1 计算物理基础

1.1 章节概述

如果是新手可能需要查看一些相关的视频。本章介绍计算基础知识，从一些工具如可视化和矩阵操作入手来熟悉 Python 生态系统。接下来对数字表示以及使用浮点数的限制和后果进行讨论。然后回顾微分、积分和随机数生成的一些基本数值方法。最后讨论如何解常/偏微分方程。每章有许多习题供练手，大多数问题都需要一些可视化。本书倾向于使用 Matplotlib 或 VPython（以前称为 Visual）。虽然在程序中包含可视化会使代码变长，但提高了调试速度。也可以将结果输出到数据文件，然后使用 gnuplot 或 Grace 等程序可视化。

1.2 Python 生态系统

本书给出了习题解法的代码。Python 是免费的、鲁棒的、可移植的和通用的，它是最简单的解释性语言。它包含了高级的内置数据类型，使矩阵操作和图形操作变得容易，并有很多免费强大的包和库，非常适合数值分析和符号运算。 为学习 Python，我们推荐在线教程 [Ptut(14), Pguide(14), Plearn(14)], 和 Langtangen 的书 [Langtangen(08), Langtangen(09)], 还有一些 Python 参考资料 [Beazley(09)]。

本书中用到这些库：

1. Matplotlib (Mathematics Plotting Library) http://matplotlib.org
2. NumPy (Numerical Python) http://www.numpy.org/
3. SciPy (Scientific Python) http://scipy.org
4. SymPy (Symbolic Python) http://sympy.org
5. VPython (Python with Visual package) http://vpython.org/

如果不想一个一个地安装包,也可以用 Python 包集合，推荐

1. Anaconda https://store.continuum.io/cshop/anaconda/

2. Enthought Canopy https://www.enthought.com/products/canopy/

3. Spyder (in Anaconda) https://pythonhosted.org/spyder/

1.2.1 Python 可视化工具

vpython 可视化包对创建 3-D 实体、2-D 绘图和动画特别有用。可以通过导入模块 `vpython` 使用。

```
from vpython import *
y1 = gcurve(color = blue)
```

图 1.1 中是由代码清单 1.1 `EasyVisualVP.py` 程序生成的两个图。注意 vpython 的绘图过程首先是创建一个 plot 对象，然后将点添加到对象中。相比之下，Matplotlib 先创建了一个点向量，然后 plot 了整个向量。

