vírgula flutuante são ***dízimas finitas na base 2*** (a que correspondem geralmente dízimas infinitas periódicas na base 10). São, portanto, um subconjunto dos números racionais. Os

números de vírgula flutuante têm, tal como os inteiros, um ***alcance***, que é função do número de bits atribuído ao expoente, e uma ***precisão***, ou número de ***algarismos significativos***, que é função do número de dígitos atribuídos à mantissa. Tradicionalmente referem-se como de ***precisão simples*** números de vírgula flutuante com 4 bytes (32 bits) dos quais 3 (24 bits) são reservados para a mantissa e 1 (8 bits) se destina ao expoente. Estes números têm, ***na base decimal***, um pouco mais de 7 algarismos significativos (o número não é exato nesta base) e um alcance aproximadamente dado por:

$$-10^{+38} < x < -10^{-38}, x = 0, 10^{-38} < x < 10^{+38} \tag{2-4}$$

Nota-se que neste caso existe um valor máximo absoluto que pode ser representado mas também um valor mínimo absoluto, excetuando o caso particular do número 0. Assim, são possíveis condições de ***overflow***, quando o resultado de uma operação é um número com valor absoluto excessivamente grande e de ***underflow***, quando esse resultado é um número de valor absoluto excessivamente pequeno. Neste último caso, as aplicações podem arredondar automaticamente para 0.

Contrariamente à aritmética inteira, a aritmética de vírgula flutuante só excecionalmente produz resultados exatos. Ao erro introduzido pelas limitações de precisão (comprimento finito da mantissa) chama-se geralmente ***erro de arredondamento*** (*roundoff*).

2.4 Erros e estabilidade

Assim, as operações em vírgula flutuante são normalmente afetadas por um erro de arredondamento. Esse erro é inevitável e só pode ser reduzido mediante a utilização de representações com maior precisão, mas nunca eliminado completamente. Numa sequência de operações de vírgula flutuante é de esperar que se observe um crescimento do erro acumulado e pode provar-se estatisticamente que esse crescimento é, em média, inevitável.

Para além do erro de arredondamento, inevitável, que resulta da representação dos números, existe um outro tipo de erro que resulta da utilização de expressões aproximadas (por exemplo, séries truncadas) para a avaliação de um dado resultado. Este tipo de erro é função do *software* utilizado e designa-se por ***erro de truncatura***. O erro de truncatura pode (e deve) ser controlado pelo programador.

Em certos casos, apesar de as fórmulas utilizadas serem corretas, no sentido em que produziriam o resultado exato com aritmética exata, o erro pode crescer descontroladamente (exponencialmente) tornando os resultados obtidos completamente inúteis. Tal crescimento resulta de uma interação entre o erro de arredondamento e o erro de truncatura. Nesse caso diz-se que se está na presença de ***instabilidade numérica***. A principal preocupação da Análise Numérica consiste exatamente em prevenir a ocorrência desta situação.

Um exemplo de obtenção de um resultado afetado de um erro elevado é dado pela utilização da ***fórmula resolvente*** usual da equação do segundo grau:

$$y = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{2-5}$$

Se $b^2 \gg |4ac|$ a expressão anterior produzirá um resultado com um erro relativo muito elevado para uma das raízes, devido ao erro de arredondamento. Se $b > 0$ o erro será muito elevado para a raiz

de maior valor (sinal positivo no radical). Caso contrário será a outra raiz a aparecer afetada por um erro elevado. A título de exemplo considere-se a equação:

$$-y^2 + 88550000y - 1 = 0 \tag{2-6}$$

Com 8 algarismos significativos, as raízes são:

$$\begin{cases} y_1 = 88550000 \\ y_2 = 1.1293055 \times 10^{-8} \end{cases} \tag{2-7}$$

No entanto, utilizando a fórmula resolvente e com aritmética em dupla precisão (o standard em PYTHON, correspondente a cerca de 16 dígitos) obtém-se:

$$\begin{cases} y_1 = 88550000 \\ y_2 = 1.4901161194 \times 10^{-8} \end{cases}$$

Verificando-se, portanto, que a raiz de menor valor absoluto vem afetada de um erro relativamente grande (da ordem dos 30%), muitíssimo superior ao erro de arredondamento esperado.

O crescimento anormal do erro de arredondamento observado neste exemplo resulta da diminuição do número de algarismos significativos quanto se subtraem dois números de valor muito próximo, visto que o resultado tem ordem de grandeza muito inferior à de cada uma das parcelas. Na verdade, a raiz afetada por um erro elevado é a que resulta da escolha do sinal positivo para o radical presente no numerador da fórmula, ocorrendo a diminuição do número de algarismos significativos na subtração: $-b + \sqrt{b^2 - 4ac}$. É claro que se b fosse negativo o cancelamento aconteceria na outra raiz.

No caso da equação do segundo grau o problema pode ser evitado recorrendo à fórmula resolvente alternativa:

$$y = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}} \tag{2-8}$$

Comparando as expressões (2-5) e (2-8) nota-se que se ocorrer cancelamento na primeira raiz da expressão (2-5) é de esperar cancelamento na segunda raiz da expressão (2-8). Assim, pode utilizar-se a expressão mais conveniente para calcular cada uma das raízes, evitando o crescimento do erro.

2.5 Aplicações

Exemplo 2-1 Resolução da equação do 2º grau

Resolva a equação $y^2 + 885500y - 1 = 0$.

A sequência de comandos seguintes realiza o cálculo proposto anteriormente em PYTHON. Estes comandos podem ser executados diretamente na janela de comandos (IPython) ou executados a partir do editor num ficheiro '.py'.

```
#eq2grau.py
from math import sqrt
a=-1.;b=88550000.;c=-1.;
y1=(-b+sqrt(b*b-4*a*c))/(2*a)
y2=(-b-sqrt(b*b-4*a*c))/(2*a)
y1a=2*c/(-b-sqrt(b*b-4*a*c))
y2a=2*c/(-b+sqrt(b*b-4*a*c))
print('Raiz 1 standard=',y1,' Raiz 1 Alternativa=',y1a)
print('Raiz 2 standard=',y2,' Raiz 2 Alternativa=',y2a)

>>Raiz 1 = 1.4901161193847656e-08  Sol Alternativa= 1.1293054771315643e-08
>>Raiz 2 = 88549999.99999999  Sol Alternativa= 67108864.0
```

Tratando-se do primeiro *script* PYTHON utilizado neste texto vamos ter atenção a vários detalhes:

(a) Inicia-se o script com o comando **`import`**. O uso de funções requer a sua importação. O módulo **`math`** inclui um conjunto básico de funções matemáticas, mas neste caso só precisamos da função **`sqrt`**.

(b) O *script* consiste numa sequência de instruções em linhas diferentes ou, quando na mesma linha, separadas por ";".

(c) Neste *script* utilizam-se várias variáveis definidas pelo utilizador (a,b,c,y1,y2,y1a,y2a). Estas variáveis não necessitam de ser previamente declaradas. O tipo de variável (int, single, double, etc.) é atribuído automaticamente, em, função do valor que lhe é atribuído;

(d) As variáveis calculadas só são mostradas se tal por pedido, por exemplo com o comando *print*. O output desse comando está apresentado no exemplo, em linhas começadas por ">>", para o distinguir do texto do script.

Muita atenção: em PYTHON uma variável pode ser classificada automaticamente como *int* (inteira) e daí resultar o seu processamento em aritmética inteira (e o PYTHON 2 e 3 podem apresentar comportamentos diferentes!). Assim "1" será interpretado como *int*, mas "1." será interpretado como *float* (floating point de 64 bits).

As variáveis PYTHON são designadas por textos (como "y1"), sendo o caracter inicial uma letra ou o símbolo '_' (*underscore*). É importante notar que as letras maiúsculas não são equivalentes às correspondentes minúsculas ("y1" é diferente de "Y1").

No problema do Exemplo 2-1, eram conhecidas a raízes exatas e podíamos concluir que a solução pretendida era constituída pelo par (y1a,y2). Podemos recorrer a uma função mais avançada do PYTHON, capaz de resolver equações polinomiais com métodos simbólicos, para confirmar essa solução e calcular o erro de cada uma das fórmulas resolventes. Vamos também usar este exemplo

para melhorar a forma de utilização da função *print*, recorrendo à possibilidade de escolha do formato da apresentação dos resultados numéricos.

Considere-se então o seguinte código adicional, a acrescentar ao script anterior:

```
#eq2grauB.py
from sympy import Symbol,solve
y=Symbol('y')
solution=solve(a*y**2+b*y+c,y)
print('sympy solution=',solution)
print('Erro Raiz 1=%23.16e Alternativa=%23.16e ' % (y1-solution[0],y1a-
      solution[0]))
print('Erro na Raiz 2=%23.16e Alternativa= %23.16e ' % (y2-solution[1],y2a-
      solution[1]))

>>sympy solution= [1.12930547713156e-8, 88550000.0000000]
>>Erro Raiz 1= 3.6081064225320135e-09 Alternativa=  0.0000000000000000e+00
>>Erro Raiz 2= 0.0000000000000000e+00 Alternativa= -2.1441135999999985e+07
```

No código anterior importaram-se 2 funções (`Symbol` e `solve`) do módulo `sympy`. Os parâmetros `a,b,c` são os números já definidos. `y` é declarado como uma variável simbólica. A função `solve` resolve a equação polinomial $ay^2 + by + c = 0$. O resultado desta função, a variável `solution`, inclui as duas raízes na forma de um objeto PYTHON designado por `list` (lista). Acede-se a cada um dos membros da lista por intermédio do seu índice, neste caso [0] ou [1]. **Em PYTHON todas as sequências são ordenadas a partir do número 0.**

Na forma mais elaborada agora utilizada para a função `print` ela tem a forma:

```
print(‘texto %23.16e mais texto %10.5f’ % (x1,x2))
```

No texto entre `‘ ’` o símbolo `%` indica o local de escrita de uma variável, e o texto imediatamente à sua direita o formato (`%23.16e` significa 23 caracteres em notação “científica” com 16 algarismos à direita do ‘`.`’, `%10.5f` significa notação de vírgula flutuante com 5 algarismos à direita do ‘`.`’). A lista a escrever aparece na forma `(x1,x2)` separada do texto inicial por um novo símbolo `%`.

Exemplo 2-2 Erros de arredondamento

Um outro exemplo de erros de arredondamento devidos a perda de algarismos significativos em subtração de números próximos é dado no cálculo da função $(x-1)^7$ *na vizinhança de 1. Um resultado preciso é apresentado no painel superior da Figura 2-1. No painel inferior apresenta-se um resultado alternativo, obtido por desenvolvimento do binómio, equivalente em aritmética exata,* $(x-1)^7 = x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1$*), mas que dá origem a*

um resultado extremamente impreciso (em 40 ordens de grandeza!) devido à subtração sucessiva de números comparáveis.

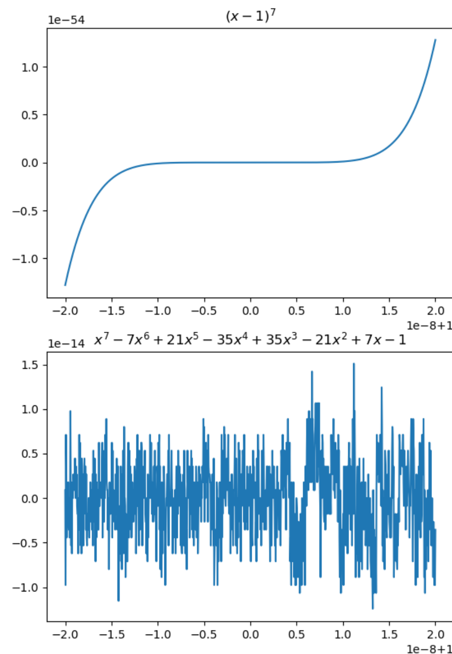

Figura 2-1 Duas formas de calcular $(x-1)^7$, evidenciando erros de arredondamento. Notar a forma condensada como é representada a escala no eixo dos xx, sem perda de algarismos significativos.

Os resultados apresentados na Figura 2-1 foram obtidos com a sequência de comandos:

```
#precisionEx.py
import matplotlib.pyplot as plt
import numpy as np

x=np.linspace(0.99999998,1.00000002,1000)
f1=(x-1)**(7)
f2=x**7-7*x**6+21*x**5-35*x**4+35*x**3-21*x**2+7*x-1

plt.close('all')
plt.subplot(2,1,1)
plt.plot(x,f1)
plt.title(r"$(x-1)^{7}$")
plt.subplot(2,1,2)
plt.plot(x,f2)
plt.title(r"$x^{7}-7x^{6}+21x^{5}-35x^{4}+35x^{3}-21x^{2}+7x-1$")
```

Nas duas primeiras linhas procede-se à importação de 2 módulos extremamente importantes. O módulo **`matplotlib.pyplot`** contém uma biblioteca de funções gráficas 2D e 3D que vamos utilizar frequentemente para representar resultados de cálculos. O módulo **`numpy`** é ainda mais importante pois ele contém a definição dos objetos básicos da álgebra (e.g. vectores, matrizes, e tabelas de maior dimensão) e um número alargado de funções para manipular tais objetos. Em ambos os casos estamos a importar a totalidade das funcionalidades disponíveis, atribuindo-lhe um nome simplificado (**`np`**, **`plt`**) que será utilizado de seguida.

Assim, por exemplo, a instrução **x=np.linspace(0.99999998,1.00000002,1000)** utiliza a função **linspace** do módulo **np** (**numpy**) para criar um objeto **x** do tipo vetor, i.e. um **array** 1D **float**, com o primeiro elemento **x[0]=0.99999998**, o último elemento **x[999]= 1.00000002**, num total de **1000** elementos. As instruções

```
f1=(x-1)**(7)
f2=x**7-7*x**6+21*x**5-35*x**4+35*x**3-21*x**2+7*x-1
```

criam os novos ***vetores*** **f1** e **f2**, com a mesma dimensão e tipo de **x**, utilizando as duas fórmulas equivalentes (para aritmética exata). Deve notar-se que se trata de uma operação vetorial em que todos os elementos do vetor são calculados sem necessidade de realizar um ciclo: esta é uma característica extremamente eficiente das operações com objetos **numpy**.

No que se refere às instruções gráficas, a instrução **plt.close('all')** fecha todas as figuras que estejam abertas. A instrução **plt.subplot(2,1,1)** indica que se vai realizar um conjunto de 2 gráficos (2 linhas, 1 coluna) e que este é o primeiro gráfico (superior). A instrução **plt.plot(x,f1)** executa o gráfico da função $(x-1)^7$, nos pontos definidos pelo vetor **x**. A instrução **plt.title(r"$(x-1)^{7}$")** acrescenta um título a esse gráfico, utilizando um processador de expressões matemáticas que permite colocar expoentes, letras gregas, etc. Note-se que o texto a imprimir é delimitado, no interior dos delimitadores normais de texto **" "** (ou **' '**) por **$$**.

Incidentalmente, é interessante notar que o desenvolvimento do binómio pode ser calculado diretamente com uma instrução do módulo simbólico **sympy**. O código seguinte foi executado diretamente na consola IPython:

```
In [42]: from sympy import expand,Symbol; x=Symbol('x'); expand((x-1)**7)
Out[42]: x**7 - 7*x**6 + 21*x**5 - 35*x**4 + 35*x**3 - 21*x**2 + 7*x - 1
```

2.6 Números, listas, vetores e matrizes

Neste capítulo focámos o nosso interesse na representação e processamento de números, algo que constitui a base da computação, iniciando uma exploração dos problemas desse processo no âmbito da linguagem PYTHON. O que é que aprendemos?

Há vários tipos de números (inteiros, racionais, reais, complexos) e grupos de números podem dar origem a objetos matemáticos estruturados (vetores, matrizes, e outros). Apesar de todos estes números poderem ser representados computacionalmente, essa representação tem limitações que resultam da natureza finita dos recursos computacionais e se traduzem em ***erros de representação*** e em ***erros de cálculo***. A manutenção desses erros dentro de limites aceitáveis é uma das preocupações permanentes do cálculo numérico.

Os números são representados em PYTHON na forma de diversos tipos de ***objetos***. Quando uma instrução PYTHON atribui a uma dada variável um número, cria um objeto identificado pelo nome dessa variável dentro de um conjunto disponível de ***tipos***. A instrução "**x=1**" cria a variável **x**, de tipo **int** (variável inteira), mas a instrução "**y=1.**" cria a variável y do tipo **float**. Em caso de dúvida, a função **type()** indica o tipo de variável criado. O exemplo seguinte foi calculado diretamente na consola IPython:

```
In [51]:x=1;y=1.;type(x),type(y)
Out[51]: (int, float)
```

No que se refere à representação de conjuntos de números, o PYTHON oferece duas alternativas básicas: os objetos **list** e **array**. A lista pode ser qualquer agregado de números e textos, mesmo de tipos diferentes e pode ser criada de forma explícita como **A=[1,1.,3.14,'text']** (lista heterogénea), **B=[1,2,3,4,5]** (lista de inteiros) ou **C=[1.,2.,3.,4.,5.]** (lista de "reais" i.e. float). Trata-se de um procedimento expedito e muito genérico, mas estes objetos não podem ser processados de forma eficiente como estruturas algébricas. Alternativamente, os objetos B e C podem ser também criados na forma **B1=np.array([1,2,3,4,5])** (vector de inteiros de 32 bit), **C1=np.array([1.,2.,3.,4.,5.])** (vector de float de 64 bit), admitindo que se importou o módulo numpy com a designação **np**. B1 e C1 podem ser processados como vetores, de forma muito eficiente, utilizando todas as funções numpy aplicáveis.

Apesar de as listas e vetores poderem ter o mesmo conteúdo *aparente*, i.e. a mesma sequência de números, as suas regras de operação são muito diferentes. Em geral, o cálculo científico baseia-se em vetores (ou matrizes). O código seguinte exemplifica as diferenças fundamentais entre estes dois tipos de objetos:

```
#listVSarray.py
import numpy as np
A=[1,2,3];B=[4,5,6] #listas
C=np.array([1,2,3]);D=np.array([4,5,6]) #arrays np
print('list ',A,'+list ',B,' =',A+B)
print('np.array ',C,'+ np.array',D,'=',C+D)

>>list  [1, 2, 3] +list  [4, 5, 6]  = [1, 2, 3, 4, 5, 6]
>>np.array  [1 2 3] + np.array [4 5 6] = [5 7 9]
```

No código anterior introduziram-se ***comentários***, i.e. textos não interpretados pelo PYTHON, sempre à direita do caracter **#**.

A função **np.array** pode definir vetores, matrizes ou agrupamentos de números de maior dimensão. Por exemplo:

```
M=np.array([[1,2,3],[4,5,6],[7,8,9]])
```

cria a matrix M:

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

2.7 Revisão

Neste capítulo introduzimos alguns conceitos elementares de cálculo numérico, mostrando como eles podem ser explorados em PYTHON. O PYTHON processa estruturas de dados designadas por ***objetos***, onde se incluem variáveis escalares de diversos ***tipos***, representando números inteiros, lógicos, de vírgula flutuante, complexos, e estruturas mais complexas como ***listas***. O utilizador de PYTHON tem a possibilidade definir novas estruturas e estabelecer regras e funções para a sua manipulação. Melhor ainda, o utilizador tem a possibilidade de ***importar*** livremente módulos de código aberto nos quais estão definidos objetos úteis para as suas aplicações.

Neste capítulo introduzimos 3 módulos fundamentais do PYTHON: o módulo **numpy** permite-nos utilizar estruturas algébricas na forma de **array**'s multidimensionais, nomeadamente vetores e matrizes, e fazer o seu processamento eficiente; o módulo **matplotlib** permite-nos produzir

gráficos; o módulo **sympy** permite a realização de operações simbólicas. Cada um destes módulos contém inúmeros objetos, estruturas de dados e funções, cuja relevância nos limitámos a aflorar.

A instrução mais básica de qualquer linguagem de programação, a instrução de ***atribuição*** (**=**), merece uma menção especial sobre o seu significado em PYTHON. Considere a sequência de instruções (e o correspondente *output*):

```
import numpy as np
import copy
a=[1,2,3]
b=[1.,2.,3.]
d=copy.copy(a)
x=a
x[0]=10
d[0]=100
y=b*2
c=np.array(b)
z=c*2
print('a=',a,'b=',b,'c=',c,'d=',d,'\n','x=',x,'y=',y,'z=',z)~

>>a= [10, 2, 3] b= [1.0, 2.0, 3.0] c= [ 1.  2.  3.] d= [100, 2, 3]
>>x= [10, 2, 3] y= [1.0, 2.0, 3.0, 1.0, 2.0, 3.0] z= [ 2.  4.  6.]
```

a e **b** são duas listas: uma lista de números inteiros (**a**) e uma lista de "reais", i.e. float (**b**). A instrução **d=copy(a)** cria um objeto **d** idêntico a **a**. A instrução **x=a** faz algo diferente, como se pode ver pelo que se segue: **x** passa a ser um nome alternativo de **a**, de modo que uma modificação de **x** afeta **a**; o mesmo não acontece com **d**. Aqui também se mostra que a operação ***2** afeta de forma diferente uma lista e um array. A operação ***2.** é ilegal quando aplicada a uma lista.

No código anterior utilizaram-se dois tipos de ***parêntesis***: os parêntesis curvos **()** são utilizados para delimitar os argumentos das funções, i.e., os seus parâmetros de entrada; os parêntesis retos **[]** são utilizados para aceder aos elementos de uma lista ou de um array.

3 Ajustamento de dados experimentais por regressão linear

3.1 Problema

Vamos considerar o problema da determinação da constante de Planck h, a partir de dados obtidos numa experiência com uma célula fotoelétrica, que consistem numa série de valores do potencial de paragem V_p, em função do comprimento de onda da luz incidente na célula.

Utilizando a lei de Planck:

$$E = h\nu \tag{3-1}$$

em que E é a energia de um fotão e V a sua frequência, e a lei de conservação da energia pode escrever-se:

$$eV_p = h\nu - W \tag{3-2}$$

em que W é a energia de arranque da célula, considerada constante para cada célula.

O problema consiste, portanto, em: dados $(V_{p,k}, \nu_k)(k = 1,2, \dots, N)$, calcular a melhor aproximação possível (num sentido a definir) para h. A solução do problema fornecerá igualmente um valor para W.

3.2 Formulação matemática

O problema referido é um exemplo de uma família muito extensa de problemas de física (e não só) que se pode reduzir ao problema da determinação dos parâmetros de uma reta (declive e ordenada na origem) que "melhor" se ajusta a uma série de dados.

Formalmente pode escrever-se: Dados $(x_k, y_k)(k = 1,2, \dots, N)$, calcular a e b tais que a recta $y = ax + b$, é a que melhor se ajusta aos dados (num sentido a definir). A solução geralmente utilizada para este problema consiste em escolher os parâmetros da reta que levam à minimização do erro médio quadrático:

$$\overline{e^2} = \frac{1}{N}\sum_{k=1}^{N}[y_k - (ax_k + b)]^2 \tag{3-3}$$

Impondo a condição de minimização, i.e., resolvendo as duas equações:

$$\frac{\partial}{\partial a}\left(\overline{e^2}\right) = \frac{\partial}{\partial b}\left(\overline{e^2}\right) = 0 \tag{3-4}$$

Obtém-se:

$$a = \frac{\sum x_k y_k - \frac{1}{N}\sum x_k \sum y_k}{\sum(x_k^2) - \frac{1}{N}(\sum x_k)^2} \quad (3\text{-}5)$$

$$b = \frac{1}{N}\left(\sum y_k - a\sum x_k\right) \quad (3\text{-}6)$$

em que todos os somatórios são de 1 a N.

3.3 Métodos numéricos

O método consiste unicamente no cálculo dos somatórios necessários e na sua introdução nas fórmulas (3-5) e (3-6). No caso de *N* ser grande ou para valores particulares da série $\{x_i, y_i\}$, pode haver problemas graves de arredondamento no cálculo dos somatórios. Para limitar esses erros deve recorrer-se a cálculos em dupla precisão (`float64`), a precisão standard em PYTHON com processadores x86.

3.4 Exemplos

Exemplo 3-1 Determinação da constante de Planck

A Tabela 3-1 mostra o resultado de uma experiência real (realizada por estudantes da FCUL) com uma célula fotoelétrica. Calcule a constante de Planck, utilizando uma regressão linear.

Tabela 3-1 Efeito fotoeléctrico

Comprimento de onda (nm)	Frequência (10^{12}Hz)	V_P (V)
400	749	1
414	724	0.99
429	699	0.89
444	674	0.79
462	649	0.68
480	624	0.57
500	599	0.47
522	574	0.37
545	549	0.28
571	524	0.17
600	499	0.07

Neste caso vamos trabalhar com séries de valores experimentais e não com uma função analítica. Podemos realizar explicitamente o cálculo proposto com o programa:

```
import numpy as np
import matplotlib.pyplot as plt
nu=1e12*np.array([749,724,699,674,649,624,599,574,549,524,499],dtype=float)
Vp=np.array([1,0.99,0.89,0.79,0.68,0.57,0.47,0.37,0.28,0.17,0.07])
e=1.609e-19 #carga do eletrão
eVp=e*Vp
Sxx=0;Sxy=0
N=len(nu)
for k in range(N):
    Sxy=Sxy+nu[k]*eVp[k]
    Sxx=Sxx+nu[k]*nu[k]
Sx=np.sum(nu)
Sy=np.sum(eVp)
a=(Sxy-Sx*Sy/N)/(Sxx-Sx**2/N)
b=(Sy-a*Sx)/N
h=a
W=-b
plt.close('all')
plt.scatter(nu,Vp)
fit=(h*nu-W)/e;
plt.plot(nu,fit)
plt.text(6.5e14,0.4,'h='+str(h))
plt.xlabel(r'$\nu$')
plt.ylabel(r'$V_p$')
plt.title('Determinação da constante de Planck')
```

`print('h=',h)` Este programa dá origem à Figura 3-1 e à escrita no ecrã da estimativa de h (o valor tabelado para h é de 6.626068×10^{-34} m^2 kg s^{-1}):

```
>>h= 6.32483272727e-34
```

No programa anterior introduzimos um conjunto de novas funcionalidades. O programa utiliza vetores (**nu**, **Vp**, **eVp** e **fit**). Os dois primeiros são convertidos em vetores numpy a partir de listas de números. A operação de multiplicação do escalar 10^{12} na definição dos valores de **nu** só pode ser efetuada depois de definido o vetor, pois tal operação não pode ser realizada com uma lista. O vetor **eVp** é definido numa operação algébrica como o produto (*termo a termo*) do vetor **Vp** pelo escalar **e**.

O cálculo dos somatórios **Sxy** e **Sxx** é feito explicitamente, utilizando um ciclo **for k in range(N):**, depois de **N** ser calculado como o número de elementos do vector **nu**: **N=len(nu)**. Os acumuladores (**Sxy**, **Sxx**) precisam de ser inicializados a 0.

O ciclo **for** é um exemplo de uma ***estrutura de controlo***, um dos elementos chave da programação. Em PYTHON, estas estruturas têm sempre o mesmo formato: uma linha inicial que estabelece a estrutura, terminada pelo caracter “**:**”. As instruções seguintes, que constituem o código a executar, estão ***indentadas*** para a direita. A estrutura termina quando a indentação é revertida, não existindo nenhuma instrução ou caracter especial a indicar o seu término. A instrução **for** delimita um fragmento de código a executar repetidamente para o conjunto de valores de **k** definidos pela função **range(N)**. A função **range(N)** define a sequência de números inteiros [0,1,2,…,N-1] , isto é N números inteiros, com início em 0, e poderia também ser escrita na forma **range(0,N)**, i.e. com indicação explícita do valor inicial do ***iterador***.

No caso dos somatórios **Sx** e **Sy**, o seu cálculo é realizado pela função **numpy.sum**. O programa utiliza duas novas instruções gráficas (do módulo matplotlib.pyplot): **scatter(x,y)** produz um gráficos de pontos com coordenadas (***x,y***) dadas pelos vectores correspondentes; **text(xA,yA,'TEXTO')** imprime um texto na coordenada **(xA,yA)**. No programa mostram-se algumas possibilidades de incluir textos no gráfico, em título e nos eixos.

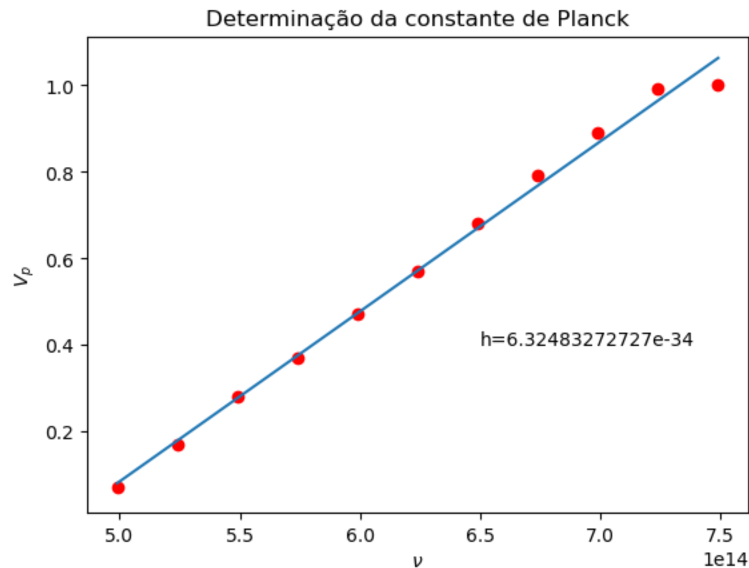

Figura 3-1 Cálculo da constante de Planck por ajuste de dados experimentais

Na verdade, é possível realizar as operações anteriores com um programa mais compacto, utilizando uma função **numpy** que faz diretamente o ajuste linear. A função **polyfit(x,y,n)** faz um ajuste a um polinómio de grau **n**, sendo o resultado um vetor com **n+1** termos contendo os coeficientes do ajuste.

```
import numpy as np
import matplotlib.pyplot as plt
nu=1e12*np.array([749,724,699,674,649,624,599,574,549,524,499])
Vp=np.array([1,0.99,0.89,0.79,0.68,0.57,0.47,0.37,0.28,0.17,0.07])
e=1.609e-19
eVp=e*Vp
p=np.polyfit(nu,eVp,1)
h=p[0]
W=-p[1]
```

Problema 3-1 Ajuste de uma função exponencial

Um condensador de capacidade desconhecida *C* é carregado à tensão V_0 (Figura 3-2, com interruptor na posição A). Seguidamente realiza-se a sua descarga através de uma resistência $R=2k\Omega$ (interruptor na posição B). Durante o processo de descarga é medida a tensão aos terminais do condensador a intervalos de 1s, obtendo-se os dados apresentados na Tabela

3-2. Admitindo que a tensão no condensador segue a equação $V = V_0 e^{-t/(RC)}$: (a) calcule C por mínimos quadrados; (b) represente graficamente os dados experimentais e a curva de ajuste. Sugestão: utilizando logaritmos transforme a lei exponencial numa relação linear e utilize o ajuste linear do exemplo anterior.

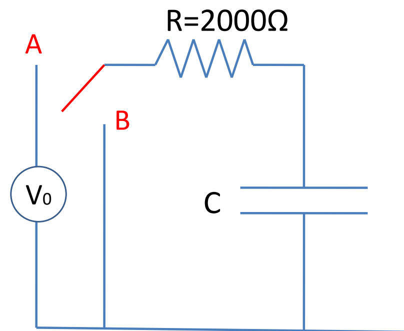

Figura 3-2 Circuito RC

Tabela 3-2 Descarga de um condensador

t(s)	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
V(V)	9.40	7.31	5.15	3.55	2.81	2.04	1.26	0.97	0.74	0.58

3.5 Revisão

Neste capítulo, para além de termos explorado algumas novas funções dos módulos **numpy** e **matplotlib**, introduzimos uma importante estrutura de controlo: o ciclo **for**, i.e., uma estrutura de repetição com um contador, na forma.

```
for k in lista:
      instrução 1
      instrução 2
      ….
      instrução n
```

onde a lista pode ser explícita (`for k in [1,3,5,7]:`) ou definida por um iterador (`for k in range(1,9,2):`). Duas outras estruturas de controlo, com uma forma muito semelhante, são essenciais para a programação, o ciclo **while**, estrutura de repetição controlada por uma condição:

```
k=1
while k<9:
      instrução 1
      ….
      instrução m
      k=k+2
```

e a estrutura **if**, de execução condicional:

```
if x>0:
```

```
      instrução 1
      …´
      Instrução n
elif x==0:
      instruções
else:
      instruções…
```

4 Interpolação

4.1 Interpolação Polinomial.

Os resultados de medições experimentais ou simulações numéricas fornecem, em geral, um conjunto de valores de uma função em pontos discretos de uma variável independente. Esses valores podem ser apresentados na forma de uma tabela para valores discretos de x. O processo de calcular a função para valores intermédios aos valores conhecidos de $f(x)$ é chamado ***interpolação***.

Consideremos a seguinte tabela de valores $(x_i, f(x_i))$:

$$\begin{array}{cccccc} x_0 & x_1 & \cdots & x_k & \cdots & x_N \\ f(x_0) & f(x_1) & \cdots & f(x_k) & \cdots & f(x_N) \end{array}$$

A tabela acima define $N+1$ pontos de uma função desconhecida. Admitindo que a função pode ser escrita como série de Taylor, essa função pode ser aproximada por uma série da forma

$$f(x) \approx a_0 + a_1 x + a_2 x^2 + \cdots + a_N x^N = \sum_{k=0}^{N} a_k x^k \tag{4-1}$$

Representando um polinómio de grau N. Pode ser ainda demonstrado que, se os pontos $x_j\{j = 0, \ldots, N\}$ forem distintos, existe um único polinómio $p(x)$ de grau $\leq N$ tal que $p(x_j) = f(x_j)$ $(0 \leq j \leq N)$. Esse polinómio é o ***polinómio interpolador*** dos $N+1$ pontos. É geralmente possível determinar os coeficientes do polinómio interpolador. Em PYTHON utiliza-se a função `polyfit` para determinar esse polinómio.

Embora um polinómio interpolador de grau N possa ser sempre encontrado, nem sempre é o melhor modelo matemático que se ajusta aos dados experimentais. De facto, dada uma experiência com 100 pontos experimentais de um problema físico ou de engenharia, em geral, o modelo matemático que melhor aproxima os dados experimentais não será (quase de certeza!) um polinómio de grau 99.

Uma estratégia mais razoável para interpolar os dados será ajustar um polinómio de baixo grau, tal como primeira, segunda ou terceira ordem, entre cada dois, três ou quatro pontos, respetivamente. Por exemplo, um polinómio de primeiro grau pode ajustar um conjunto de segmentos de reta entre dois pontos consecutivos da tabela, desenhando uma ***linha quebrada***. Então, pontos intermédios entre os pontos experimentais "tabelados" podem ser determinados pela aproximação linear no respetivo intervalo. O polinómio linear entre x_i e x_{i+1} é dado por:

$$P_1(x) \approx y_j + \left(\frac{y_{j+1} - y_j}{x_{j+1} - x_j}\right)(x - x_j) \tag{4-2}$$

para $x_i \leq x \leq x_{i+1}$. Na equação (4-2) utilizou-se $y_i = f(x_i)$ por conveniência.

Usualmente, utiliza-se uma sequência de polinómios lineares (P_1) ou cúbicos (P_3) em sub-intervalos do intervalo original dos dados para a interpolação. Isso evita as oscilações (não realistas) de um polinómio de grau superior que passasse por todos os pontos experimentais. Essas oscilações que claramente não são o melhor ajuste aos dados derivam dos valores elevados das derivadas do polinómio.

Vejamos agora um exemplo de um problema de interpolação utilizando funções PYTHON. O comando **yfit=polyfit(x,y,N)** permite calcular o polinómio de grau N que ajusta $N+1$ pontos, ou se as séries **x,y** tiverem mais de N+1 pontos o ajuste por mínimos quadrados. O comando **polyval(p,xi)** permite avaliar o valor do polinómio nos elementos de um vetor **xi** definido pelos coeficientes do polinómio fornecidos pelo vector **p**. Já o comando **interp** é utilizado para ajustar uma sequência de polinómios lineares para a totalidade do intervalo especificado. O comando **spline** do módulo **scipy.interpolate**, faz o ajuste por troços com polinómios de ordem variável (no caso **order=1** será idêntico a interp).

Como exemplo destes comandos e do problema da interpolação entre diferentes graus de polinómios, vamos utilizar a função de *Runge*. A função de *Runge* define-se como:

$$y(x) \approx \frac{1}{1+x^2} \tag{4-3}$$

e vamos avaliá-la no intervalo $[-5,5]$. Este tipo de funções são normalmente referidas como maus exemplos da interpolação polinomial, uma vez que são suaves mas o polinómio interpolador afasta-se significativamente da função na extremidade do intervalo. O que se quer mostrar com este exemplo é que, mesmo para uma função muito suave, o polinómio interpolador pode não apresentar um bom ajuste. Se os dados a ser interpolados provêm de uma função com vários pontos de inflexão, a aproximação polinomial poderá ainda ser de pior qualidade.

A interpolação com recurso a polinómios lineares entre os pontos experimentais é muitas vezes perfeitamente adequada. O *script* PYTHON que se apresenta permite calcular um polinómio interpolador que passa por 11 pontos experimentais da equação (4-3) no intervalo acima descrito, interpolando-o para uma malha mais fina, com 101 pontos. A seguir a função **interp** é utilizada para interpolação linear entre cada conjunto de dois pontos consecutivos do intervalo e a função **spline**, para uma interpolação por ***splines cúbicas*** entre cada 3 pontos, sempre para a malha fina.

```
"""
Programa exemplo para executar interpolação polinomial. Função de Runge
      f(x)=1/(1+x^2)
interpolada em N+1 pontos com um polinómio de
grau N; depois compara com a interpolação linear
em N intervalos;
teste com N=10 no intervalo [-5, 5].
"""
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import spline
x=np.linspace(-5,5,11)         # 11 pontos
N=len(x)-1 # escolha de grau 10 para polinómio
y=1./(1+x**2)       # função de Runge
# Ajuste polinomial de grau N
p=np.polyfit(x,y,N)   #p coeficientes do polinómio
xplot=np.linspace(-5,5,101)    # define grelha mais fina c/ 101 pontos
f=np.polyval(p,xplot) # avalia f nos pontos xplot
#interpolação linear
ylinear=np.interp(xplot,x,y);
ycubic=spline(x,y,xplot,order=3)
plt.close('all')                # clear all figures
plt.subplot(3,1,1); plt.scatter(x,y,color='red');plt.plot(xplot,f,'blue')
```

```
plt.title('Interpolação polinomial')
plt.subplot(3,1,2);
plt.scatter(x,y,color='red');plt.plot(xplot,ylinear,'blue')
plt.title('Interpolação linear')
plt.subplot(3,1,3);
plt.scatter(x,y,color='red');plt.plot(xplot,ycubic,color='blue')
plt.title('Interpolação por spline cúbica'
```

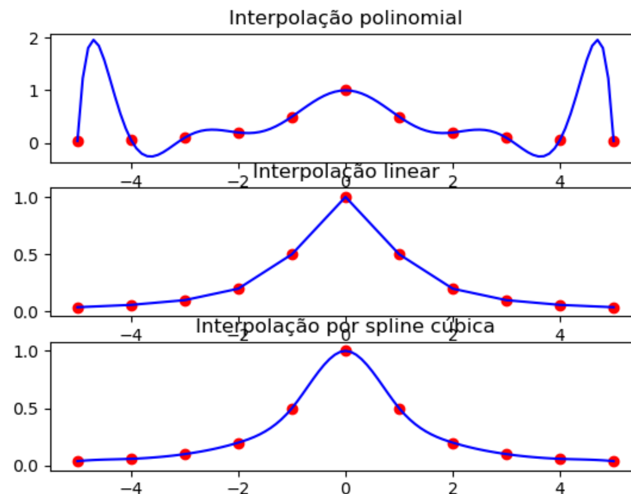

Figura 4-1 A função de Runge interpolada de várias maneiras. Em cima a) uma interpolação polinomial de grau 11, passando por todos os pontos do intervalo escolhido $x = [-5:1:5]$. Ao centro uma interpolação linear por troços utilizando a função interp; e em baixo c) a função spline, com interpolador cúbico. Repare como a interpolação polinomial de ordem 11 é inadequada, especialmente nos extremos do intervalo escolhido.