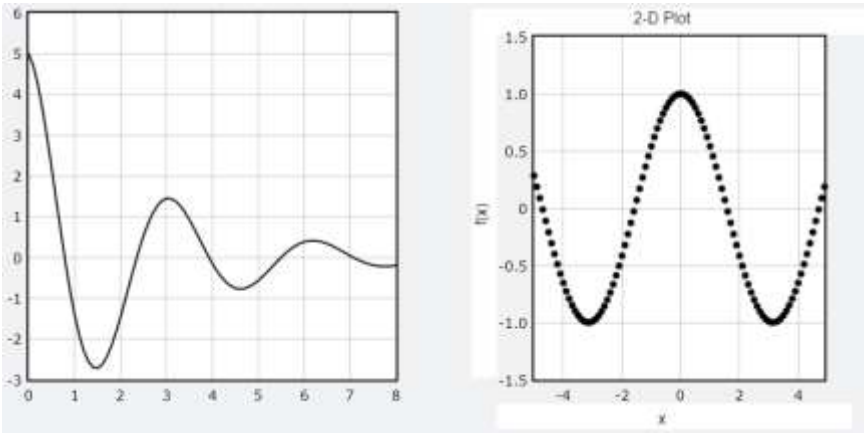

图 1.1 清单 1.1 vpython 绘图实例

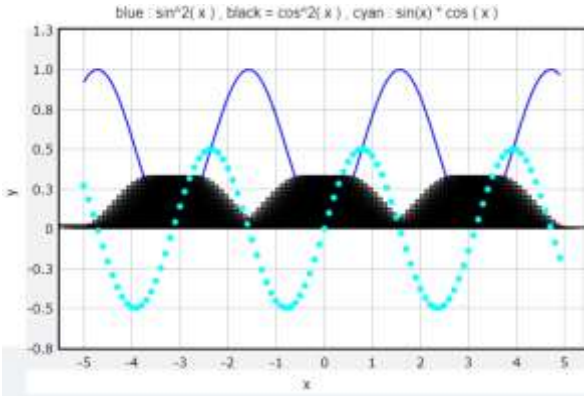

图 1.2 清单 1.2 3GraphVP.py 用 vpython 画出 2D 图形

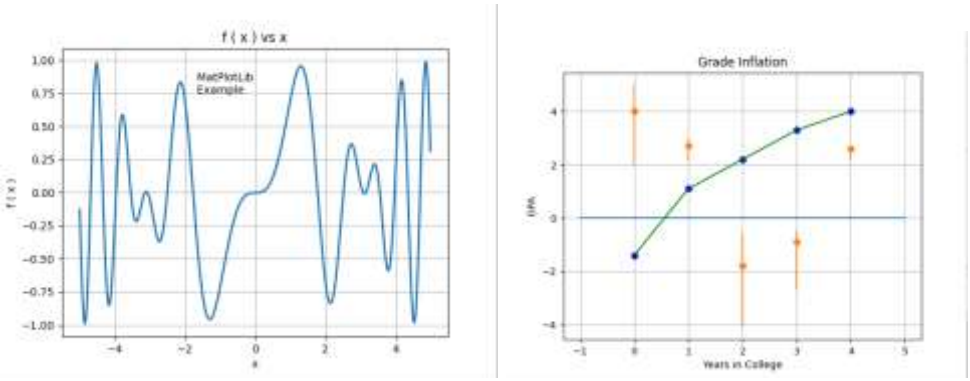

图 1.3 清单 1.3 EasyMatPlot.py 用 matplotlib 画出 2D 图形

清单 1.2 中的程序 `3GraphVP.py` 在同一个图中放置了几个图。垂直条用 `gvbars`、点用 `gdots`，曲线用 `gcurve`。

Matplotlib 是一个强大的绘图包，用于 2-D 、 3-D 图形和各种数据图。 它使用了 NumPy 和 LAPACK 的复杂数据类型[Anderson et al.(113)] 和类似于 MATLAB 的命令。将希望绘制的 x 和 y 值放入一维数组（向量）中，然后用调用 plot 绘制这些向量。 在 `EasyMatPlot.py` 中，如清单 1.3 所示，我们导入 Matplotlib 作为 pylab 库：

```
from pylab import *
```

然后我们计算并输入关于 x,y 的向量：

```
x = arange(Xmin, Xmax, DelX) # x array in range + increment
y = -sin(x)*cos(x) # y array as function of x array
```

NumPy 的 arange 方法构造一个数组，覆盖 Xmax 和 Xmin 之间的“a range”，步长为 DelX。 因为范围的限制是浮点数，所以每个 xi 都是浮点数。并且因为 x 是一个数组，所以 y = -sin(x)*cos(x) 也是一个数组。事实上绘图是用破折号“-”来表示一条线，`lw=2` 来设置它的宽度。结果如图 1.3 所示，带有所需的标签和标题。用 `show()` 命令在的桌面上生成图形。

在清单 1.4 中代码 `GradesMatplot.py` 给出图 1.3 的右侧图像。通过重复 plot 命令在同一个图中绘制多个数据集。这里导入 Matplotlib 的 pyplot 和 NumPy，因为需要 `array` 命令。一条水平线是通过绘制一个所有值都等于零的数组来得到的，还可以画上下误差线以及网格线。