A seguir apresenta-se um exemplo em que se simula um conjunto de 21 pontos experimentais a partir de um polinómio de grau 3, por mínimos quadrados. A simulação é feita adicionando ruído aleatório à amostragem do polinómio nos 21 pontos igualmente espaçados. Depois é feito um ajuste polinomial a um novo polinómio de grau 3, que é de seguida comparado graficamente com os resultados experimentais assim como com o polinómio original.

```
"""
Script para gerar dados a partir de um polinómio de
grau 3 c/ ruido sobreposto
Ajusta um novo polinómio de grau 3 aos dados gerados
Comparação gráfica
"""
import numpy as np
import matplotlib.pyplot as plt
p3=np.array([0.74,0.97,1.1,0.86]);  # coef. do polinomio gerador
x=np.arange(-1,1.1,0.1);            # dominio a considerar
y=np.polyval(p3,x)                  # avalia o polinomio no intervalo
nc=len(y)                           # dimensao da amostragem
```

```
noise=np.random.randn(nc);              # gera ruido aleatório
noise=noise/np.linalg.norm(noise);      # normaliza o ruido
ey=y+noise;                             # adiciona ruido
ec=np.polyfit(x,ey,3)                   # ajuste polinomial (grau 3)
plt.close('all')
plt.figure()
plt.plot(x,y,'b--');
plt.plot(x,ey,'r+');
fy=np.polyval(ec,x);                    # avalia o ajuste
plt.plot(x,fy,'g');
```

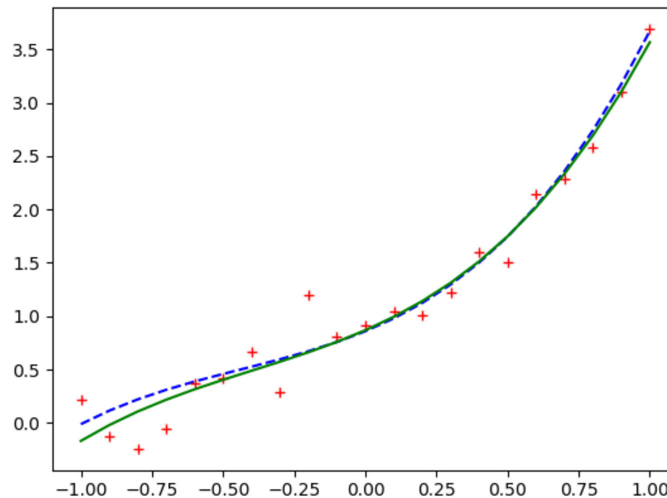

Figura 4-2. Resultado gráfico do Script PYTHON acima apresentado com a comparação gráfica de dados experimentais simulados a partir de um polinómio de grau 3 (linha tracejada a azul), com um ajuste a um polinómio de grau 3 (linha contínua a verde).

4.2 Interpolação por funções spline.

O termo *spline* refere-se a um dispositivo flexível que é usado para desenhar uma curva suave através de um conjunto de pontos num gráfico. Do ponto de vista matemático, a interpolação *spline* consiste em encontrar um conjunto de polinómios que passe através dos pontos experimentais dados por $f(x_i), \{i = 0, 1, 2, 3, \dots, N\}$. Embora a teoria das *splines* seja bastante extensa, vamos aqui restringir-nos a *splines* cúbicos, uma vez que estes são os mais utilizados em aplicações de engenharia.

Uma função *spline* de grau n pode ser definida no intervalo [a, b] com subintervalos definidos pelos N+1 pontos

$$a = x_0 < x_1 < \cdots < x_N = b \tag{4-4}$$

A função *spline s(x)* tem as seguintes propriedades:

(a) Em cada subintervalo $[x_i, x_{i+1}], s(x)$ é um polinómio de grau n;

(b) $s(x)$ e as primeiras $(n-1)$ derivadas são contínuas em $[a, b]$.

A função *spline* é construída por pedaços de polinómio unidos nos extremos de cada subintervalo. Na interpolação por *spline* cúbica, os polinómios de grau 3 são unidos de maneira a que a função resultante, assim como a primeira e segunda derivadas, sejam contínuas em cada ponto do intervalo. Portanto, em cada subintervalo $[x_i, x_{i+1}]$, a função spline é dada por:

$$s_i(x) = a_{j0} + a_{i1}(x - x_i) + a_{i2}(x - x_i)^2 + a_{i3}(x - x_i)^3 \quad (4\text{-}5)$$

para $1 \leq i \leq N$. Como existem $N + 1$ pontos, uma equação cúbica com 4 parâmetros ajusta cada um dos N intervalos. Portanto $4N$ parâmetros devem ser determinados no total.

Uma condição que se impõe aos polinómios da equação (4-5) é de que interpolem os pontos $f(x_i), \{i = 0, 1, 2, \dots, N\}$ isto é que passem por esses pontos, traduzindo-se pois em $N + 1$ equações. Os polinómios de grau 3 e a primeira e segunda derivadas respetivas devem ser contínuas em cada ponto interior. Essas condições equivalem a $3(N + 1)$ equações, para um total de $4N - 2$ equações. As duas outras condições que restam podem definir-se pelos valores da primeira e segunda derivada nos extremos do intervalo. As *splines* chamadas *naturais cúbicas* têm como condições fronteira $s''(x_0) = 0$ e $s''(x_N) = 0$. As $4N$ equações podem ser resolvidas dando os coeficientes da função *spline* (4-5) nos N intervalos.

Na Figura 4-1c mostrou-se o resultado de uma interpolação por splines da função de Runge.

4.3 Revisão

Dicionário e posição.

5 Raízes de equações não lineares

5.1 Equilíbrio térmico de um painel solar

Considere o seguinte problema de Termodinâmica: determinar a temperatura de equilíbrio de uma placa negra exposta ao Sol e ao ar, considerando dados a temperatura do ar, $T_0 = 273K$, e a irradiância solar ($E_S = 500Wm^{-2}$).

Fisicamente, o problema consiste em resolver a equação de balanço energético da placa (primeiro princípio da Termodinâmica ou Princípio da conservação da energia) que se pode escrever:

$$|\text{Fluxo de calor recebido}| - |\text{Fluxo de calor perdido}| = 0 \quad (5\text{-}1)$$

Nas condições referidas, a placa perde calor sob duas formas: por radiação e por condução/convecção. O calor emitido pela placa sob a forma de radiação (medido em Wm^{-2}) é dado pela lei de Stefan-Boltzmann:

$$\dot{Q}_{rad} = \varepsilon\sigma T^4 \quad (5\text{-}2)$$

em que $\sigma = 5.67 \times 10^{-8}Wm^{-2}K^{-4}$ é a constante de Stefan-Boltzmann, T a temperatura absoluta da placa (Kelvin) e ε é a emissividade (=1 no caso do corpo negro).

O calor perdido diretamente para o ar sob a forma de condução/convecção pode ser dado, em primeira aproximação pela expressão:

$$\dot{Q}_{conv} = \alpha(T - T_0) \quad (5\text{-}3)$$

em que α é uma constante numérica. Considerar-se-á $\alpha = 0.4Wm^{-2}K^{-1}$. Nestas condições, a condição de balanço térmico da placa escreve-se:

$$E_S - \dot{Q}_{rad} - \dot{Q}_{conv} = 0 \Rightarrow E_S - \sigma T^4 - \alpha(T - T_0) = 0 \quad (5\text{-}4)$$

o que constitui uma equação não linear para T .

Diversos problemas de Física envolvem a determinação de raízes de equações não lineares. Neste exemplo vamos considerar unicamente o caso do problema da determinação de raízes reais. Em muitos casos, como no exemplo considerado, a função tem mais do que uma raiz real. Os métodos introduzidos neste problema permitem calcular essas raízes uma a uma.

5.2 Formulação matemática:

Determinar *T* que satisfaz a equação:

$$500 - 5.67 \times 10^{-8} \times T^4 - 0.4(T - 273) = 0 \quad (5\text{-}5)$$

O problema proposto pertence a uma classe de problemas que passam pela determinação de raízes de uma equação não linear. O problema pode ser escrito na forma geral:

Calcular $x \in \Re: f(x) = 0$.

No caso de existir mais do que uma raiz, será necessário considerar o processo de seleção das diferentes raízes.

No caso concreto da equação (5-5), dado que se trata de uma equação do quarto grau, é sabido que ela tem 4 raízes, sendo que só uma delas é relevante para o problema físico posto. Acrescenta-se que neste caso existem duas raízes reais

$$\mathrm{T} = -338.5214924, +304.4922920$$

e duas raízes complexas

$$\mathrm{T} = -0.01460017 \pm 322.4060715\ \mathrm{i}$$

sendo claro que a raiz relevante é a única real e positiva (dado tratar-se de uma temperatura absoluta). No entanto, deve notar-se que em alguns problemas poderão existir várias soluções com sentido físico.

5.3 Métodos numéricos

Salvo casos particulares, as equações não lineares não podem ser resolvidas analiticamente, i.e., não podem ser resolvidas explicitamente em ordem a x. Todos os métodos disponíveis são métodos iterativos, que partem de uma estimativa inicial do valor da raiz e procedem por aproximações sucessivas. Vamos considerar unicamente dois métodos, entre os muitos disponíveis. Uma descrição pormenorizada destes métodos pode ser estudada na literatura de análise numérica.

5.3.1 Método da bissecção

O método da bissecção baseia-se diretamente no teorema do valor médio. Dado o intervalo $[a, b]$ se $sinal(f(a)) \neq sinal(f(b))$ então existe $x \in [a, b]: f(x) = 0$. O método consiste na divisão sucessiva do intervalo original em duas partes iguais seguida da escolha do sub-intervalo em que ocorre a mudança de sinal.

O método da bissecção converge sempre (para uma raiz ou para uma singularidade). A sua convergência é, no entanto, lenta (linear).

Exemplo 5-1

Calcule a temperatura de equilíbrio do painel solar descrito em 5.1, resolvendo a equação (5-5), pelo método da bissecção.

O problema novo mais interessante que se surge nesta aplicação é o problema do controle da convergência de um método iterativo. Existem diferentes metodologias para fazer esse controle. Em qualquer caso, elas envolvem o estabelecimento de uma medida numérica do grau de convergência atingido e a execução da iteração *enquanto* essa medida não atingir o valor pretendido. A

implementação desse critério faz-se facilmente com recurso a uma ***Estrutura de Controlo*** designada como Iteração Condicional (sem contador) com teste inicial ou Estrutura **while**.

```
while (condição):
    conjunto de instruções
```

Resta naturalmente estabelecer qual a condição de paragem. As condições mais comuns baseiam-se no valor do ERRO ABSOLUTO (avaliado como o valor absoluto da correção introduzida pela última iteração) do ERRO RELATIVO (a razão entre o erro absoluto e o valor da solução, admitindo que este não é nulo) e do NÚMERO DE ITERAÇÕES. Este último critério destina-se unicamente a impedir a continuação indefinida da iteração em caso de não convergência e utiliza-se em simultâneo com um dos anteriores.

Uma possível solução do problema é apresentada no *script* painel_solar.

```
# painel_solar.py
# determinação da temperatura de equilíbrio de painel solar

def bissec(y,xLow,xHigh,maxErro,maxIter):
    nIter=0;
    if y(xLow)*y(xHigh)>0:
        print('Erro na escolha do intervalo inicial')
        erroAbs=-1
        raiz=float(‘nan’)
    else:
        erroAbs=xHigh-xLow
        while erroAbs>maxErro and nIter<maxIter:
            nIter=nIter+1
            raiz=0.5*(xHigh+xLow)
            if y(xLow)*y(raiz)<0:
                xHigh=raiz
            elif y(xLow)*y(raiz)>0:
                xLow=raiz
            else:
                xHigh=raiz
                xLow=raiz
            erroAbs=xHigh-xLow;
            plt.scatter(raiz,y(raiz),marker='.',color='b')
            plt.text(raiz,y(raiz),str(nIter))
    return raiz,erroAbs,nIter

def y(T):
    f=500-5.67e-8*T**4-0.4*(T-273)
    return f

import matplotlib.pyplot as plt
plt.close('all')
TLow=273; THigh=340; #intervalo inicial
maxIter=100; #numero maximo de iteracoes
maxErro=0.001; #erro absoluto máximo
Teq,erroAbs,nIter=bissec(y,TLow,THigh,maxErro,maxIter)
plt.scatter(Teq,y(Teq),marker='o',color='r')
plt.text(Teq-10,y(Teq),r'$T_{e}=%10.5f \pm %8.6f$' % (Teq,erroAbs))
plt.grid() #marca grelha na figura
```

A solução anterior apresenta algumas novidades interessantes na utilização do PYTHON. O *script* começa com a definição de uma ***função*** (**bissec**). Em exemplos anteriores utilizámos funções definidas em módulos importados, neste caso a função é construída no nosso código (e poderia ser importada futuramente). A definição da função utiliza a ***estrutura de controlo*** **def**:

```
def nome_da_função(arg1,arg2,…,argn):
       instruções…
       return out1,out2,…,outm
```

Nesse *script* define-se ainda a função **y(T)** O *script* painel_solar calcula a raiz da equação "**y(T)=0**", pelo método da bissecção, começando no intervalo [**xLow,xHigh**], e utilizando um número de iterações não superior a **maxIter** até que o erro absoluto seja inferior a **maxErro**. Esta função devolve o valor da **raiz**, do **erroAbs** estimado e do número de iterações efetivamente realizadas **nIter**. Notas importantes:

(a) As variáveis utilizadas por cada função são locais, podendo ter nomes diferentes que as variáveis correspondentes noutros subprogramas;

(b) A função bissec devolve 3 elementos;

(c) Utiliza-se o "número" nan (not a number) para indicar uma condição de erro;

(d) Utilizam-se algumas instruções novas de output gráfico.

A execução do script painel_solar dá origem à Figura 5-1 e ao output na consola:

```
nIter= 1 raiz= 306.5
nIter= 2 raiz= 289.75
nIter= 3 raiz= 298.125
nIter= 4 raiz= 302.3125
nIter= 5 raiz= 304.40625
nIter= 6 raiz= 305.453125
nIter= 7 raiz= 304.9296875
nIter= 8 raiz= 304.66796875
nIter= 9 raiz= 304.537109375
nIter= 10 raiz= 304.4716796875
nIter= 11 raiz= 304.50439453125
nIter= 12 raiz= 304.488037109375
nIter= 13 raiz= 304.4962158203125
nIter= 14 raiz= 304.49212646484375
nIter= 15 raiz= 304.4941711425781
nIter= 16 raiz= 304.49314880371094
nIter= 17 raiz= 304.49263763427734
```

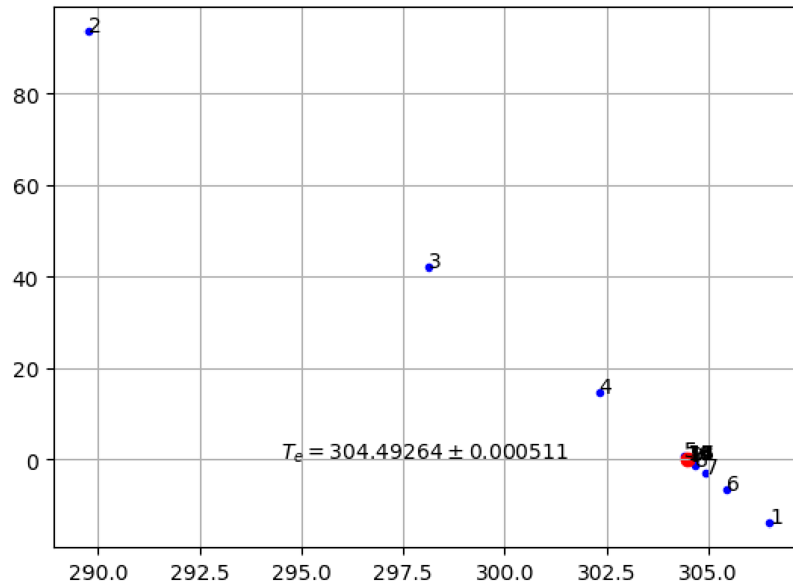

Figura 5-1 Temperatura de equilíbrio de um painel solar (solução final a vermelho).

O conjunto completo de soluções (2 reais, 2 complexas) poderia obter-se com o script seguinte (com 16 algarismos significativos). Note-se que a solução encontrada faz parte do conjunto de soluções e apresenta um erro inferior a 0.001K, como pretendido.

```
from sympy import Symbol,solve
T=Symbol('T')
Teq=solve(500-5.67e-8*T**4-0.4*(T-273),T)
print(Teq)

>> [-338.521492377835, 304.492292028057, 17.0146001748888 -
      322.406071549869*I, 17.0146001748888 + 322.406071549869*I]
```

Salienta-se que a função bissec anteriormente apresentada é absolutamente genérica, podendo ser utilizada para outras funções: bastaria rescrever a função **y(T)**.

5.3.2 Método de Newton-Raphson

O método de Newton-Raphson baseia-se diretamente no desenvolvimento em série de Taylor da função em torno de um ponto nas proximidades de uma raiz:

$$f(x_{j+1}) = f(x_j) + f'(x_j)(x_{j+1} - x_j) + \cdots \tag{5-6}$$

Impondo a condição $f(x_{j+1}) = 0$, e desprezando os termos de ordem superior à primeira, obtém-se:

$$x_{j+1} = x_j - \frac{f(x_j)}{f'(x_j)} \tag{5-7}$$

A expressão anterior pode então ser utilizada de forma recursiva para, a partir de uma estimativa inicial do valor da raiz, obter aproximações sucessivamente melhores do seu valor.

Contrariamente ao método da bissecção, não existe garantia de convergência do método de Newton-Raphson. No caso em que há convergência esta é bastante rápida (quadrática).

5.3.3 Solução genérica

Como acontece frequentemente, o PYTHON dispõe de uma função genérica para calcular raízes de equações, a função **fsolve**. Um exemplo da sua utilização para o problema do painel é dado pelo programa:

```
from scipy.optimize import fsolve
Teq2=fsolve(y,300) #300 é uma estimativa à priori da solução
print(Teq2)

>>[ 304.49229203]
```

Esta solução é obtida numericamente, recuperando só uma raiz, enquanto a solução de sympy.solve utiliza álgebra simbólica e recupera todas as raízes.

Problema 5-1

Calcule a temperatura de equilíbrio do painel solar descrito em 5.1, resolvendo a equação (5-5), pelo método de Newton-Raphson.

6 Processamento de dados multidimensionais

Neste capítulo vamos estudar alguns problemas simples que requerem a utilização de dados em duas ou mais dimensões. Em geral, tais dados têm origem em observações de campos distribuídos, por exemplo por redes de estações de observação ou por sistemas de deteção remota, mas também podem ser produzidos por modelos numéricos. Por simplicidade vamos focar-nos em dados bidimensionais, representados por tabelas de números, i.e. por matrizes. Em geral, estes dados precisam de ser lidos de um ficheiro externo, e esta é uma boa oportunidade para introduzir métodos de leitura e escrita.

6.1 Funções simples de leitura e escrita de tabelas de dados

Dados multidimensionais podem ser transferidos por muitas formas. Aqui vamos só explorar a manipulação de ficheiros produzidos por folhas de cálculo (xlsx). Apesar de parecer contra-intuitivo, a manipulação de folhas de cálculo é bastante simples, desde que estejamos familiarizados com a sua estrutura, pois esta é absolutamente pré-determinada: cada *workboot* é um conjunto de *worksheets*, que por sua vez são matrizes bidimensionais compostas por *células*.

Como exemplo, vamos considerar as distribuições de várias variáveis numa simulação em alta resolução da Península Ibérica com o modelo WRF. Alguns dados desta simulação foram escritos num ficheiro xlsx (‘meteo_model.xlsx’). A Figura 6-1 mostra uma das suas worksheets (‘Info’), onde se inscreveram informações genéricas sobre as variáveis representadas. Na Figura 6-2 mostra-se um dos campos bidimensionais, a pressão no nível 4 do modelo, cerca de 470 m acima da superfície.