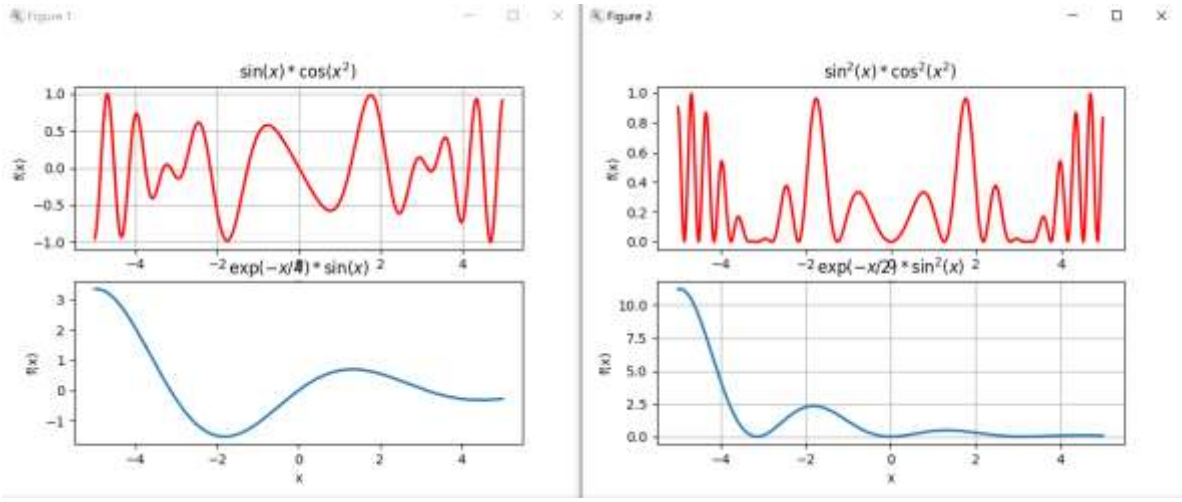

图 1.4 清单 1.4 用 MatPlot2fig.py 绘制多个图，每个图多个子图

如果把几条曲线都画在一个 plot 中，几个 plot 画在一个 figure 中，数据通常会更清楚。Matplotlib 允许使用 `plot` 和 `subplot` 命令来执行此操作。例如，在代码清单 1.5 和图 1.4 `MatPlot2figs.py` 中，一个 figure 中有两个 subplot，然后输出两个不同的 figures。这里的关键是重复 subplot 命令：

```
figure(1) # 1st figure
subplot(2,1,1) # 1st subplot, 2 rows, 1 column
```

```
subplot(2,1,2) # 2nd subplot
```

如果你想可视化偶极势

$$V(x,y) = Bx + \frac{Cx}{(x^2+y^2)^{\frac{3}{2}}}$$

需要一个 3-D 图，其中山高$z = V(x,y)$，而 x，y 轴定义山下的平面。三个维度的感官是通过着色、视差、鼠标旋转等技巧获得的。

图 1.5 从清单 1.6 中的程序 `Simple3Dplot.py` 中生成的是三维的线框图。关键是利用 meshgrid 方法创建网格矩阵，然后构造 Z(x, y) 曲面。