	A	B	C	D
1	Variable	units	level	height
2	Longitude	deg		
3	Latutide	deg		
4	Pressure	hPa	4	470m
5	Temperature	K	4	470m
6	qv	g/kg	4	470m
7	U	m/s	4	470m
8	V	m/s	4	470m
9	W	m/s	4	470m
10	Rain	mm	0	
11	Terrain	m	0	
12				
13	year	2015		
14	month	6		
15	day	9		
16	hour	22		
17	min	0		
18	sec	0		
19				
20				

Figura 6-1 Print-screen do ficheiro ‘meteo_model.xlsx’. Informação na worksheet ‘Info’.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	983.3611	983.3722	983.3919	983.3975	983.4002	983.4039	983.4056	983.4034	983.4015	983.3984	983.3859	983.3911	983.3892	983.387
2	983.3563	983.3577	983.3566	983.3467	983.3462	983.363	983.343	983.3503	983.3434	983.3438	983.3261	983.3241	983.3248	983.3314
3	983.3382	983.3393	983.3293	983.3243	983.3352	983.3422	983.3238	983.3148	983.3176	983.2998	983.3027	983.2913	983.2891	983.3028
4	983.3339	983.3243	983.3216	983.3129	983.3193	983.3053	983.2951	983.2959	983.2946	983.2881	983.2691	983.2651	983.2634	983.2669
5	983.3125	983.3077	983.308	983.3094	983.3062	983.299	983.2887	983.2861	983.2705	983.2581	983.25	983.2441	983.2376	983.253
6	983.3126	983.2994	983.2999	983.2986	983.2935	983.2877	983.2829	983.2646	983.2503	983.2424	983.2319	983.2291	983.2323	983.2238
7	983.3043	983.2934	983.2885	983.2878	983.2826	983.2849	983.2666	983.2473	983.2359	983.2135	983.2175	983.2109	983.1991	983.215
8	983.2956	983.2914	983.2781	983.2823	983.2724	983.2632	983.2432	983.2366	983.2233	983.2133	983.2063	983.2013	983.1966	983.1903
9	983.2908	983.2779	983.2723	983.267	983.2547	983.2494	983.2341	983.2208	983.2082	983.1994	983.1905	983.1888	983.1818	983.1763
10	983.258	983.2707	983.2632	983.2565	983.2485	983.2455	983.2206	983.207	983.1964	983.2005	983.1808	983.1844	983.1723	983.176
11	983.2521	983.2455	983.2363	983.2552	983.2373	983.2188	983.2062	983.1984	983.18	983.1609	983.1664	983.1622	983.1622	983.1753
12	983.2419	983.232	983.2319	983.2273	983.2034	983.1893	983.172	983.173	983.1633	983.1527	983.1539	983.1605	983.1766	983.1805
13	983.2155	983.2166	983.2127	983.207	983.1923	983.1758	983.1643	983.1574	983.1614	983.1539	983.1582	983.1673	983.1784	983.1715
14	983.2074	983.2105	983.1917	983.1835	983.1634	983.1603	983.1535	983.1533	983.1605	983.1616	983.1576	983.1696	983.1602	983.1663
15	983.178	983.1814	983.1648	983.1577	983.1692	983.1622	983.153	983.1552	983.1559	983.1465	983.1567	983.1573	983.1541	983.1419
16	983.156	983.1577	983.1484	983.147	983.1426	983.1406	983.1354	983.1405	983.1374	983.1394	983.1313	983.1348	983.1234	983.1285
17	983.1318	983.128	983.1311	983.1236	983.1341	983.1364	983.1356	983.1499	983.131	983.1355	983.1324	983.1298	983.1213	983.1083
18	983.1139	983.0936	983.0998	983.1045	983.1097	983.1159	983.1256	983.1263	983.1166	983.1073	983.1112	983.1159	983.1061	983.1124
19	983.0876	983.0741	983.075	983.0822	983.08	983.1007	983.1066	983.1069	983.1038	983.0959	983.0986	983.0979	983.0921	983.0901
20	983.0509	983.0523	983.0635	983.0704	983.0642	983.0853	983.1021	983.1005	983.0913	983.0726	983.0712	983.0645	983.0842	983.0795
21	983.036	983.0322	983.0309	983.0497	983.053	983.0831	983.0764	983.067	983.0596	983.0477	983.0448	983.0452	983.0553	983.0709
22	983.0019	983.008	983.0197	983.0344	983.0492	983.0598	983.0617	983.0516	983.026	983.052	983.042	983.0405	983.0542	983.0644
23	983.0015	982.9941	983.0112	983.0258	983.0363	983.032	983.045	983.0448	983.0405	983.0402	983.043	983.0408	983.0491	983.0401
24	982.974	982.9824	983.0013	983.023	983.0329	983.0461	983.0403	983.0549	983.0516	983.035	983.0362	983.033	983.0247	983.0237
25	982.9645	982.9855	983.0118	983.0217	983.0352	983.0441	983.0388	983.0431	983.0459	983.0325	983.0159	983.0101	983.0154	982.9993
26	982.9702	982.9834	983.0054	983.0209	983.0316	983.0416	983.0395	983.0362	983.0258	983.0191	983.0077	983.008	983.0088	982.9975
27	982.9635	982.9921	983.0082	983.0252	983.0296	983.023	983.0256	983.0264	983.0169	983.0072	982.9973	982.9959	982.99	982.9877
28	982.9691	982.9794	983.0099	983.0206	983.0237	983.0191	983.0129	983.0091	982.9998	983.0023	982.9863	982.9772	982.9799	982.972
29	982.9709	982.9906	982.9913	983.0098	983.0103	983.0084	982.9969	982.9915	982.9795	982.9696	982.9681	982.9688	982.9606	982.9514
30	982.9784	982.987	982.9991	983	983.0031	982.9867	982.9805	982.9716	982.9711	982.955	982.938	982.945	982.9509	982.9468
31	982.9563	982.9747	982.9846	982.9902	982.9801	982.9698	982.963	982.9569	982.9533	982.9474	982.9296	982.9214	982.9298	982.906
32	982.9608	982.9675	982.9737	982.9725	982.9679	982.9662	982.9599	982.925	982.945	982.9286	982.9163	982.908	982.9002	982.8939
33	982.9563	982.956	982.9626	982.9588	982.956	982.9459	982.9231	982.9183	982.9052	982.9002	982.9028	982.8883	982.8856	982.8759
34	982.9398	982.949	982.9583	982.9411	982.9288	982.9176	982.9064	982.8878	982.8866	982.8896	982.8646	982.8802	982.8811	982.8614
35	982.9385	982.937	982.9332	982.9345	982.9148	982.9027	982.8788	982.8683	982.8745	982.8628	982.8552	982.8392	982.8434	982.8505
36	982.9227	982.9238	982.9292	982.8992	982.8794	982.8773	982.8748	982.8361	982.8479	982.8458	982.8495	982.8378	982.8312	982.8247

Figura 6-2 Print-screen do ficheiro ‘meteo_model.xlsx’. Worksheet ‘Pressure’. As variáveis encontram-se uma por worksheet.

O facto de os dados em xlsx estarem organizados de forma rígida, facilita a sua leitura pelo python. Neste caso cada worksheet contém um campo bidimensional, nos pontos de longitude e latitude indicados na worksheet respetiva, sendo muito explícita a organização dos dados. O *script* seguinte exemplifica a utilização dos dados no ficheiro ‘meteo_model.xlsx’.

```
#ReadXLS.py
import numpy as np
import openpyxl as pyxl
import matplotlib.pyplot as plt

dados=['meteo_model.xlsx']
sheets=['Longitude','Latitude','Pressure','Temperature','U','V','W','qv','R
       ain','Terrain']
nvar=len(sheets)
wb=pyxl.load_workbook(dados[0]) #abre o workbook
wsI=wb['Info'] #abre a worksheet Info
ano=wsI['B13'].value
mes=wsI['B14'].value
dia=wsI['B15'].value
```

```
hora=wsI['B16'].value
minuto=wsI['B17'].value
segundo=wsI['B18'].value
timeS=str(ano)+'-'+str(mes)+'-'+str(dia)+'
      '+str(hora)+':'+str(minuto)+':'+str(segundo)

ivar=0
for variable in sheets: #lê os dados e coloca-os no array 3D, var
    ws=wb[variable]
    if variable=='Longitude':
        rows=ws.max_row #identifica dimensão da worksheet
        cols=ws.max_column
        var=np.zeros((rows,cols,nvar))

    for r in range(rows):
        for c in range(cols):
            var[r,c,ivar]=ws.cell(row=r+1,column=c+1).value
    ivar=ivar+1

lon=var[:,:,0];lat=var[:,:,1]

plt.close('all') #Fecha as figuras pre-existentes

plt.figure() #nova figura
Qmap=plt.contourf(lon,lat,var[:,:,7],cmap='jet') #campo qv
plt.colorbar(Qmap,label=r'$q_v (g/kg)$')
plt.contour(lon,lat,var[:,:,9],colors='black',levels=[10]) #linha de costa
plt.title('qv'+timeS+'@470m')
plt.quiver(lon[::10,::10],lat[::10,::10],var[::10,::10,4],var[::10,::10,5])
      #vento (1 em cada 10*10 pontos)

plt.figure() #zoom
Rmap=plt.contourf(lon,lat,var[:,:,8],cmap='cool') #chuva
plt.contour(lon,lat,var[:,:,9],colors='red',levels=[10]) #linha de costa
plt.title('zoom of wind field and rain')
plt.colorbar(Rmap,label='mm')
passo=5
plt.barbs(lon[::passo,::passo],lat[::passo,::passo],var[::passo,::passo,4],
      var[::passo,::passo,5]) #vento

plt.xlim(-4,0);plt.ylim(36,38) #zoom
```

A acesso ao ficheiro xlsx foi feito utilizando o módulo `openpyxl`. No código acede-se às células de cada worksheet de duas formas: como um ***dicionário*** `ws['A1'].value` acede à célula da linha A, coluna 1; `ws.cell[row=1,column=1].value` faz exatamente o mesmo. O módulo `openpyxl` também permite escrever em células, ou até criar um novo workbook.

6.2 Gráficos bidimensionais

No script ReadXLS, os dados bidimensionais lidos são representados de diversas formas. A sequência

```
Qmap=plt.contourf(lon,lat,var[:,:,7],cmap='jet') #campo qv
plt.colorbar(Qmap,label=r'$q_v (g/kg)$')
```

```
plt.contour(lon,lat,var[:,:,9],colors='black',levels=[10]) #linha de costa
plt.title('qv'+timeS+'@470m')
plt.quiver(lon[::10,::10],lat[::10,::10],var[::10,::10,4],var[::10,::10,5])
      #vento (1 em cada 10*10 pontos)
```

calcula o gráfico de isolinhas da variável q_v (humidade específica) com o código de cores 'jet', imprime a legenda correspondente (**colorbar**), sobrepõe-lhe a isolinha da altitude dos 10m (uma representação grosseira da linha de costa) e o campo vetorial da velocidade horizontal do vento, (**quiver**) dando origem à Figura 6-3. Notar a forma como fui introduzido o *subscript* na legenda.

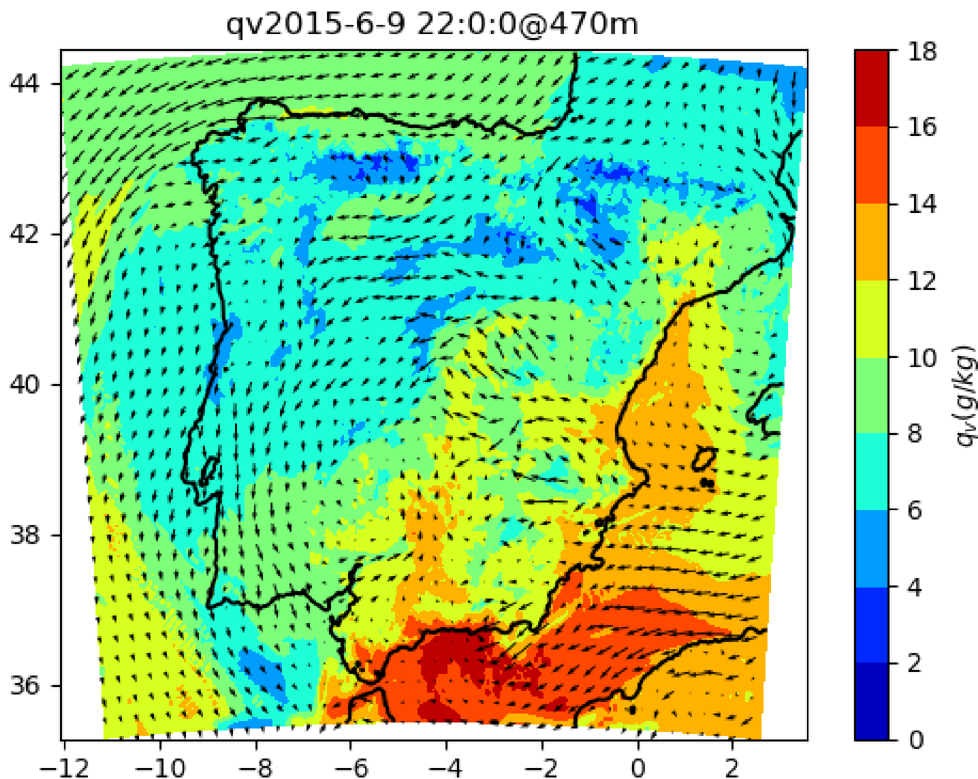

Figura 6-3 Distribuição da humidade específica e vento aos 470m. Representado o vento em $1/(10 \times 10)$ pontos.

O código seguinte mostra o campo de precipitação (Rain), mas só numa região no SE da península, onde teve lugar uma tempestade com *flash flood*. Neste caso, utiliza-se uma representação alternativa para o vento, na convenção meteorológica. O resultado é apresentado na Figura 6-4. Em ambas as figuras na representação do campo vetorial selecionou-se um subconjunto de pontos regularmente espaçados, para evitar empastelamento da figura. A instrução **variavel[::n]** seleciona 1 em cada n elementos do vetor. Esta forma de seleção escreve-se, na forma geral **variável[inicio:fim:passo]**, onde, por defeito (i.e. se não for indicado, como é parcialmente o caso do exemplo), **inicio** o primeiro elemento (0), **fim** o último elemento e **passo**=1.

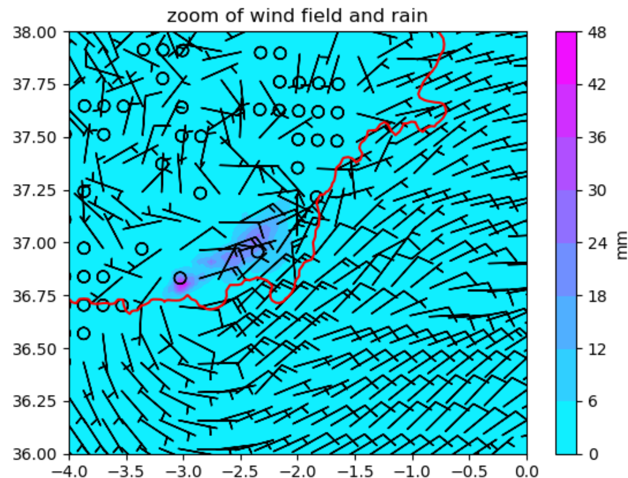

Figura 6-4 Distribuição da chuva e vento (aos 470m) na vizinhança de Adra. Círculo indica calma (vento muito fraco). Representado o vento em $1/(5 \times 5)$ pontos. Linha de costa (aproximada) a vermelho.

6.3 Cartografia

No caso de informação com base geográfica, é interessante recorrer a projeções padronizadas na cartografia, para a sua representação gráfica. O python dispõe de um conjunto alargado de funções para esse efeito, incluídas no módulo ***basemap***, que precisa de ser instalado (antes de importado), pois ele não faz parte do pacote ***anaconda***. A definição do sistema de projeção deve ser realizada antes de produzidos os mapas, obrigando à escolha de vários parâmetros numéricos, dependendo da projeção selecionada.

Assim, podemos refazer o mapa representado na Figura 6-3, utilizando a projeção UTM (Transverse Mercator), frequente utilizada na cartografia nacional. O código seguinte, que admite que as matrizes lon,lat,qv foram já lidas ou calculadas, produz a Figura 6-5. As keyword **`height`** e **`width`** determinam a extensão do mapa em metros, resolution='f' indica que a linha de costa será introduzida com resolução fine, **`lon_0, lat_0`** localizam o centro do mapa. A projeção dos dados experimentais é realizada com o comando **`x,y=mymap(lon,lat)`** que converte a malha do modelo (em graus) na malha (em m) da projeção. As outras instruções são auto-explicativas. Notar que a linha de costa é fornecida pelo módulo basemap.

```
plt.close('all')
plt.figure()
mymap = Basemap(height=1.2e6,width=1.2e6,resolution='f',area_thresh=0.1,\
                projection='tmerc',lon_0=-5,lat_0=40)
mymap.drawcoastlines(linewidth=1)
mymap.drawmeridians(np.arange(0, 360, 5),labels=[False,False,False,True])
mymap.drawparallels(np.arange(-90, 90, 5),labels=[True,False,False,False])
x, y = mymap(lon,lat) #projeção cartográfica
mymap=mymap.contourf(x,y,qv,cmap='jet') #campo qv
plt.colorbar(mymap,label=r'$q_v (g/kg)$')
plt.title('Projecção UTM')
```

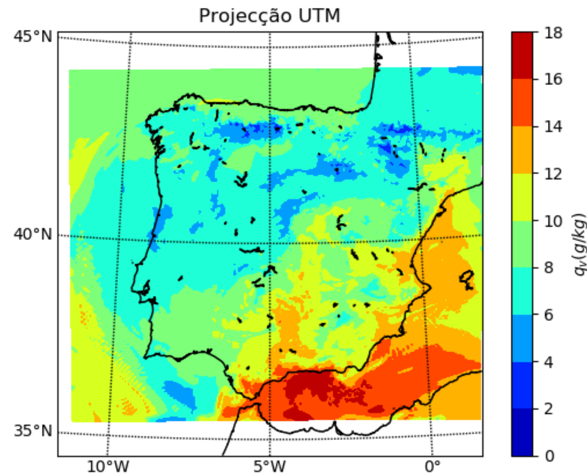

Figura 6-5 – O mesmo que a Figura 6-3, utilizando a projeção UTM.

No caso, por exemplo de uma projeção Mercator, o código seria:

```
plt.figure()
mymap2 = Basemap(projection='merc',llcrnrlat=35,urcrnrlat=45,\
            llcrnrlon=-12,urcrnrlon=3,lat_ts=40,resolution='f')
mymap2.drawcoastlines(linewidth=1)
mymap2.drawmeridians(np.arange(0, 360, 5),labels=[False,False,False,True])
mymap2.drawparallels(np.arange(-90, 90, 5),labels=[True,False,False,False])
x, y = mymap2(lon,lat) #projeção cartográfica
mymap2=mymap2.contourf(x,y,qv,cmap='jet') #campo qv
plt.colorbar(mymap2,label=r'$q_v (g/kg)$')
plt.title('Projecção Mercator')
```

dando origem à Figura 6-6. Existem muitas outras projeções disponíveis.

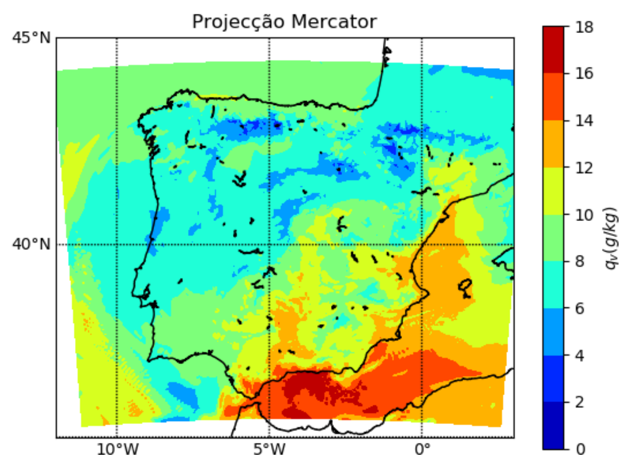

Figura 6-6 – O mesmo que a Figura 6-3, utilizando a projeção Mercator.

Os exemplos apresentados mostram que o módulo **basemap** não prescinde das funções do módulo **matplotlib.pyplot**. Estas últimas continuam a ser utilizadas mas o seu resultado é modificado pelas projeções.

6.4 Produção de grelhas bidimensionais

7 Sistemas de equações lineares

7.1 Identificação do problema e formulação matemática

Muitos problemas passam pela solução de sistemas de equações algébricas lineares. Formalmente, esses problemas consistem na determinação do conjunto de N variáveis $\{x_1, x_2, \dots, x_N\}$, dados $\{a_{ij}, b_i, i = 1, \dots, M\}$ e as relações:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1N}x_N = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2N}x_N = b_2 \\ \dots \\ a_{M1}x_1 + a_{M2}x_2 + \cdots + a_{MN}x_N = b_M \end{cases} \quad (7\text{-}1)$$

O problema pode também ser escrito de forma condensada, usando notação matricial:

$$A\vec{x} = \vec{b} \quad (7\text{-}2)$$

Vamos considerar unicamente o caso em que M=N. Os outros dois casos (casos com mais ou menos equações do que incógnitas), que estão também associados a importantes problemas de Física, são demasiado especializados para o âmbito deste curso.

Se M=N (a partir deste ponto referir-se-á unicamente N como a dimensão do sistema de equações) e as equações forem linearmente independentes, o sistema tem solução. Caso contrário, diz-se que a matriz dos coeficientes A é singular (nesse caso, tem determinante nulo). Em sistemas quase-singulares, ou quando N é muito grande, o erro de arredondamento pode dificultar a obtenção da solução.

7.2 Métodos numéricos

Existem basicamente duas famílias de métodos de solução de sistemas de equações lineares: métodos **diretos** e **iterativos**. Neste curso vamos considerar unicamente métodos diretos, que são geralmente os mais apropriados para sistemas de dimensão "manejável". Tanto os métodos diretos como os métodos iterativos podem ser desenvolvidos de modo a tirarem partido de casos em que a matriz dos coeficientes é ***esparsa***, isto é tem grande número de termos nulos, com uma disposição organizada. O caso mais típico de uma matriz esparsa é constituído pela ***matriz tridiagonal*** em que todos os elementos são nulos exceto os da diagonal principal e das duas subdiagonais adjacentes.

Os métodos de solução de sistemas de equações lineares baseiam-se na imposição de uma sequência de transformações equivalentes da matriz dos coeficientes (A) e do vetor dos termos independentes ($\vec{b}$). Cada transformação equivalente, por definição, procede a modificações daqueles termos sem alterar a solução do sistema. As transformações básicas são:

a) Multiplicação de equação por uma constante;

b) Substituição de uma equação pelo resultado de uma combinação linear dessa equação com outra (adição de uma equação com o produto de outra por uma constante);

c) Troca de duas equações.

Deve notar-se que as modificações de uma equação implicam alterações em todos os seus termos (na matriz e no vetor independente).