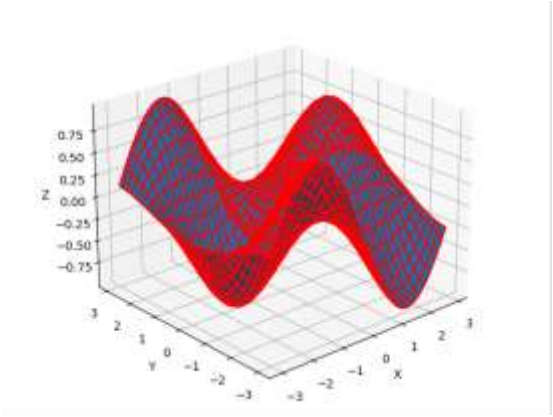

图 1.5 清单 1.6 Simple3Dplot.py 用 Matplotlib 绘制三维图

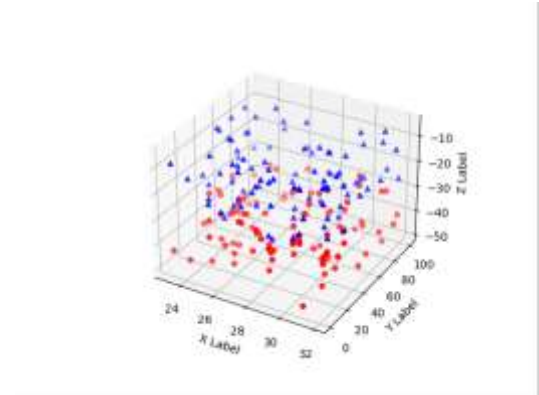

图 1.6 清单 1.8 用 matplotlib 生成的三维散点图

散点图是一种在 3-D 中可视化单个点(x_i, y_j, z_k)的有用方法。如清单 1.8 Scatter3dPlot.py 生成的图 1.6。

用 vpython 创建动画本质上是只是一遍又一遍地制作相同的二维图，每个图都略有不同时间。

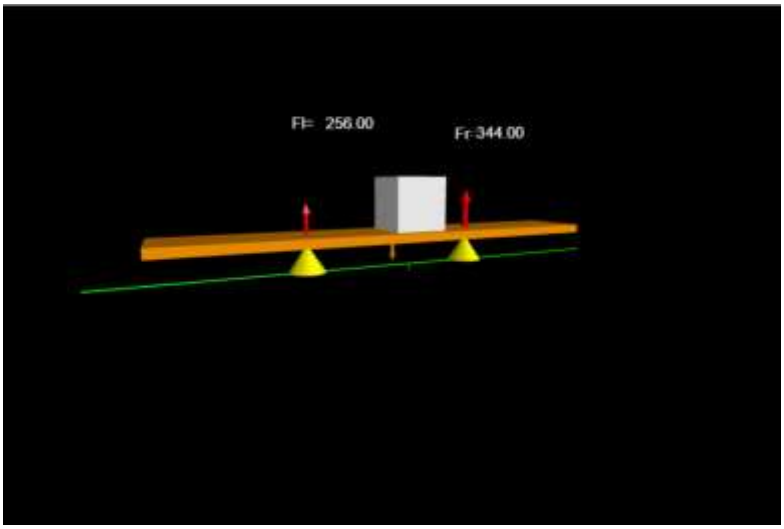

图 1.7 清单 1.10 用 vpython 生成的动画截图

1. 如图 1.7 所示，长度 L = 10 m，重量 W = 400 N 的梁靠在两个相距 d = 2 m 的支架上。 一个箱子重量 Wb = 800 N，最初在左端支架的上方，以一定速度无摩擦地向右滑动 v=7m/s。

a. 编写一个程序，计算当盒子沿着梁滑动时，左右两端支架对梁施加的力。

b. 将程序创建成一个动画，实时显示当滑块沿着梁滑动时的位置和梁所受的力。 在代码清单 1.10 SlidingBox.py，展示了动画截取的截图。可以针对手头的问题去修改。

c. 将双支撑问题扩展到：梁的右侧边缘下有第三个支撑。

2. 如图 1.8 左边所示，一个 2 kg 的物体接着两条 5 米的绳子，每个绳子的一端由学生拿着。 最初绳子是垂直的，但随着两个学生以 4 m/s 的恒定速度向左向右移动而分开。忽略摩擦。

(a) 每根绳索的初始张力是多少？

(b) 计算学生分开时每根绳索的张力。

(c) 学生能不能让两条绳子都水平？

(d) 扩展的程序使其实时显示绳子随学生分开时的张力。图 1.8 中展示了这段代码的动画中的截图。 修改代码为适合的问题。

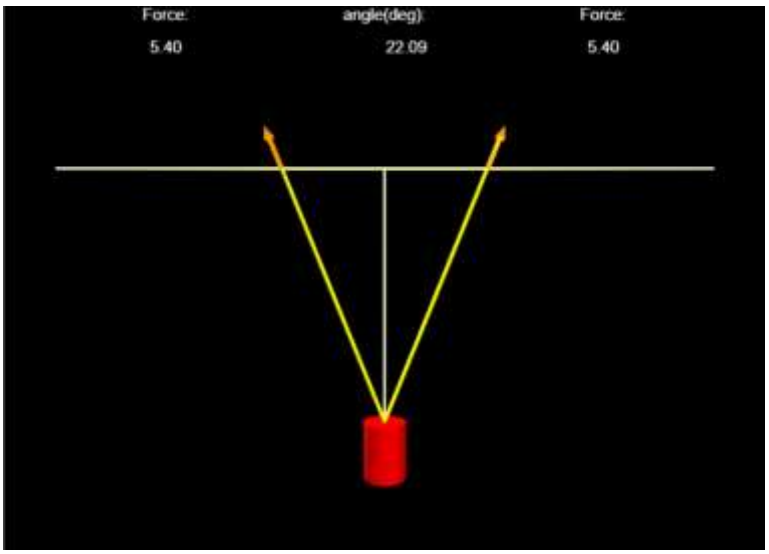

图 1.8 清单 1.9 用 vpython 生成的动画截图

1.2.2 Python 矩阵工具

一次处理多个数字是计算的主要优势，计算机语言具有用于此目的的抽象数据类型。 list 是 Python 中以一定顺序保存的数字或对象序列的内置数据类型。 array 是 NumPy 包中可用的更高级别的数据类型，可以像向量一样操作。对于 list，用带逗号分隔符的方括号[1, 2, 3]表示，方括号也用于指示列表项的索引：

```
>>> L = [1, 2, 3] # Create list
>>> L[0] # Print element 0
1
>>> L # Print entire list
[1, 2, 3]
>>> L[0]= 5 # Change element 0
>>> len(L) # Length of list
3
```

NumPy 的 arrays 可以将 lists 转换为 arrays, 使之可以像向量一样操作：

```
>>> vector1 = array([1, 2, 3, 4, 5]) # Fill array wi list
>>> print("vector1 =", vector1)
vector1 = [1 2 3 4 5]
>>> vector2 = vector1 + vector1 # Add 2 vectors
>>> print("vector2=", vector2)
vector2= [ 2 4 6 8 10]
>>> matrix1 = array(([0,1],[1,3]) # An array of arrays
>>> print(matrix1)
[[0 1]
[1 3]]
>>> print (matrix1 * matrix1) # Matrix multiply
[[0 1]
[1 9]]
```

描述 arrays 时的维度 ndim 最高可以达到 32，矩阵的形状用 shape 描述：

```
>>> import numpy as np
>>> np.arange(12) # List 12 ints
array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
>>> np.arange(12).reshape((3,4)) # Reshape to 3x4
array([[ 0, 1, 2, 3],
[ 4, 5, 6, 7],
[ 8, 9, 10, 11]])
>>> a = np.arange(12).reshape((3,4))
>>> a.shape
(3L, 4L)
>>> a.ndim # Dimension?
2
>>> a.size # Number of elements?
12
```

两个数组的矩阵乘积用 dot 函数，星号 * 用于逐个元素乘积：

```
>>> matrix1= array( [[0,1], [1,3]])
>>> matrix1
array([[0, 1],
[1, 3]])
>>> print ( dot(matrix1,matrix1) ) # Matrix or dot product
[[ 1 3]
[ 3 10]]
>>> print (matrix1 * matrix1) # Element-by-element product
[[0 1]
[1 9]]
```

为了速度和可靠性，我们推荐使用完善的库而不是自己编写矩阵运算。 虽然 NumPy 的 array 对象与数学矩阵不同，但 LinearAlgebra 包可将二维数组视为数学矩阵。 考虑线性的标准解方程

$$A\boldsymbol{x} = \boldsymbol{b}$$

例如：

```
>>> from numpy import *
>>> from numpy.linalg import *
>>> A = array( [ [1,2,3], [22,32,42], [55,66,100] ] ) # Array of arrays
>>> print ("A =", A)
A = [[ 1 2 3]
[ 22 32 42]
[ 55 66 100]]
```

用 Numpy 的 solve 函数求解：

```
>>> b = array([1,2,3])
>>> from numpy.linalg import solve
>>> x = solve(A, b) # Finds solution
>>> print ("x =", x)
x = [ -1.4057971 -0.1884058 0.92753623] # The solution
>>> print ("Residual =", dot(A, x) - b) # LHS-RHS
Residual = [4.44089210e-16 0.00000000e+00 -3.55271368e-15]
```

另一种直接但不是非常有效的方式是求 $\boldsymbol{x} = A^{-1}\boldsymbol{b}$：

```
>>> from numpy.linalg import inv
>>> dot(inv(A), A) # Test inverse
array([[ 1.00000000e+00, -1.33226763e-15, -1.77635684e-15],
[ 8.88178420e-16, 1.00000000e+00, 0.00000000e+00],
[ -4.44089210e-16, 4.44089210e-16, 1.00000000e+00]])
>>> print ("x =", multiply(inv(A), b))
x = [-1.4057971 -0.1884058 0.92753623] # Solution
>>> print ("Residual =", dot(A, x) - b)
Residual = [ 4.44089210e-16 0.00000000e+00 -3.55271368e-15]
```