7.3 Eliminação de Gauss

O método da eliminação de Gauss baseia-se no seguinte facto: se o sistema de equações for transformado (por uma sequência de transformações equivalentes) num sistema **triangular superior**, isto é um sistema da forma:

$$\begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1N} \\ 0 & u_{22} & \cdots & u_{2N} \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & u_{NN} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_N \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \cdots \\ c_N \end{bmatrix} \tag{7-3}$$

A solução pode ser obtida muito facilmente começando na última equação e substituindo para trás ("backsubstitution"). O método da eliminação de Gauss consiste pois na construção de um conjunto de transformações que permitam a eliminação progressiva dos termos da matriz dos coeficientes que se encontram abaixo da diagonal principal.

Para uma boa compreensão do método recomenda-se o estudo dos exemplos simples apresentados na bibliografia e uma consulta sobre o desenvolvimento do algoritmo. Em geral, para $k = 1, \dots N - 1$, usa-se a equação k para eliminar o termo em x_k das equações $j = k + 1, \dots, N$. Para o efeito multiplica-se a equação k por $-a_{jk}/a_{kk}$ e soma-se à equação j. É evidente, no entanto, que se $a_{kk} = 0$ o processo não funciona. A situação anterior não implica necessariamente que a matriz seja singular, podendo acontecer que existam outras equações ainda disponíveis que possam ser utilizadas nessa fase da eliminação. O problema consiste, portanto, em, em cada passo, escolher para eliminar a variável x_k, não a equação k, mas qualquer outra das equações ainda disponíveis (*m*) para a qual se possa calcular $-a_{jk}/a_{mk}$. A essa equação chama-se equação *pivot*. O algoritmo da eliminação de Gauss deve pois, no caso geral, incluir uma escolha de *pivot*. Em certos casos há garantia de que a situação $a_{kk} = 0$ nunca ocorre e a estratégia de *pivot* é desnecessária.

A utilização de estratégia de *pivot* é equivalente à realização da transformação equivalente de troca entre duas equações no sistema. No entanto, para evitar os custos computacionais de proceder a essa troca, que envolve em cada caso a $3(N + 1)$ movimentações de coeficientes, utiliza-se um sistema de indexação indireta que mantém um registo das equações já utilizadas e das que falta utilizar.

7.3.1 Algoritmo sem pivot

Neste caso vamos escrever uma rotina PYTHON usável a partir de outras rotinas ou no espaço de trabalho. Para o efeito escrevemos o ficheiro "gaussElim.py". Esse ficheiro define a função **gaussElim** que resolve o sistema de n equações lineares $A\vec{x} = \vec{b}$. A é uma matriz quadrada ($n \times n$), $\vec{x}$ e $\vec{b}$ são vetores coluna com *n* componentes. Notar: a utilização do operador **or** (OU lógico); a forma das relações de igualdade "**==**" e desigualdade "**!=**" em operações lógicas; a utilização de um ciclo regressivo `range()`, i.e., com passo negativo; a omissão do passo no ciclo de passo 1 (`k=1:n`).

```
"""
Eliminação de Gauss sem pivot resolve Mx=d
"""
def gaussElim(M,d):
```

```
    A=np.copy(M) #evita a alteração da matriz de coeficientes, copiando-a
    b=np.copy(d) #idem para os termos independentes
    x=np.zeros(b.shape)
    Ash=A.shape;
    n=Ash[0];n2=Ash[1]
    Bsh=b.shape
    n3=Bsh[0]
    if n!=n2 or n3!=n or len(Bsh)!=1 or len(Ash)!=2:
        print('Erro de dimensão')
        x=float('nan')
        return x
    #triangulação
    for k in range(0,n-1):
        if A[k,k]==0:
            x=float('nan')
            return x
        for j in range(k+1,n):
            e=A[j,k]/A[k,k]
            for m in range(k+1,n):
                A[j,m]=A[j,m]-e*A[k,m]
            b[j]=b[j]-e*b[k]
    #substituição
    for k in range(n-1,-1,-1):#=n:-1:1 %indice regressivo
        sum=0.
        for j in range (k+1,n):
            sum=sum+A[k,j]*x[j]

        x[k]=(b[k]-sum)/A[k,k]
    return x
```

A função **gauss** definida anteriormente pode ser utilizada diretamente no espaço de trabalho ou em outro *script* (ficheiro .py).

Problema 7-1

Resolva os 4 sistemas de equações (todos com solução $x_1=x_2=x_3=1$). Verifique a solução calculando $\boldsymbol{A}\vec{x}$ e comparando com $\vec{b}$.

$$\begin{bmatrix} 0.00001 & 5000 & 1000 \\ 4 & 1 & 1 \\ 2 & 3 & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2000.0001 \\ 6 \\ 6 \end{bmatrix}, \begin{bmatrix} 0.01 & 1000 & 1000 \\ 4 & 1 & 1 \\ 2 & 3 & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2000.01 \\ 6 \\ 6 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1000 & 1000 \\ 4 & 1 & 1 \\ 2 & 3 & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2001 \\ 6 \\ 6 \end{bmatrix}, \begin{bmatrix} 1000 & 1000 & 1000 \\ 4 & 1 & 1 \\ 2 & 3 & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3000 \\ 6 \\ 6 \end{bmatrix}$$

Solução da alínea a:

```
import numpy as np
A=np.array(([0.00001,1000.,1000.],[4.,1.,1,],[2.,3.,1.]))
b=np.array([2000.0001,6.,6.])
x=gaussElim(A,b)
print('x=',x)
y=np.matmul(A,x)
print('A*x=',y)

>> x= [ 0.99999997  0.99999997  1.00000012]
>> A*x= [ 2000.0001         5.99999999     5.99999998]
```

Notas importantes:

(a) É necessário importar **numpy** antes de definir os dados e de chamar a rotina de eliminação de Gauss;

(b) Em PYTHON não existe diferença entre vetor linha e vetor coluna;

(c) O produto matricial Ax é calculado com recurso à função **numpy.matmul**.

(d) Note que o vetor **y** é idêntico ao vetor **b** (a menos do erro de arredondamento). O erro de arredondamento depende dos dados.

7.4 Soluções recorrendo a funções pré-existentes

De facto, o PYTHON inclui funções muito otimizadas que podem ser utilizadas para resolver sistemas de equações lineares. Assim poderíamos escrever simplesmente:

```
x=np.linalg.solve(A,b)
print('x=',x)

>> x= [ 0.99999998  0.99999998  1.00000011]
```

Neste caso existe ainda a vantagem de que o PYTHON utilizar uma rotina mais geral e mais eficiente que a eliminação de Gauss (sem pivot).

Alternativamente também poderíamos fazer:

```
Ainv=np.linalg.inv(A)
x=np.matmul(Ainv,b)
print('x=',x)

>> x= [ 0.99999998  0.99999998  1.00000011]
```

Neste caso utilizamos uma função que calcula a matriz inversa de A (**inv**) e resolvemos o sistema com um produto matricial ($\vec{x} = A^{-1}\vec{b}$). É uma operação mais lenta que a solução do sistema de equações, mas pode ser útil em diversas aplicações, por exemplo se se pretender resolver repetidamente o mesmo sistema de equações com mudança do termo independente ($\vec{b}$).

Problema 7-2

Um painel solar térmico (superfície negra ideal) possui 3 vidros completamente transparentes para a radiação solar, e com absorvidade *α* para a radiação infravermelha. A Figura 7-1 mostra

o diagrama de fluxos radiativos desse painel. Admitindo que o sistema se encontra em equilíbrio radiativo (não existindo outros fluxos de energia para o exterior), note que os fluxos de radiação infravermelha emitida (E_0,E_1,E_2,E_3) satisfazem o sistema de 4 equações:

$$\begin{cases} -E_0 + E_1 + (1-a)E_2 + (1-a)^2E_3 + E_S = 0 \\ aE_0 - 2E_1 + aE_2 + a(1-a)E_3 = 0 \\ a(1-a)E_0 + aE_1 - 2E_2 + aE_3 = 0 \\ a(1-a)^2E_0 + a(1-a)E_1 + aE_2 - 2E_3 = 0 \end{cases}$$

(a) Calcule os fluxos emitidos pelas 4 superfícies, resolvendo o sistema de equações, para o caso $a=0.8$, E_S=500Wm^{-2};

(b) Admita que cada uma das superfícies (0 a 3) satisfaz a lei de Stefan-Boltzmann: $E = \varepsilon\sigma T^4$, em que ε é emissividade, igual á absorvidade (a, lei de Kirchoff), e $\sigma = 5.67 \times 10^{-8} Wm^{-2}K^{-4}$ é a constante de Stefan-Boltzmann, calcule as temperaturas da superfície negra e dos vidros;

(c) Escreva uma função que dados os valores de a e de E_S, devolva o valor dos fluxos e das temperaturas;

(d) Generalize o caso de *N* vidros;

(e) Generalize para o caso de vidros com absorvidade variável $(a_1, a_2, \dots, a_N)$;

(f) Generalize para o caso em que os vidros absorvem também radiação solar.

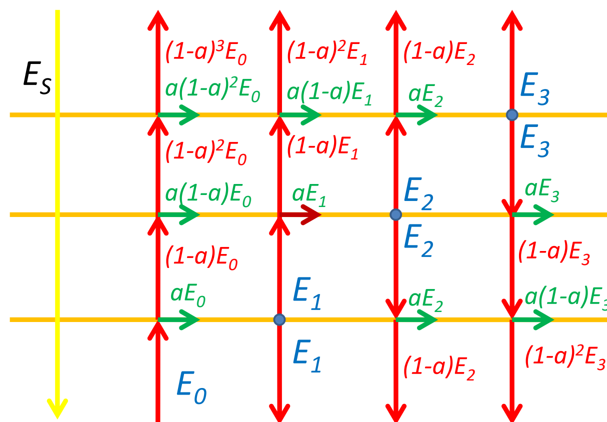

Figura 7-1 Painel solar com múltiplos vidros.

8 Integração numérica

Existem muitas situações em que a solução de problemas em Física passam pelo cálculo de integrais. A necessidade de proceder a esse cálculo por métodos numéricos surge sempre que a função integranda oferece dificuldades ao tratamento analítico, por não ser integrável analiticamente ou por a integração analítica ser muito laboriosa, ou quando os limites de integração impedem ou dificultam o recurso a métodos analíticos. Outro caso em que o recurso a métodos numéricos é obrigatório é quando a função integranda não é conhecida analiticamente, sendo o seu valor dado (por exemplo a partir de medida experimental) num número finito de pontos.

Para simplificar, vão considerar-se dois exemplos de Mecânica. O primeiro, passa pela solução de um integral simples. O segundo, exige o cálculo de um integral triplo difícil de calcular analiticamente dada a definição dos limites de integração.

8.1 Métodos numéricos

Existem diversos métodos numéricos utilizáveis no cálculo de integrais. No caso do problema 1.1, vão considerar-se 3 métodos possíveis, de simples implementação, mas que são, em geral, suficientemente eficientes: método do ponto médio, método do trapézio, método de Simpson. No caso do problema 1.2 introduzir-se-á uma técnica extremamente geral de integração numérica, o método de Monte Carlo, que permite resolver, ainda que com um tempo de cálculo relativamente elevado, muitos problemas de difícil tratamento.

8.1.1 Cálculo de integrais por interpolação entre pontos regularmente espaçados

O método mais simples de calcular integrais consiste em utilizar diretamente a definição de integral de Riemman. Neste caso, divide-se o domínio de integração em N intervalos, limitados por N+1 pontos, e calcula-se o integral como a soma das áreas definidas em cada intervalo. Para calcular estas áreas considera-se que a função varia em cada intervalo de acordo com uma lei simples, concretamente, que ela é aproximada por um polinómio de baixo grau.

Assim, em todos os casos vai considerar-se a discretização:

$$x_k = x_0 + kh\Delta x \tag{8-1}$$

em que h é o intervalo de discretização.

(a) Método do ponto médio

Este método consiste em admitir que o valor médio da função em cada intervalo é igual ao seu valor no ponto médio. Esta hipótese será satisfeita, em particular, se a variação for linear dentro do intervalo.

Neste caso, com N intervalos, calcula-se o integral na forma:

$$\int_{x_0}^{x_N} f(x)dx \approx S_N = h\sum_{k=1}^{N} f_{k-\frac{1}{2}} = h\left[f_{1/2} + f_{3/2} + \cdots + f_{N-1/2}\right] \tag{8-2}$$

(b) Método do trapézio

Este método é semelhante ao anterior. Neste caso, no entanto, toma-se como valor médio da função integranda a média aritmética do seu valor nas duas extremidades do intervalo. Geometricamente, esta regra consiste em aproximar o integral pela área de um trapézio, isto é, em aproximar a função integranda por um segmento de reta no intervalo.

Com N intervalos tem-se:

$$\int_{x_0}^{x_N} f(x)dx \approx S_N = h\sum_{k=1}^{N}\frac{f_{k-1}+f_k}{2} = h\left[\frac{f_0}{2} + f_1 + \cdots + f_{N-1} + \frac{f_N}{2}\right] \tag{8-3}$$

(c) Método de Simpson

Tanto no método do ponto médio como no método do trapézio o erro decresce com N^2. É possível obter métodos de convergência mais rápida, se a função integranda for suficientemente suave, recorrendo a polinómios interpoladores de ordem mais elevada. O método de Simpson corresponde ao passo seguinte nesse sentido, utilizando como interpolador uma parábola. Nesse caso, mercê de um cancelamento que ocorre no desenvolvimento do problema, obtém-se um método cujo erro decresce com N^4.

Para definir uma parábola são precisos três pontos. Assim, impondo o valor da função nos dois extremos do intervalo e no ponto médio, obtêm-se por substituição:

$$\int_{x_0}^{x_1} f(x)dx \approx h\left[\frac{1}{3}f_0 + \frac{4}{3}f_1 + \frac{1}{3}f_2\right] \tag{8-4}$$

com N+1 pontos obtém-se:

$$\int_{x_0}^{x_N} f(x)dx \approx S_N = h\left[\frac{1}{3}f_0 + \frac{4}{3}f_1 + \frac{2}{3}f_2 + \frac{4}{3}f_3 + \cdots + \frac{2}{3}f_{N-2} + \frac{4}{3}f_{N-1} + \frac{1}{3}f_N\right] \tag{8-5}$$

8.1.2 Método de Monte Carlo

O método de Monte Carlo baseia-se no teorema do mesmo nome. De acordo com este teorema, dado um volume V e N pontos (x_k) aleatoriamente distribuídos nesse volume, com uma distribuição uniforme, tem-se:

$$\iiint_V f(x,y,z)\,dV \approx V\langle f\rangle \pm V\sqrt{\frac{\langle f^2\rangle - \langle f\rangle^2}{N}} \tag{8-6}$$

em que:

$$\langle f\rangle = \frac{1}{N}\sum_{k=1}^{N} f(x_k, y_k, z_k),\quad \langle f^2\rangle = \frac{1}{N}\sum_{k=1}^{N}[f(x_k, y_k, z_k)]^2 \tag{8-7}$$

Em geral, só é fácil obter uma série de pontos aleatoriamente distribuídos com distribuição uniforme em volumes muito simples, como por exemplo um paralelepípedo. Se não for esse o caso, o método pode ainda ser utilizado substituindo o volume V por um paralelepípedo P que contenha esse volume e redefinindo a função integranda. Nesse caso faz-se:

$$\iiint_V f dV = \iiint_P g dP,\ g(x) = \begin{cases} f(x), x \in V \\ 0, x \notin V \end{cases} \tag{8-8}$$

O preço a pagar pela substituição anterior consiste numa redução da velocidade de convergência do método, inversamente proporcional ao aumento do volume de V para P.

8.2 Aplicações

Exemplo 8-1 Cálculo do centro de massa de uma placa fina homogénea

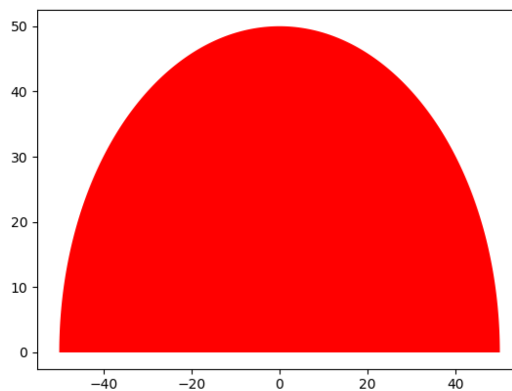

Figura 8-1 Placa fina

Vai-se considerar o problema de calcular o centro de massa da placa fina apresentada na Figura 8-1, de densidade ρ=3.5kg m^{-2}, cuja forma semicircular com raio $R = 50m$, i.e.:

$$f(x) = \sqrt{R^2 - x^2} \tag{8-9}$$

A solução do problema consiste no cálculo do integral:

$$\vec{r}_{CM} = \frac{1}{m}\int_S \rho\vec{r}dS \tag{8-10}$$

onde $\vec{r}$ é o centro de massa do elemento dS, m é a massa da placa, dada por:

$$m = \int_S \rho dS \tag{8-11}$$

e ρ é a massa por unidade de área.

Vamos utilizar o método do trapézio, considerando $h = \Delta x = 10\ m, 1m, 0.1m, 0.01m, 0.001m\ e\ 0.0001$. Os elementos dS são trapézios com centro de massa em $((x_k + x_{k+1})/2, \left(f(x_k) + f(x_{k+1})\right)/2)$ e tem-se que $dS = f(x)dx$.

```
"""
Cálculo do centro de gravidade de uma placa semi-circular
Integrando pelo método do trapézio
"""
import numpy as np
from math import pi
import matplotlib.pyplot as plt
plt.close('all')
ro=3.5
R=50
kp=0
for n in [11,101,1001,10001,100001,1000001]:
    x=np.linspace(-50.,50.,n)
    f=np.sqrt(R**2-x**2)
    y=f/2
    kp=kp+1
    plt.subplot(2,3,kp)
    plt.plot(x,f)
    massa=ro*(f[0]+f[n-1])/2
    xcm=ro*(f[0]*x[0]+f[n-1]*x[n-1])/2
    ycm=ro*(f[0]*y[0]+f[n-1]*y[n-1])/2
    for k in range(1,n):
        massa=massa+ro*f[k]
        xcm=xcm+ro*f[k]*x[k]
        ycm=ycm+ro*f[k]*y[k]

    massa=massa*dx
    xcm=xcm*dx/massa
    ycm=ycm*dx/massa
    print('n=%6i massa=%10.3f kg xcm=%10.3f m ycm=%12.8f m\n' %
      (n,massa,xcm,ycm))
    plt.scatter(xcm,ycm)
    plt.text(xcm+5*dx,ycm,'y=%8.5f \n n=%8i' % (ycm,n))
```

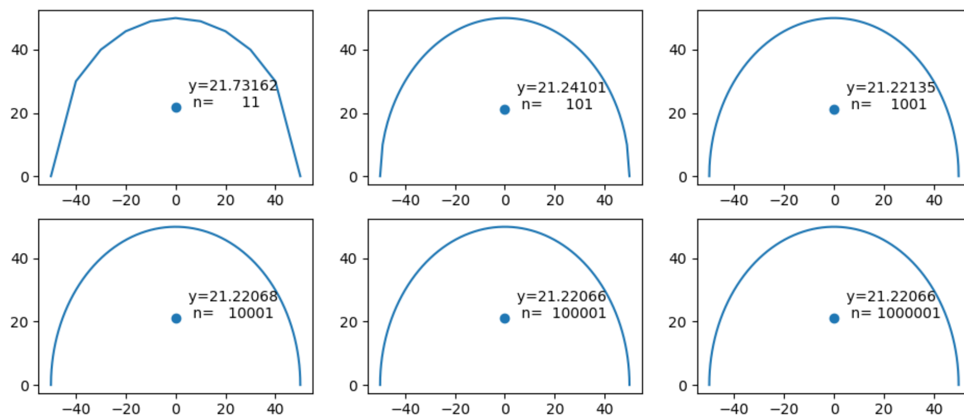

Figura 8-2 Centro de massa da placa fina

Notar:

(a) Utilizou-se a função **linspace** cujo argumento é o número de pontos (= número de intervalos +1) e não a resolução h;

(b) Utilizaram-se **subplot**'s para apresentar as várias soluções;

(c) Fez-se a escrita do resultado no gráfico utilizando a função **text**, com formato, em duas linhas com o código de mudança de linha **\n**.

(d) Convergência da solução para grandes valores de n.

Exemplo 8-2 Cálculo do centro de massa de uma secção de um toro

Considere-se o problema do cálculo do centro de massa do sólido, de densidade $\rho = 1.2{\times}10^3\ kg\ m^{-3}$*, obtido a partir de um toro (de raio menor 2 e raio maior 4) dado por:*

$$z^2 + \left(\sqrt{x^2 + y^2} - 3\right)^2 \leq 1 \qquad (8\text{-}12)$$

por intermédio do corte:

$$x \geq -0.5, y \geq -3 \quad (8\text{-}13)$$

Neste caso o problema consiste no cálculo do integral triplo:

$$\frac{\iiint_V \rho \vec{r} dV}{\iiint_V \rho dV} \quad (8\text{-}14)$$

com limites de integração dados pelas desigualdades anteriores. O problema não é tratável analiticamente dada a geometria peculiar do corpo.