对于本征值问题 $I\omega = \lambda\omega$，可以用 eig 方法：

```
>>> from numpy import *
>>> from numpy.linalg import eig
>>> I = array( [[2./3,-1./4], [-1./4,2./3]] )
>>> print('I =\n', I)
I = [[ 0.66666667 -0.25 ]
[-0.25 0.66666667]]
>>> Es, evectors = eig(A) # Solve eigenvalue problem
>>> print("Eigenvalues =", Es, "\n Eigenvector Matrix =\n", evectors)
Eigenvalues = [ 0.91666667 0.41666667]
```

```
Eigenvector Matrix = [[ 0.70710678 0.70710678]
[-0.70710678 0.70710678]]
>>> Vec = array([ evectors[0, 0], evectors[1, 0] ] )
>>> LHS = dot(I, Vec)
>>> RHS = Es[0]*Vec
>>> print("LHS - RHS =", LHS-RHS) # Test for 0
LHS - RHS = [ 1.11022302e-16 -1.11022302e-16]
```

1. 求矩阵的逆

$$A = \begin{pmatrix} 4 & -2 & 1 \\ 3 & 6 & -4 \\ 2 & 1 & 8 \end{pmatrix}$$

a. 检查 $AA^{-1} = A^{-1}A = I$

b. 注意小数位数，这会给一些计算精度的信息。

c. 确定数值结果和解析结果的小数位数一致：

$$A^{-1} = \frac{1}{263}\begin{pmatrix} 52 & 17 & 2 \\ -32 & 30 & 19 \\ -9 & -8 & 30 \end{pmatrix}$$

2. 继续考虑矩阵 A，对三个不同的 $\boldsymbol{b}$，求解三个线性方程组：

$$\boldsymbol{b}_1 = \begin{pmatrix} 12 \\ -25 \\ 32 \end{pmatrix},\ \boldsymbol{b}_2 = \begin{pmatrix} 4 \\ -10 \\ 22 \end{pmatrix},\ \boldsymbol{b}_3 = \begin{pmatrix} 20 \\ -30 \\ 40 \end{pmatrix}$$

3. 考虑矩阵 $A = \begin{pmatrix} \alpha & \beta \\ -\beta & \alpha \end{pmatrix}$，对任意给定的 α, β，给出本征值和本征向量：

$$\boldsymbol{x}_{1,2} = \begin{pmatrix} 1 \\ \mp i \end{pmatrix},\ \lambda_{1,2} = \alpha \mp i\beta$$

1.2.3 Python 代数工具

符号运算软件代表了一种补充但功能强大的方法物理计算工具 [Napolitano(18)]。 Python 发行版通常包含符号运算包 Sage 和 SymPy，它们之间有很大的不同。 Sage 与 Maple 和

MATHEMATICA 属于同一类，超过本书讨论范围。 相比之下，SymPy 包的运行方式非常像 Python shell 中的任何其他 Python 包。 例如，这里我们使用 Python 的交互式 shell 从 SymPy 导入方法，然后进行一些分析：

```
>>> from sympy import *
>>> x, y = symbols('x y')
>>> y = diff(tan(x),x); y
tan^2(x) + 1
>>> y = diff(5*x**4 + 7*x**2, x, 1); y # dy/dx 1 optional
20 x^3 + 14 x
>>> y = diff(5*x**4+7*x**2, x, 2); y # d^2y/dx^2
2 (30 x^2 + 7)
```

Symbols 命令将变量 x 和 y 声明为代数变量，而 diff 命令取关于第二个参数的导数（第三个参数是导数的顺序）。 以下是一些扩展：

```
>>> from sympy import *
>>> x, y = symbols('x y')
>>> z = (x + y)**8; z
(x + y)^8
>>> expand(z)
x^8 + 8 x^7 y + 28 x^6 y^2 + 56 x^5 y^3 + 70 x^4 y^4
+ 56 x^3 y^5 + 28 x^2 y^6 + 8 x y^7 + y^8
```

Sympy 可以对函数做某个点处的级数展开：

```
>>> sin(x).series(x, 0) # Sin x series
x - x^3/6 + x^5/120 + \mathcal{O}(x^6)$
>>> sin(x).series(x,10) # sin x about x=10
sin(10) + x cos(10) - x^2 sin(10)/2 - x^3 cos(10)/6
+ x^4 sin(10)/24 + x^5 cos(10)/120 +O(x^6)
>>> z = 1/cos(x); z # Division, not an inverse
$1/\cos(x)$
>>> z.series(x, 0) # Expand 1/cos x about 0
1 + x^2/2 + 5 x^4/24 + O(x^6)
```