O código PYTHON seguinte, resolve este problema para uma série de valores do número de amostras.

```
"""
Cálculo da massa e do centro de gravidade de um toro cortado,
pelo método de MonteCarlo
"""
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import random
doplot=False #liga o output grafico (pode ser lento)
fPlot=1 #frequência de scatter (if doplot:)

if doplot:
    plt.close('all')
ro=1.2e3 # densidade
for n in [1000,10000,100000,1000000,10000000]:
    xcut=-0.5;ycut=-3;# %pontos de corte do sólido
    # NOTA: devia limitar-se o paralelipipedo fazendo xmin=xcut;ymin=ycut
    # aqui decidiu incluir-se o corte na definição do sólido
    # mantendo o paralelepípedo no limite exterior do toro
    rhmax=4 # raio exterior do toro
    rzmax=1 # raio transversal
    # limites do paralelepípedo exterior
    xmin=-rhmax;xmax=rhmax;xlen=xmax-xmin;
    ymin=-rhmax;ymax=rhmax;ylen=ymax-ymin;
    zmin=-rzmax;zmax=rzmax;zlen=zmax-zmin;
    V=xlen*ylen*zlen; #volume do paralelepípedo
    Xp=[xmin,xmin,xmin,xmax]; #grafico
    Yp=[ymin,ymin,ymax,ymax]; #grafico
    Zp=[zmin,zmax,zmax,zmax]; #grafico
    if doplot:
        fig=plt.figure()
        ax=fig.add_subplot(111, projection='3d')
        ax.plot(xs=Xp,ys=Yp,zs=Zp)
    Sro=0;Sro2=0;Sx=0;Sx2=0;Sy=0;Sy2=0;Sz=0;Sz2=0;
    kIN=0
    Axes3D.view_init(ax,elev=60,azim=-145)
```

```
    for k in range(n):
        X=xmin+xlen*np.random.rand();
        Y=ymin+ylen*np.random.rand();
        Z=zmin+zlen*np.random.rand();
        if (Z**2+np.sqrt(X**2+Y**2)-3)**2<=1 and X>xcut and Y>ycut:#
      %Definicao do toro e do corte
            Sro=Sro+ro;
            Sro2=Sro2+ro*ro;
            Sx=Sx+ro*X;
            Sx2=Sx2+(ro*X)**2;
            Sy=Sy+ro*Y;
            Sy2=Sy2+(ro*Y)**2;
            Sz=Sz+ro*Z;
            Sz2=Sz2+(ro*Z)**2;
            kIN=kIN+1
            if kIN % fPlot==0 and doplot:
                ax.scatter(X,Y,Z,'.',color='blue',s=1) #%Output gráfico
    plt.xlabel('x');plt.ylabel('y');
    Sro=Sro/n;Sro2=Sro2/n;massa=V*Sro;delmas=V*np.sqrt((Sro2-Sro**2)/n);
    Sx=Sx/n;Sx2=Sx2/n;xCM=V*Sx;delx=V*np.sqrt((Sx2-Sx**2)/n);
    xCM=xCM/massa;delx=delx/massa;
    Sy=Sy/n;Sy2=Sy2/n;yCM=V*Sy;dely=V*np.sqrt((Sy2-Sy**2)/n);
    yCM=yCM/massa;dely=dely/massa;
    Sz=Sz/n;Sz2=Sz2/n;zCM=V*Sz;delz=V*np.sqrt((Sz2-Sz**2)/n);
    zCM=zCM/massa;delz=delz/massa;
    ax.scatter(xCM,yCM,zCM,color='red')
    print('n=',n)
    print('massa=%15.7e kg +-%15.7e kg (Err:%8.5f %%) ' %
      (massa,delmas,delmas/massa*100))
    print('xCM=%15.7e +-%15.7e (Err:%8.5f %%) ' %
      (xCM,delx,delx/rhmax*100))
    print('yCM=%15.7e +-%15.7e (Err:%8.5f %%) '% (yCM,dely,dely/rhmax*100))
    print('zCM=%15.7e +-%15.7e (Err:%8.5f %%) ' %
      (zCM,delz,delz/rzmax*100))
```

O cálculo de uma série de *N* números aleatórios uniformemente distribuídos no intervalo [0,1] é realizado com a função **numpy.random.rand()**. De seguida estes números são re-escalados para o paralelepípedo que contém o toro. Os cálculos limitam-se a aplicar as fórmulas. Utilizando **n=10000** obtém-se um cálculo com uma precisão de cerca de 1.5%, com **n=1000000**, obtém-se um cálculo com uma precisão de cerca de 1.5/1000. O método converge, pois, muito lentamente. Para a série de valores de n considerada, obtêm-se os resultados:

```
n= 1000
massa=  4.0550400e+04 kg +-  2.1410760e+03 kg (Err: 5.28004 %)
xCM=  1.6195630e+00 +-  1.0882174e-01 (Err: 2.72054 %)
yCM=  1.8168655e-01 +-  1.2346054e-01 (Err: 3.08651 %)
zCM= -1.0372600e-02 +-  3.5871234e-02 (Err: 3.58712 %)
n= 10000
massa=  4.2101760e+04 kg +-  6.8514759e+02 kg (Err: 1.62736 %)
xCM=  1.6501142e+00 +-  3.4397884e-02 (Err: 0.85995 %)
yCM=  2.1816820e-01 +-  3.8036253e-02 (Err: 0.95091 %)
zCM=  2.1531656e-03 +-  1.0656458e-02 (Err: 1.06565 %)
n= 100000
massa=  4.2347520e+04 kg +-  2.1705452e+02 kg (Err: 0.51256 %)
```

```
xCM=  1.6567390e+00 +-  1.0799212e-02 (Err: 0.26998 %)
yCM=  2.8423524e-01 +-  1.2037349e-02 (Err: 0.30093 %)
zCM=  7.1545927e-04 +-  3.3715172e-03 (Err: 0.33715 %)
n= 1000000
massa=  4.1716685e+04 kg +-  6.8318380e+01 kg (Err: 0.16377 %)
xCM=  1.6518402e+00 +-  3.4461274e-03 (Err: 0.08615 %)
yCM=  2.6723530e-01 +-  3.8247234e-03 (Err: 0.09562 %)
zCM= -1.1108773e-03 +-  1.0748827e-03 (Err: 0.10749 %)
n= 10000000
massa=  4.1719695e+04 kg +-  2.1604657e+01 kg (Err: 0.05179 %)
xCM=  1.6552814e+00 +-  1.0906022e-03 (Err: 0.02727 %)
yCM=  2.6552905e-01 +-  1.2101689e-03 (Err: 0.03025 %)
zCM=  4.4124870e-04 +-  3.3987856e-04 (Err: 0.03399 %)
```

Se se ativar a opção de output gráfico (`doplot=True` e `n=10000`), obtém-se a Figura 8-3. Deve notar-se que a saída gráfica não deve ser ativada com n grande, sob pena de o sistema bloquear.

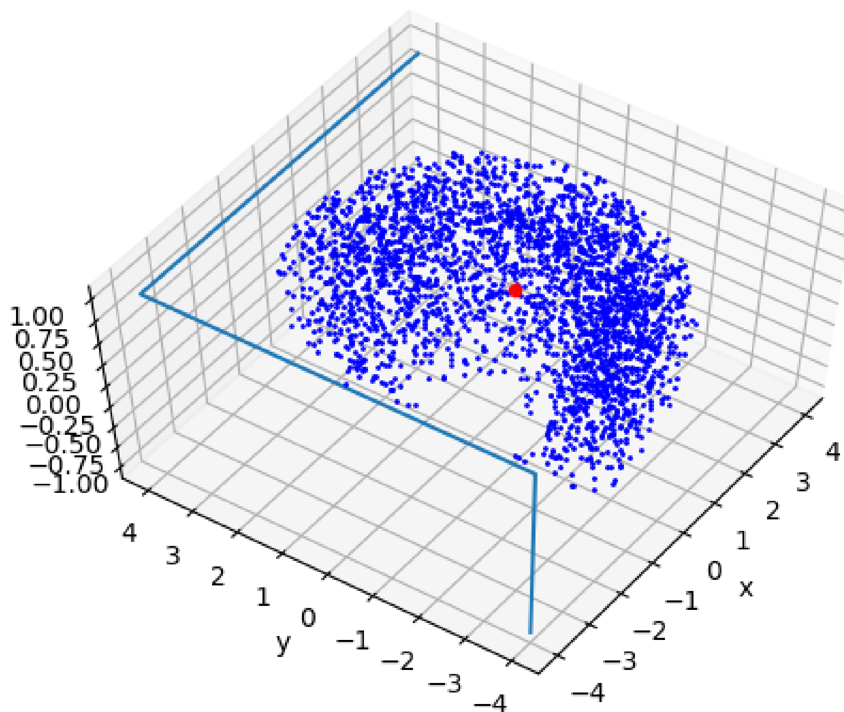

Figura 8-3 Centro de massa de um toro cortado (círculo vermelho). Método de Monte Carlo com 10000 pontos. Nuvem contém pontos interiores ao toro.

Problema 8-1

Utilize o método de Monte Carlo para calcular o volume de uma esfera de raio 5 cm a que foi retirado um cilindro de raio 1 cm, centrado, para a passagem de um eixo.

(a) Escreva o programa na forma de uma *função* com input do máximo de iterações (n), e output do volume e do erro estimado.

(b) Construa um algoritmo que, utilizando a função anterior, calcule o volume com um erro relativo especificado.

9 Solução de equações diferenciais ordinárias com condições fronteira num ponto

9.1 Equações diferenciais ordinárias

9.1.1 Lei de Newton do arrefecimento

Vamos considerar um problema simples de termodinâmica: o problema do arrefecimento de uma chávena de café. Em primeira aproximação o processo de arrefecimento obedece à lei de Newton do arrefecimento, expressa pela relação diferencial:

$$\frac{dT}{dt} = -\alpha(T - T_a) \qquad (9\text{-}1)$$

em que $T(t)$ é a temperatura do café em cada instante, T_a a temperatura do ar e α um coeficiente de transferência.

O problema considerado é um problema unidimensional e a sua solução consiste na solução da equação diferencial ordinária (9-1) dado o valor inicial da temperatura do café e os valores das constantes T_a e κ. A solução analítica é dada por:

$$T = T_a + (T_0 - T_a)e^{-\alpha t} \qquad (9\text{-}2)$$

9.2 Integração das equações da Mecânica

Existem muitos outros problemas em Física que apresentam uma estrutura formalmente idêntica. Esse é o caso nomeadamente da solução do problema típico de Mecânica: dado um ponto material atuado por uma lei de forças $\vec{F}(t)$, inicialmente na posição $\vec{r}_0$ e com velocidade $\vec{v}_0$, calcular a sua posição no instante t. Um exemplo muito simples desta família de problemas é dado pelo movimento unidimensional de um ponto material de massa m, com velocidade inicial ascendente $= v_0\vec{e}_z$, num campo gravítico uniforme de aceleração $-g\vec{e}_z$. A lei do movimento desse corpo (segunda lei de Newton da Mecânica) pode escrever-se:

$$\frac{d^2z}{dt^2} = -g \qquad (9\text{-}3)$$

O problema apresentado na equação (9-3) apresenta uma dificuldade adicional em relação ao problema (9-1) visto exigir a solução de uma equação diferencial ordinária de segunda ordem. Por esse motivo, a sua solução requer o estabelecimento de duas condições iniciais, para a posição e para a velocidade. É claro, no entanto, que uma equação diferencial de segunda ordem pode ser facilmente substituída por um sistema equivalente de equações diferenciais de primeira ordem. Assim, em vez da equação (9-3) pode escrever-se:

$$\begin{cases} \frac{dv}{dt} = -g \\ \frac{dz}{dt} = v \end{cases} \quad (9\text{-}4)$$

Prosseguindo na mesma linha, podem igualmente considerar-se problemas bi ou tridimensionais. Um exemplo interessante é fornecido pelo movimento da Terra na sua órbita em torno do Sol. A lei do movimento é neste caso dada pela conjunção da Lei da Atracão Universal com a Segunda Lei de Newton da Mecânica, expressa na relação:

$$\frac{d^2\vec{r}}{dt^2} = -\frac{GM}{r^3}\vec{r} \quad (9\text{-}5)$$

em que M e m são, respectivamente, as massas do Sol e da Terra, G é a constante de gravitação e $\vec{r}$ o vector posição da Terra com referência ao centro do Sol. Numa boa aproximação, pode considerar-se que o Sol e a Terra são pontuais e que o Sol se encontra fixo. Por outro lado o problema pode ser analisado de forma bidimensional, no plano da eclítica. Neste caso, a equação vetorial dá origem a um sistema de duas equações escalares, uma para cada componente:

$$\begin{cases} \frac{d^2x}{dt^2} = -\frac{GM}{(x^2+y^2)^{3/2}}x \\ \frac{d^2y}{dt^2} = -\frac{GM}{(x^2+y^2)^{3/2}}y \end{cases} \quad (9\text{-}6)$$

Desdobrando as duas equações diferenciais de segunda ordem, obtemos, portanto, neste caso, um sistema de 4 equações diferenciais, cuja solução exige quatro constantes de integração (as condições iniciais para as duas componentes da velocidade e da posição).

Todos os problemas considerados anteriormente têm tratamento analítico simples, pelo que é fácil confirmar os resultados a obter numericamente. No entanto, é preciso esclarecer que os métodos numéricos podem ser utilizados (e nesse caso é que são verdadeiramente úteis) em situações de difícil ou impossível solução analítica.

Finalmente deve notar-se que todos os problemas apresentados são traduzidos por sistemas de equações diferenciais ordinárias (uma só variável independente) em que as condições fronteira são fornecidas num único ponto, isto é, para t=0. Outros casos exigirão tratamento diferenciado.

9.3 Métodos numéricos (Método de Euler)

A solução numérica de (sistemas de) equações diferenciais, ordinárias ou não, passa necessariamente pela sua transformação (em sistemas) de equações algébricas. Existem várias metodologias possíveis para operar essa transformação. Todas elas exigem, no entanto, a redução do problema de um domínio contínuo (no tempo e/ou no espaço) para um domínio discreto, isto é a limitação do número de graus de liberdade a um número finito. Na prática, isso quer dizer que a equação (ou o sistema) vai ser resolvida numa ***rede*** discreta de pontos da variável independente. Isto é define-se um intervalo de discretização Δx e resolve-se o problema para $x = x_0 + k\Delta x\ \{k = 1, \dots, N\}$. Deve notar-se que x é a variável independente que, nos casos apresentados anteriormente é a variável tempo.

Uma vez estabelecida a rede, a metodologia mais simples consiste em substituir diretamente as derivadas presentes nas equações por aproximações obtidas por razões entre ***diferenças finitas*** das diferentes variáveis. Assim, uma derivada de primeira ordem pode ser, por exemplo, substituída por:

$$\left(\frac{dy}{dx}\right)_{x=a} \approx \frac{y(a+\Delta x)-y(a)}{\Delta x} \qquad (9\text{-}7)$$

A utilização da aproximação anterior, ou de qualquer outra metodologia, envolve, pelo menos, a consideração de três conceitos básicos, que não serão, no entanto, analisados em nenhuma profundidade neste curso:

(i) Qual é a ***precisão*** desta representação ?

(e) Existe ***convergência*** ? Isto é a solução numérica tende para a solução analítica quando o intervalo de discretização $\Delta x \to 0$?

(f) O método é ***estável*** ? Isto é, o erro cresce de forma "controlada" ou exponencialmente ?

No caso da representação sugerida anteriormente, é simples verificar que existe ***convergência***. O seu grau de ***precisão*** pode ser analisado recorrendo à série de Taylor:

$$y(x+\Delta x) = y(x) + \frac{dy}{dx}\Delta x + \frac{1}{2}\frac{d^2y}{dx^2}\Delta x^2 + \frac{1}{3!}\frac{d^3y}{dx^3}\Delta x^3 + \cdots \qquad (9\text{-}8)$$

Resolvendo para a primeira derivada pode escrever-se:

$$y' = \frac{dy}{dx} = \frac{y(x+\Delta x)-y(x)}{\Delta x} + E(\Delta x) \qquad (9\text{-}9)$$

em que $E(\Delta x)$ representa o ***erro de truncatura***, cujo desenvolvimento em série de potências de Δx tem como termo de menor ordem um termo de primeira ordem. Nestas condições diz-se que a aproximação utilizada é uma aproximação de ***primeira ordem***, querendo significar que à medida que $\Delta x \to 0$ o erro de truncatura se torna proporcional a Δx.

O estudo do problema da ***estabilidade*** é, em geral, muito mais complexo. No caso dos problemas de Mecânica existe muitas vezes um teste simples que permite verificar, à posteriori, a estabilidade de um método a partir do cálculo da energia mecânica total do sistema em cada instante. A conservação de energia é uma ***condição suficiente*** de estabilidade (mas não de precisão ou de convergência).

Estamos agora em condições de apresentar o ***método de Euler***. O método consiste simplesmente em, a partir da posição inicial x_0 (notar, mais uma vez, que a variável independente tanto pode ser o espaço como o tempo) e para cada posição na rede $x = x_0 + \Delta x$, calcular o valor de cada uma das variáveis dependentes (e.g. temperatura, velocidades e posições) a partir da fórmula:

$$y_{k+1,j} = y_{k,j} + y'(x_{k,j}, y_{k,j})\Delta x \qquad (9\text{-}10)$$

em que k identifica o ponto da rede e j a equação do sistema.

9.3.1 Comentários

O método de Euler tem, em geral, uma precisão insuficiente. Na prática, é sempre preferível recorrer a métodos de 2ª ou mesmo de 4ª ordem (e.g. métodos de Runge-Kutta), cuja utilização é ainda relativamente simples. A introdução do método de Euler neste curso deve-se, pois, essencialmente a razões pedagógicas, dado o carácter altamente intuitivo do seu desenvolvimento.

Quando o método de Euler se utiliza para resolver equações de ordem superior à primeira, é possível utilizar os valores do primeiro integral no extremo superior do intervalo de integração para calcular o segundo integral. Muitas vezes esse procedimento é necessário para evitar instabilidade. Na integração das equações da Mecânica utilizar-se-á essa variante do método de Euler. Concretamente, far-se-á:

$$\begin{cases} \vec{v}_{k+1} = \vec{v}_k + \vec{a}(\vec{r}_k)\Delta t \\ \vec{r}_{k+1} = \vec{r}_k + \vec{v}_{k+1}\Delta t \end{cases} \qquad (9\text{-}11)$$

9.4 Aplicações

Exemplo 9-1 Arrefecimento de um corpo

Vamos considerar uma chávena com café cuja temperatura inicial é de 80ºC e que se encontra numa sala em que a temperatura do ar é de T_a=20ºC. Admite-se que a temperatura do ar é constante. É conveniente neste caso utilizar como unidades ºC (para a temperatura) e minutos (para o tempo). Nessas unidades podemos utilizar o valor de $\kappa = 0.1$ (min^{-1}). Sugere-se o estudo do comportamento da solução para outros valores dos parâmetros. Faça aparecer na mesma figura a solução analítica (9-2). Estime o valor do passo de tempo máximo para produzir uma simulação aceitável.

```
"""
Lei do arrefecimento de Newton
Resolve a equação DT/dt=-alpha*(T-Tar), pelo método de Euler
"""
import numpy as np
import matplotlib.pyplot as plt
Tar=20 #Temperatura do ar
Tinicial=80 #Temperatura inicial do corpo
alpha=0.1 # Coeficiente de transferência de calor
dt=0.1 #Passo de tempo
kc=0
cores=['red','blue','cyan']
plt.close('all')

for dt in[0.1,2,5]:
    tempo=np.arange(0.,120.,dt)
    n=len(tempo)
    T=np.zeros(tempo.shape)
    k=0 #Indice dos vectores
    T[k]=Tinicial
    for k in range(1,n):
        T[k]=T[k-1]-alpha*(T[k-1]-Tar)*dt
    plt.plot(tempo,T,color=cores[kc])
```

```
    plt.xlabel('Time (min)')
    plt.ylabel(r'$T ^{o} C$') # Notar símbolo
    plt.text(60,60-kc*5,r'$Euler  \Delta t=%6.3f$' %(dt) ,color=cores[kc])

    tempoAn=np.arange(0.,120.,5)
    Tan=Tar+(Tinicial-Tar)*np.exp(-alpha*tempoAn)
    plt.scatter(tempoAn,Tan,color='red')
    kc=kc+1
```

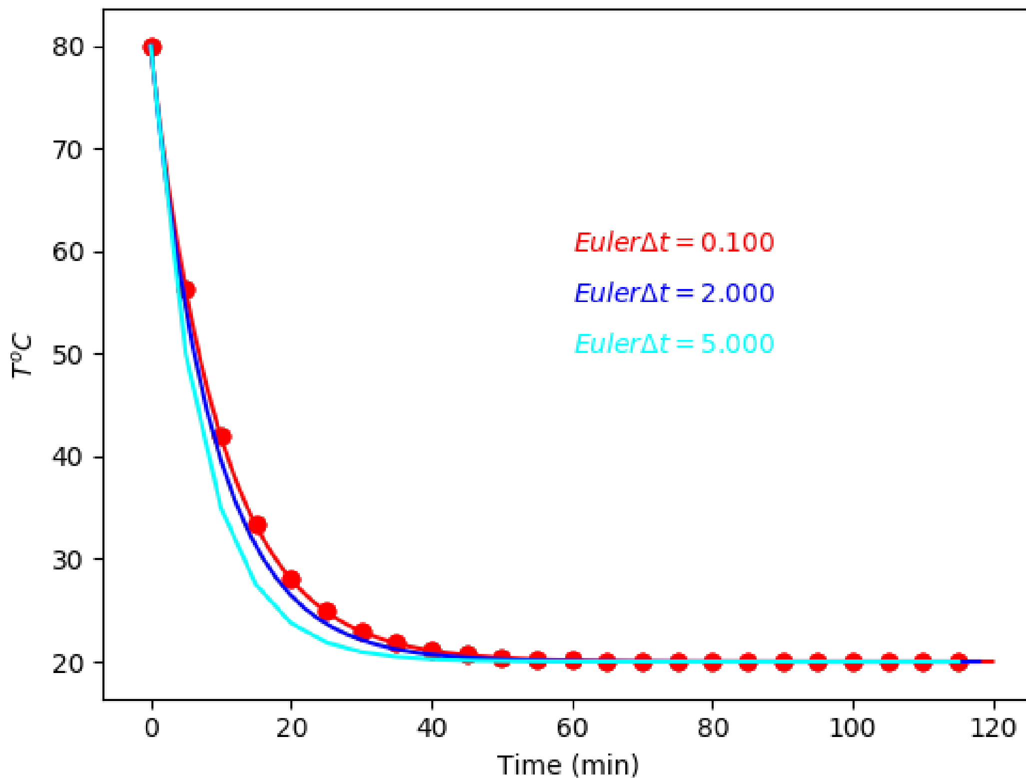

Figura 9-1 Lei de Newton do arrefecimento para diferentes passos de tempo (em s). Símbolos indicam a solução analítica cada 5 minutos.

Exemplo 9-2 Movimento planetário

As equações relevantes para este caso foram apresentados na secção 9.2. Dado que neste problema as variáveis tomam valores muito elevados no Sistema Internacional de unidades, com consequentes problemas numéricos, é preferível utilizar como unidades de base o ano (unidade de tempo) e a "unidade astronómica" (unidade de distância), em que 1 u.a. é a distância média da Terra ao Sol ≈150 000 000 km. Sugestões: (a) varie a velocidade inicial de modo a obter uma trajetória com maior excentricidade (caso dos cometas); (b) escolha uma velocidade inicial correspondente a uma órbita aberta (não elíptica); (c) tente aumentar o passo de tempo e veja as consequências.

```
"""
Movimento planetário utilizando o método de Euler explícito
x0,y0: posição inicial em ua
vx0,vy0: velocidade inicial em ua/ano
dt: passo de tempo em ano
n: número de passos de tempo
Output X,Y,T posição do planeta (X,Y), ao longo do tempo (T)
```

```
E Energia mecânica por unidade de massa do planeta
"""
def traject(x0,y0,vx0,vy0,dt,n):
    #Constantes
    G=6.67E-11;
    M=1.99e30;
    ua=1.5e11;
    ano=365.25*24*3600;
    GM=G*M*ano**2*ua**(-3);
    #Condições inciais
    x=x0;
    y=y0;
    vx=vx0;
    vy=vy0;
    t=np.arange(0,n*dt,dt) #vetor de tempos
    n=len(t)
    X=np.array(t)
    Y=np.array(t)
    E=np.array(t)
    X[0]=x0
    Y[0]=y0
    E[0]=(vx**2+vy**2)/2.-GM/np.sqrt(x**2+y**2)
    for k in range(1,n):
        aa=-GM*(x**2+y**2)**(-3./2.);
        ax=aa*x
        ay=aa*y
        vx=vx+ax*dt
        vy=vy+ay*dt
        x=x+vx*dt
        y=y+vy*dt
        X[k]=x
        Y[k]=y
        E[k]=(vx**2+vy**2)/2.-GM/np.sqrt(x**2+y**2)
    return X,Y,t,E

import numpy as np
import matplotlib.pyplot as plt
from math import pi
plt.close('all')
kp=1 #indice de subplot
for vy0 in[1.2*pi,2*pi,2.5*pi]:
    x0=1;y0=0;vx0=0;dt=0.001;n=5/(dt); #5 anos de simulação
    plt.subplot(2,3,kp)
    plt.scatter(x0,y0,color='black') #plot da posição inicial do planeta
    plt.quiver(x0,y0,vx0,vy0,scale=40)
    plt.text(x0+0.01,y0+0.02,'%8.3f ' % (vy0))
    plt.scatter(0.,0.,color='red') #plot da posição do Sol
    plt.grid() #traça grelha
    plt.axis('equal') #escala isotrópica
    [X,Y,T,E]=traject(x0,y0,vx0,vy0,dt,n);
    plt.plot(X,Y)
    plt.subplot(2,3,3+kp)
    plt.plot(T,E)
    plt.xlabel('Tempo (anos)')
    if kp==1:
        plt.ylabel('E (J/kg)')
    kp=kp+1
```

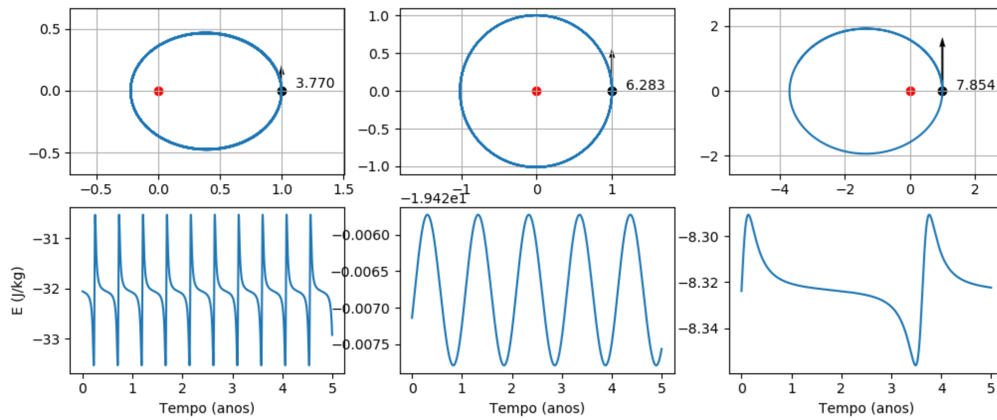

Figura 9-2 Movimento planetário: (linha superior) trajetória; (linha inferior) energia mecânica. Diferentes valores da velocidade inicial (em ua/ano). $\Delta t = 0.001\ ano$. Notar a notação utilizada na escala da Energia (específica) no caso central (órbita circular).

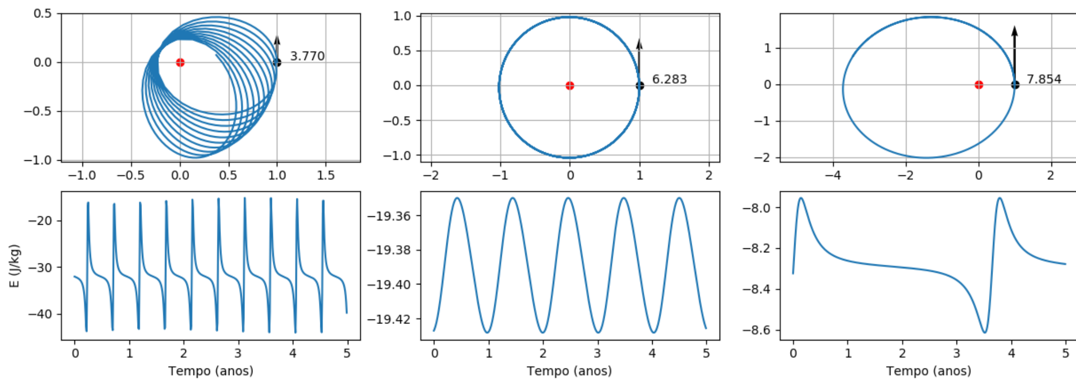

Figura 9-3 Tal como a Figura 9-2, $\Delta t = 0.01\ ano$. Notar o aumento do erro relativo devido ao aumento do passo de tempo, especialmente no exemplo da esquerda.

Problema 9-1

Introdução no exercício do movimento planetário um cálculo do momento angular, variável conservativa.

Exemplo 9-3 Movimento balístico

Vamos considerar o caso do movimento tridimensional de um corpo (ponto material) num campo gravítico uniforme, com posição inicial [x=0,y=0,z=0] e velocidade inicial [u=0,v=0,w=320]m/s, à latitude de 40N, considerando o efeito da rotação da Terra (aceleração de Coriolis), até ele voltar

a atingir o solo. Desprezamos a resistência do ar e a curvatura da Terra. Apesar de o corpo ser disparado na vertical, o efeito de Coriolis vai impor um desvio, cujo valor vamos calcular.

As equações do movimento do corpo são simplesmente:

$$\frac{d\vec{v}}{dt} = \vec{g} - 2\vec{\Omega} \times \vec{v} \tag{9-12}$$

onde, utilizando um sistema de coordenadas cartesiano $[x, y, z]$ *de versores* $[\vec{\imath}, \vec{\jmath}, \vec{k}]$*, com x na direção leste, y na direção norte e z na direção vertical,*

$$\vec{g} = -g\vec{k}$$

é a aceleração gravítica e

$$\vec{\Omega} = \Omega \cos\varphi\, \vec{\imath} + \Omega \sin\varphi \vec{k}$$

é o pseudo-vetor de rotação da Terra, de módulo igual à velocidade angular do planeta $\Omega \approx 7.292 \times 10^{-5} s^{-1}$*, colinear com o eixo da rotação, dirigido para o Norte celeste.* φ *é a latitude.*

A equação (9-12) pode ser escrita como um sistema de 3 equações escalares, uma para cada componente, com $\vec{v} \equiv [u, v, w]$*:*

$$\begin{cases} \dfrac{du}{dt} = -fv + f'w \\ \dfrac{dv}{dt} = fu \\ \dfrac{dw}{dt} = -g - f'u \end{cases} \tag{9-13}$$

onde $f = 2\Omega \sin\varphi$ *e* $f' = 2\Omega \cos\varphi$*. Aos 40N* $f \approx 0.-937 \times 10^{-5} s^{-1}, f' \approx 1.12 \times 10^{-4} s^{-1}$*.*

Dadas as condições iniciais, o corpo iniciará uma trajetória puramente ascendente, mas sofrerá uma pequena aceleração (proporcional a w) na direção x, e uma pequeníssima aceleração

(proporcional a u) na direção y. Vamos utilizar o método de Euler, testando o seu comportamento para diferentes valores do passo de tempo de integração $\Delta t = [0.1,1,10]s$.

Para o efeito, vamos começar por notar, sendo o objetivo o cálculo da trajetória [x(t),y(t),z(t)], precisamos de introduzir equações para a posição, o que duplica o número de equações:

$$\begin{cases} \frac{du}{dt} = -fv + f'w \\ \frac{dv}{dt} = fu \\ \frac{dw}{dt} = -g - f'u \\ \frac{dx}{dt} = u \\ \frac{dy}{dt} = v \\ \frac{dz}{dt} = w \end{cases} \tag{9-14}$$

Vamos procurar uma solução utilizando o método de Euler, com os 3 valores prescritos para Δt.

```
"""
Movimento balístico calculado pelo método de Euler
"""
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from math import pi
plt.close('all')
g=9.8065
Omega=7.292e-5;phi=40;
f=2*Omega*np.sin(phi*pi/180);fP=2*Omega*np.cos(phi*pi/180)
x0=0;y0=0;z0=0 #posição inicial
u0=0;v0=0;w0=320 #velocidade inicial
timeInt=1000 #tempo máximo de integração
fig=plt.figure(1) #cria figura 1
ax=fig.add_subplot(111, projection='3d') #cria figura 3D
cores=['blue','green','red']
kc=-1 #contador para os gráficos
for dt in[0.1,1,10]:
    kc=kc+1
    n=round(timeInt/dt) #número de passos de tempo
    tempo=np.arange(0.,timeInt,dt)
    X=np.zeros((n),dtype='float')*float('nan') #inicia com "not a number"
    Y=np.copy(X)
    Z=np.copy(X)
    U=np.copy(X)
    V=np.copy(X)
    W=np.copy(X)
    X[0]=x0
    Y[0]=y0
    Z[0]=z0
    u=u0;v=v0;w=w0;
    uP=u;vP=v;wP=w
    for kt in range(1,n):
        uP=u+f*v*dt-fP*w*dt
        vP=v-f*u*dt
```

```
        wP=w+fP*u*dt-g*dt
        X[kt]=X[kt-1]+u*dt
        Y[kt]=Y[kt-1]+v*dt
        Z[kt]=Z[kt-1]+w*dt
        u=uP
        v=vP
        w=wP
        if Z[kt]<=0: #atingiu o solo!
            Z[kt]=float('nan')
            break
    fig=plt.figure(1) #figura trajetoria 3D
    ax.scatter(x0,y0,z0,color='red')
    ax.plot(xs=X,ys=Y,zs=Z,color=cores[kc],label=r'$\Delta t=%6.2f s
      $'%(dt)) #notar color e label para a legenda
    plt.legend()
    plt.xlabel('x');plt.ylabel('y')
    plt.title(r'$Trajetória @\varphi=40N  $')
    Axes3D.view_init(ax,elev=30,azim=-145)
```

O código anterior dá origem à Figura 9-4, onde é claro que a solução converge quando $\Delta t \rightarrow 0$, dando no entanto origem a soluções com erro relativo significativo para certos valores desse parâmetro.

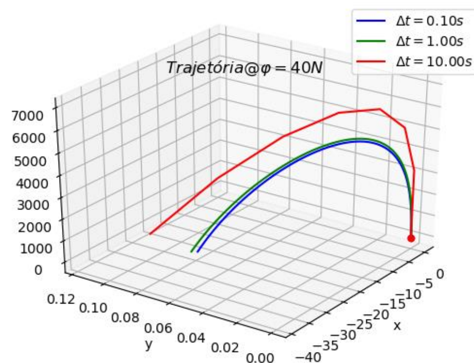

Figura 9-4 Trajetória calculada na solução do Exemplo 9-3, pelo método de Euler, para 3 valores de Δt. Círculo vermelho indica posição inicial. O corpo é desviado pata Leste e para Norte.

Antes de tentar melhorar a solução, o que é possível com pequenas alterações do método, vamos notar que aprendemos a representar gráficos em 3 dimensões com funções associadas ao objeto **`Axes3D`**. *Aprendemos também a fazer novas anotações (***`legend`***).*

O fragmento de código escrito a azul no exemplo anterior pode ser rescrito utilizando uma técnica semelhante à integração pelo método do ponto médio. De facto, por exemplo, a solução da equação

$$\frac{dx}{dt} = u$$

utilizou, no método de Euler, a discretização

$$x^{n+1} = x^n + u^n \Delta t$$

onde x^n, u^n representam a posição e a velocidade no passo de tempo n. Seria mais preciso, por satisfazer o ***teorema da média****, escrever:*

$$x^{n+1} = x^n + u^{n+1/2} \Delta t$$

onde $u^{n+1/2}$ representa a velocidade no passo de tempo intermédio. Considerando as 6 equações acopladas, o cálculo dos passos intermédios pode ser feito iterativamente, substituindo o fragmento de código azul por (nota: se se fizer **`aP=0`** *obtém-se o método de Euler):*

```
    aP=0.5;a=1-aP
    for kt in range(1,n):
        uP=u+f*v*dt-fP*w*dt
        uu=aP*uP+a*u
        vP=v-f*uu*dt
        wP=w+fP*uu*dt-g*dt
        vv=(aP*vP+a*v)
        ww=(aP*wP+a*w)
        uP=u+f*vv*dt-fP*ww*dt #segunda iteração para u
        uu=aP*uP+a*u
        X[kt]=X[kt-1]+uu*dt
        Y[kt]=Y[kt-1]+vv*dt
        Z[kt]=Z[kt-1]+ww*dt
        u=uP
        v=vP
        w=wP
```

dando origem à Figura 9-5. Nesta solução não se conseguem distinguir as soluções como $\Delta t = 0.1{,}1s$ e a solução com $\Delta t = 10s$ apresenta um erro relativo diminuto.

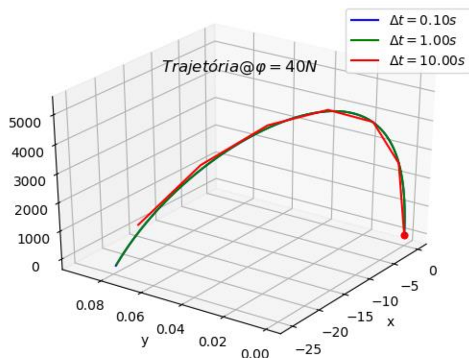

Figura 9-5 Trajetória calculada na solução do Exemplo 9-3, pelo método de Euler modificado, para 3 valores de Δt. Círculo vermelho indica posição inicial. Detalhes no texto.

Problema 9-2

Modifique o problema anterior, acrescentando o cálculo da energia mecânica (potencial + cinética) para os 3 valores de Δt, e para `ap=0.5,0`

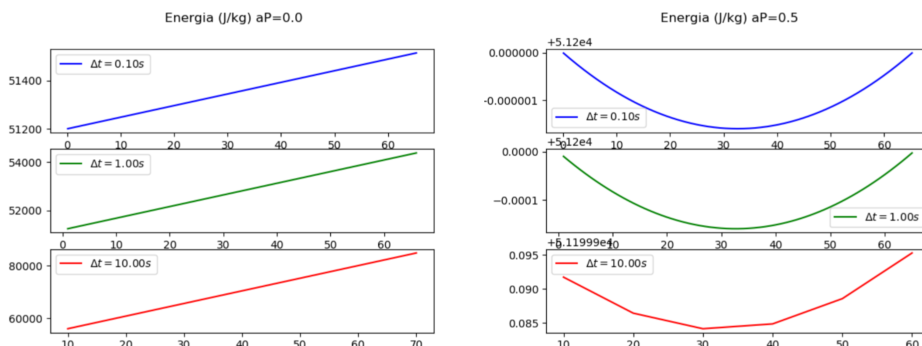

Figura 9-6 Evolução da energia mecânica específica (calculada para $g = const$ como $E_M = gh + 1/2(u^2 + v^2 + w^2)$). Com o método de Euler há violação clara da conservação (a energia cresce, em 70s, 2/500 com $\Delta t = 0.1s$, e 25% com $\Delta t = 10s$). Com o método modificado existe uma flutuação lenta com uma amplitude relativa inferior a 10^{-10} ($\Delta t = 0.1s$) ou 10^{-7} ($\Delta t = 10s$).

Problema 9-3

Modifique o problema anterior, introduzindo o efeito da resistência do ar, admitindo que a força de resistência é proporcional ao quadrado da velocidade em cada instante e tem a direção oposta à velocidade.

Problema 9-4

Determine o movimento de um oscilador harmónico governado pela lei $F = -kx$, i.e.:

$$\frac{dv}{dt} = \frac{F}{m} = -\frac{k}{m}x$$

com $\frac{k}{m} = 0.1\ s^{-2}, x(t = 0) = 0, v(t = 0) = -5\ ms^{-1}$. Integre as equações do movimento desse corpo, calculando em cada instante a sua energia total.

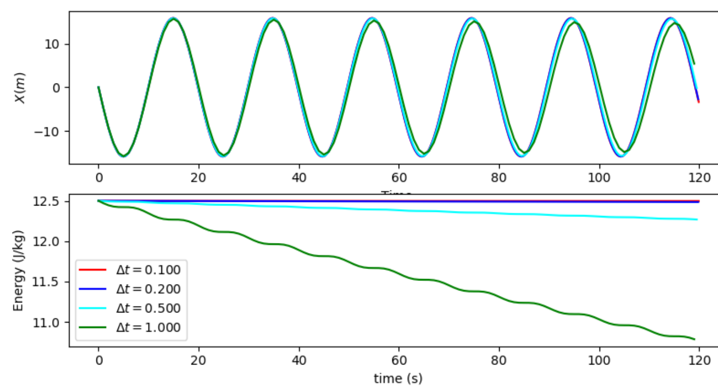

Figura 9-7 Evolução de um oscilador harmónico: soluções numéricas para diferentes passos de tempo de integração, utilizando o método iterativo descrito no Exemplo 9-3. Notar que só as soluções de menor passo de tempo ($\Delta t = 0.1, 0.2\ s$) conservam (em boa aproximação) energia mecânica.

Exemplo 9-4 Movimento balístico à escala planetária

10 Equações às derivadas parciais independentes do tempo

10.1 Identificação do Problema e Formulação Matemática

10.1.1 Equação de Poisson do potencial de uma distribuição contínua de carga

Recorda-se da Electrostática que o potencial gerado por uma distribuição contínua de carga elétrica numa placa satisfaz à equação:

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = -\frac{\rho}{\varepsilon_0} \tag{10-1}$$

em que V é o potencial eléctrico, $\rho(x, y)$ a densidade volúmica de carga e ε_0 a permitividade eléctrica do meio.

A equação anterior é uma Equação de Poisson. A solução da equação de Poisson é o que geralmente se designa como um problema de condições fronteira, na medida em que a solução é completamente determinada pelas condições impostas na fronteira do domínio do problema. A equação de Poisson surge em muitos outros problemas de Física, em particular na determinação do potencial elétrico (ou gravítico) na presença de uma distribuição de cargas (ou de massas) geradoras e em problemas de Mecânica de Fluidos.

Recorda-se que nos problemas considerados no âmbito da apresentação do método de Euler a solução dependia essencialmente das condições iniciais (valor da função ou da função e suas derivadas no instante inicial).

10.1.2 Equação de Laplace aplicada ao equilíbrio térmico de uma placa

Na ausência de fontes ou sumidouros de calor, a condução de calor num corpo satisfaz a equação:

$$\frac{\partial T}{\partial t} = \lambda \left[\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2} \right] \tag{10-2}$$

Se a temperatura na fronteira do corpo for imposta e se esperar tempo suficiente a distribuição de temperatura no seu interior pode obter-se resolvendo a equação anterior com o primeiro membro igual a zero. A equação designa-se então por equação de Laplace. Se o corpo em estudo tiver uma geometria bidimensional (caso de uma placa fina) a equação de Laplace escreve-se simplesmente:

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \tag{10-3}$$

É claro que a Equação de Laplace é um caso particular da Equação de Poisson (com $\rho = 0$), pelo que se vai considerar unicamente o caso mais geral para o desenvolvimento de um algoritmo. A equação de Laplace surge em muitos outros problemas de Física, em particular na determinação do potencial elétrico (ou gravítico) numa zona do espaço exterior às cargas (ou às massas) geradoras e em problemas de Mecânica de Fluidos.

10.2 Formulação geral

Vai-se então considerar a solução da equação:

$$\frac{\partial^2\phi}{\partial x^2}+\frac{\partial^2\phi}{\partial y^2}=f(x,y) \tag{10-4}$$

dadas condições fronteira, por exemplo, na forma:

$$\begin{cases} \phi(0,y)=\phi_1(y) \\ \phi(L_x,y)=\phi_2(y) \\ \phi(x,0)=\phi_3(x) \\ \phi(x,L_y)=\phi_4(x) \end{cases} \tag{10-5}$$

Valores concretos para as condições fronteira e para o termo independente (f), serão considerados antes da apresentação dos programas.

10.3 Métodos numéricos (Relaxação)

Contrariamente ao caso dos problemas de valores iniciais considerados num capítulo anterior (em que se apresentou o método de Euler), não é possível obter a solução da equação de Poisson partindo do valor da solução num ponto da fronteira e calculando a solução ponto a ponto, localmente. Para satisfazer as condições fronteira em todos os pontos limite é necessário obter a solução para os pontos interiores de forma global.

Existem vários métodos numéricos (iterativos e diretos) de solução da Equação de Poisson. No âmbito deste curso vai-se considerar unicamente um método iterativo, chamado método da Relaxação (sobrerrelaxação simultânea) que tem a vantagem de ser muito simples e convergir sempre, sendo no entanto relativamente lento. Para problemas em que o tempo seja um fator determinante devem ser considerados métodos diretos.O método da relaxação baseia-se (tal como a generalidade dos outros métodos) na representação das derivadas parciais presentes na equação (10-4) por diferenças finitas centradas (numa aproximação de segunda ordem):

$$\begin{cases} \frac{\partial^2\phi}{\partial x^2}\approx\frac{\phi_{i-1,j}-2\phi_{i,j}+\phi_{i+1,j}}{\Delta x^2} \\ \frac{\partial^2\phi}{\partial y^2}\approx\frac{\phi_{i,j-1}-2\phi_{i,j}+\phi_{i,j+1}}{\Delta y^2} \end{cases} \tag{10-6}$$

com:

$$\begin{cases} \phi_{k,j}=\phi(x_k,y_j) \\ x_i=(i-1)\Delta x; i=1,\dots,M \\ y_j=(j-1)\Delta y; j=1,\dots,M \end{cases} \tag{10-7}$$

A dedução das relações anteriores é muito simples e baseia-se no desenvolvimento da função considerada em série de Taylor. Assim, na vizinhança do ponto x tem-se (notar que os sinais dos termos ímpares dependem do sinal de Δx):

$$\phi(x\pm\Delta x)=\phi(x)\pm\frac{\partial\phi}{\partial x}\Delta x+\frac{1}{2}\frac{\partial^2\phi}{\partial x^2}\Delta x^2\pm\frac{1}{3!}\frac{\partial^3\phi}{\partial x^3}\Delta x^3+\cdots \tag{10-8}$$

somando as duas equações (8) e resolvendo para a segunda derivada obtém-se:

$$\frac{\partial^2\phi}{\partial x^2} = \frac{\phi_{i-1,j} - 2\phi_{i,j} + \phi_{i+1,j}}{\Delta x^2} - \frac{\partial^4\phi}{\partial x^4}\Delta x^2 - \cdots \quad (10\text{-}9)$$

pelo que as aproximações (6) consistem em desprezar termos de segunda ordem (proporcionais a Δx^2) e superiores.

Para simplificar, vai considerar-se o caso $\Delta x = \Delta y = \Delta$. Fazendo a substituição na equação (10-4) obtém-se um sistema de $(M-2)\times(N-2)$ equações:

$$\begin{gathered}\phi_{i-1,j} + \phi_{i+1,j} + \phi_{i,j-1} + \phi_{i.j+1} - 4\phi_{i,j} = \Delta^2 f_{i,j} \\ \{i = 2, \dots, M-1; j = 2, \dots, N-1\}\end{gathered} \quad (10\text{-}10)$$

Note-se que o sistema (10-10) se refere unicamente aos pontos interiores, devendo a solução na fronteira ser imposta.

O sistema (10-10) é um sistema de equações lineares mas que envolve em geral um número muito grande de equações pelo que a sua solução direta apresenta dificuldades. O método da relaxação consiste na solução iterativa desse sistema de equações. Tal como em outros métodos iterativos é necessário escolher uma primeira aproximação, para a qual se escolhe normalmente um valor constante. Como o método converge sempre, a escolha da solução "inicial" só interfere no número de iterações necessário. Assim, na iteração n pode escrever-se:

$$\phi^n_{i-1,j} + \phi^n_{i+1,j} + \phi^n_{i,j-1} + \phi^n_{i,j+1} - 4\phi^n_{i,j} - \Delta^2 f_{i,j} = R_{i,j} \quad (10\text{-}11)$$

em que $R_{i,j}$, definido por (10-11), é uma medida do erro em cada ponto (***resíduo***) após a última iteração. O objetivo da iteração é, naturalmente, fazer tender o resíduo para zero. Um método de reduzir o erro é calcular um novo valor no ponto (i,j) que garanta o anulamento do resíduo correspondente, isto é:

$$\phi^{n+1}_{i,j} = \phi^n_{i,j} + \frac{R_{i,j}}{4} \quad (10\text{-}12)$$

Deve notar-se no entanto que quando se anula o resíduo num ponto, se vai perturbar necessariamente a solução em pontos adjacentes. Mas há convergência.

Há duas pequenas modificações que podem melhorar o método, acelerando a convergência. A primeira consiste em utilizar para o cálculo dos resíduos os valores da função à medida que vão sendo conhecidos. Neste caso, alguns dos termos do primeiro membro da equação (11) pertencem à iteração $n+1$ enquanto outros ainda pertencem à iteração n. Ao método da relaxação modificado desta maneira chama-se ***relaxação simultânea***. Em termos de programação, a relaxação simultânea é ainda mais simples do que a relaxação sucessiva, porque não é necessário guardar os valores da função em várias iterações.

A segunda modificação, designada por ***sobrerrelaxação***, consiste em introduzir um peso no cálculo da correção (10-12). Nesse caso escreve-se:

$$\phi^{n+1}_{i,j} = \phi^n_{i,j} + \beta\frac{R_{i,j}}{4},\ 1 \le \beta \le 2 \quad (10\text{-}13)$$

A justificação da vantagem da sobrerrelaxação não é trivial, mas pode ser facilmente verificada nos programas apresentados posteriormente. No caso da Equação de Poisson o valor ótimo do parâmetro de sobrerrelaxação é dado por:

$$\beta_{opt} = 2 - \pi\sqrt{2}\left(\frac{1}{M^2} + \frac{1}{N^2}\right)^{1/2} \quad (10\text{-}14)$$

para uma rede de M por N pontos.

10.4 Aplicações

Exemplo 10-1 Potencial de uma distribuição bidimensional contínua de cargas (Equação de Poisson)

Considere um plano de 1×1 m. Nesse plano encontram-se 4 cargas $q_k = [10^{-9}, -10^{-9}, -1.5 \times 10^{-9}, -2 \times 10^{-9}]$ *Coulomb, nos pontos* $[(0.4,0.5), (0.6,0.5), (0.3,0.2), (0.5,0.8)]m$*. Utilizando uma malha de* $\Delta x = \Delta y = 1cm$*, calcule a função densidade de carga* ρ *dividindo a carga presente em cada elemento de malha pela sua área (Nota:* $\rho = 0$ *em todos os elementos menos em quatro, centrados em cada uma das cargas). Resolva a equação de Poisson, impondo a condição* $V = 0$ *nos pontos de fronteira. Faça a representação gráfica da distribuição do potencial elétrico.*

Nota: é de esperar, naturalmente, que a solução seja bastante diferente da solução analítica para uma distribuição de cargas pontuais no interior do retângulo elementar onde está localizada a carga e na sua vizinhança imediata. Se se aumentar o número de pontos da rede a solução converge.

Solução:

```
"""
poisson2D.py
Potencial eléctrico devido a uma distribuição pontual de cargas
Resolve a equação de Poisson Lap(V)=f
Método: sobrerrelaxação simultanea
    Input: f(M,N) forçamento
           X(N),Y(M) coordenadas X e Y
           maxiter número máximo de iteracoes
           maxres residuo máximo aceitavel (relativo ao máximo de V)
           beta parametro de sobrerrelaxação
    Output: V(M,N)
            iter iterações realizadas
            resid resíduo relativo final
            beta parametro de sobrerrelaxação (se não for dado como input
            entre 1 e 2)
"""
def poisson(f,V0,X,Y,maxiter,maxres,beta0):

    [M,N]=f.shape; #determina a dimensão das matrizes
    iter=0
    V=np.ones((M,N))*V0 #inicializa a matriz do potencial
    resid=2*maxres #garante a primeira iteração
    if (beta0<1) or (beta0>2):
        beta=2-np.pi*np.sqrt(2.)*np.sqrt(1./M**2+1./N**2) #parametro de
  sobrerrelaxação 1<=alpha<=2
    else:
        beta=beta0
```

```
    if f.shape!=X.shape or X.shape!=Y.shape: # verifica consistencia
        print('Error in matrix size')
        return V,iter,resid,beta
    dx=X[2,1]-X[1,1];dy=Y[1,2]-Y[1,1] #Admite-se espaçamento regular
    if(dx!=dy):
        print('Error in dx,dy')
        return  V,iter,resid,beta
    delta=dx
    while(resid>maxres) and (iter<maxiter): #iterações
        iter=iter+1
        resid=0
        vmax=0
        for i in range(1,M-1): #=2:M-1
            for j in range(1,N-1): #2:N-1
                R=V[i,j-1]+V[i,j+1]+V[i-1,j]+V[i+1,j]-4*V[i,j]-
      delta**2*f[i,j]
                V[i,j]=V[i,j]+beta*R/4;
                resid=max(resid,abs(R));
        vmax=np.max(np.abs(V))
        resid=resid/vmax #residuo relativo
    return V,iter,resid,beta

import numpy as np
import matplotlib.pyplot as plt
plt.close('all') #fecha figuras anteriores
plt.figure(figsize=(6,5))
Lx=1.
Ly=1.
M=51
N=51
delta=Lx/(M-1)
X=np.zeros((M,N));Y=np.zeros((M,N));ro=np.zeros((M,N))
for i in range(M):
    for j in range(N):
        X[i,j]=i*delta
        Y[i,j]=j*delta
ncargas=4 # define as cargas pontuais e sua localização
xis=[0.4,0.6,0.3,0.5];yps=[0.5,0.5,0.2,0.8];q=[1e-9,-1e-9,-1.5e-9,-2e-9];
#calcula a matriz densidade de carga
for carga in range(ncargas):
    i=round(xis[carga]/delta)+1 #; %i,j devem ser inteiros
    j=round(yps[carga]/delta)+1
    ro[i,j]=q[carga]/delta**2
#define a função forçadora na equação de Poisson (segundo membro)
eps0=8.8544e-12
f=-ro/eps0
maxiter=5000 # número máximo de iterações
maxres=1.e-8 # erro relativo
V=0*np.ones((M,N)) # %potencial na fronteira
V0=0
beta0=0 # %beta é calculado em poisson
[V,niter,res,beta]=poisson(f,V0,X,Y,maxiter,maxres,beta0)
map=plt.contourf(X,Y,V,cmap='jet') # %gráfico de isolinhas
axes = plt.gca()
axes.set_xlim([0.,Lx])
axes.set_ylim([0.,Ly])
plt.colorbar(map)
plt.axis('equal')
plt.show()
plt.title(r'$\nabla^{2} V=\rho/\epsilon, iter=%6i, Resid=%10.3e,
      \beta=%8.4f  $' % (niter,res,beta))
```

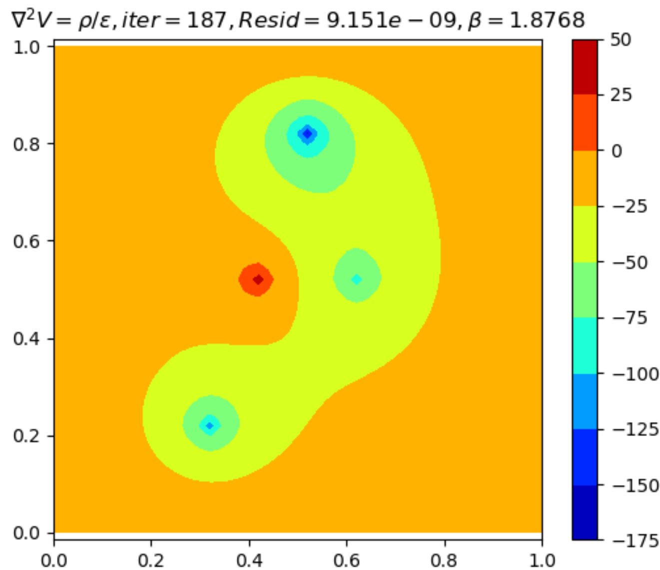

Figura 10-1 Potencial elétrico de uma distribuição pontual de cargas (M=N=51).

Exemplo 10-2 Distribuição de Temperatura numa placa em regime estacionário (Equação de Laplace)

Considere uma placa quadrada de 1x1 m cuja temperatura é mantida constante e igual a 273K em três da arestas ($x = 0, L_x; y = 0$). Na quarta aresta ($y = L_y$) a temperatura tem uma distribuição dada por $T(x, L_y) = T_0 + \theta \sin\left(\frac{x\pi}{L_x}\right)$. Resolva a equação (10-3) para calcular a distribuição de temperatura na placa quando ela atinge o equilíbrio.

Solução:

Utiliza-se a rotina **poisson** *desenvolvida no Exemplo 10-1, impondo o forçamento $f(x, y) = 0$ e a condição fronteira proposta. Na definição das condições fronteira fazem-se operações vetoriais sobre linhas e colunas da matriz T_0, usando o símbolo "**:**". Essas mesmas operações permitem a realização de um gráfico (x,y) a partir de matrizes 2D, utilizado para produzir a Figura 10-3.*

```
"""
laplace2D.py
Distribuição estacionária de temperatura numa placa
Resolve Lap(T)=0, com T imposto na fronteira.
"""
import numpy as np
import matplotlib.pyplot as plt

from poisson2D import poisson #utiliza-se código do exemplo poisson2D

plt.close('all')# fecha figuras anteriores
plt.figure()

Lx=1.
Ly=1.
M=51
N=51
delta=Lx/(M-1)
X=np.zeros((M,N));Y=np.zeros((M,N))
```

```
for i in range(M):
    for j in range(N):
        X[i,j]=i*delta
        Y[i,j]=j*delta

maxiter=5000; #número máximo de iterações
maxres=1.e-8; #erro relativo
Tf=273;theta=5;
f=np.zeros((M,N))
T0=np.zeros((M,N));T0[0,:]=Tf;T0[M-1,:]=Tf;T0[:,0]=Tf;
T0[:,N-1]=Tf+theta*np.sin(X[:,N-1]*np.pi/Lx); #temperatura na fronteira
beta0=0; #%beta é calculado em poisson
[T,niter,res,beta]=poisson(f,T0,X,Y,maxiter,maxres,beta0);

map=plt.contourf(X,Y,T,cmap='jet');#%gráfico de isolinhas
cb=plt.colorbar(map)
cb.set_label('Temperatura (K)')
plt.title(r'$\nabla^{2} T=0, iter=%6i, Resid=%10.3e, \beta=%8.4f  $' %
      (niter,res,beta))
plt.xlabel('m')
plt.ylabel('m')
plt.figure(figsize=(6,2))
plt.plot(X[:,N-1],T0[:,N-1])
plt.title('Temperatura em y=Ly')
```

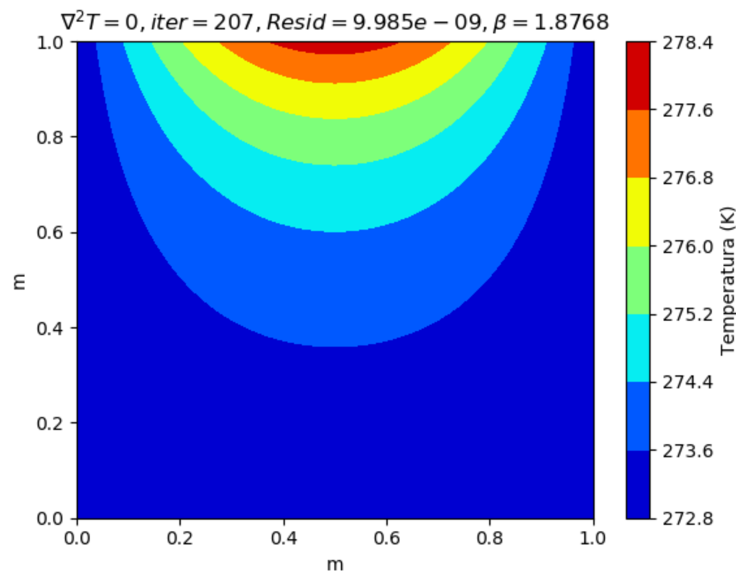

Figura 10-2 Distribuição de temperatura numa placa em situação estacionária ($M = N = 51$).

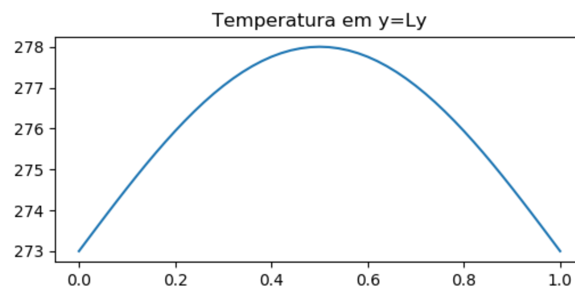

Figura 10-3 Temperatura em y=1, no exemplo da Figura 10-2.

10.5 Comentários e revisão

Os exemplos apresentados na secção anterior utilizaram um conjunto de ferramentas comuns. A função **poisson** tem uma possível função “laplace” como caso particular, desde que o termo independente seja posto a “0”. Nas linhas iniciais da função poisson, incluiu-se uma verificação da consistência das matrizes de *input*, no que se refere à sua dimensão e à utilização do mesmo intervalo de amostragem ($\Delta x = \Delta y$). A segunda condição poderia não ser imposta, mas exigiria alterações do código de solução.

No processo iterativo da relaxação utilizou-se uma estrutura **while** com dupla condição. A condição de convergência baseia-se numa estimativa do resíduo relativo. A condição do número de iterações destina-se a evitar um ciclo infinito, caso a condição de convergência não possa ser cumprida. Este duplo critério é geralmente uma boa ideia para ciclos deste tipo.

A visualização do resultado em 2 dimensões foi realizada com a função **contourf** (filled contours), com a escala de cores representada com a função **colorbar**. Introduziram-se várias anotações, usando, nalguns casos, notação matemática.