计算机代数系统的一个经典难题是生成的答案可能是正确的，但形式不够简单。 SymPy 的简化函数有 `simple`、`tellfactor`、`collect`、`cancel` 和 `apart`：

```
>>> factor(x**2 -1)
(x - 1) (x + 1) # Well done
>>> factor(x**3 - x**2 + x - 1)
(x - 1) (x^2 + 1)
>>> simplify((x**3 + x**2 - x - 1)/(x**2 + 2*x + 1))
x - 1
>>> factor(x**3+3*x**2*y+3*x*y**2+y**3)
(x + y)^3 # Much better!
>>> simplify(1 + tan(x)**2)
cos(x)^{(-2)}
```

1.3 处理浮点数

在科学计算必须考虑到数字是用有限的内存来表示的。 标准计算中整数使用表示 fixde-point 记法，其他数字则用浮点数或科学记数法。 在 Python 中通常处理 32 位整数和 64 位浮点数（在其他语言中称为双精度）。 双精度大约有 16 位小数范围内的精度和幅度：

$$4.9\times10^{-324} \leq \text{ double precision } \leq 1.8\times10^{308}$$

如果一个数大于1.8×10^{308}，则出现 overflow 错误。如果小于4.9×10^{-324}，则是 underflow 错误。对于 overflow，结果依赖于机器的模式而不是一个数字（NAN）。 对于 underflow，结果通常为零。

由于 64 位浮点数仅存储 15-16 位小数，因此浮点计算通常是近似的。 例如，在计算

$$3+1.0\times10^{-16}=3$$

机器精度ϵ_m定义为可以添加到已存储的 1.0 中而不改变其值的最大正数：

$$1.0_c+\epsilon_m=1.0_c$$

其中下标 c 提醒我们这是 1 的计算机表示。所以，除了精确表示的 2 的幂次之外，应该假设所有浮点数的第 15 位有误差。

1.3.1 计算数字的不确定度

错误和不确定性是计算的组成部分。 一些错误是计算机由于计算机存储数字的精度有限或因为算法的近似性质而引起的错误。 算法错误可能源于用有限区间替换无穷小区间或用有限区间替换无限级数总和，例如，

$$\sin(x)=\sum_{n=1}^{\infty}\frac{(-1)^{n-1}x^{2n-1}}{(2n-1)!}\simeq\sum_{n=1}^{N}\frac{(-1)^{n-1}x^{2n-1}}{(2n-1)!}+\mathcal{E}(x,N)$$

其中，$\mathcal{E}(x,N)$是误差。一个合理的算法应该使$\mathcal{E}(x,N)$随 N 的增加而下降。

在涉及许多步骤的计算中，一种常见的不确定性是舍入误差。这是由于有限位数引起的累积不精确性引起的。 为简洁起见，假设一台计算机只保留小数点后四位。 然后它将 1/3 存储为 0.3333，将 2/3 存储为 0.6667，其中计算机已经“四舍五入”了 2/3 中的最后一位数字。 因此，即使是一个简单的减法也可能是错误的：

$$2\left(\frac{1}{3}\right)-\frac{2}{3}=0.6666-0.6667=-0.0001\neq 0$$

所以虽然结果很小，但它不是 0。即使有完整的 64 位精度，如果 a 计算被重复数百万或数十亿次，累积错误答案可能会变大。

实际计算往往是一种平衡。 如果计算中包括很多步骤，那么**近似误差**通常遵循幂次递减。进行 N 步后相对的舍入误差趋于随机累积，近似于$\sqrt{N}\,\epsilon_m$。因为总误差是舍入误差和近似误差的总和，最终不断增加的舍入误差将占主导地位。根据经验，当在计算中增加步数时，应该一个小数点一个小数点地注意答案的收敛或稳定性。 一旦你看到最后一位数字出现随机噪声，这时舍入误差开始占主导地位。

1. 写一个程序来确定计算机的下溢和上溢限制（利用因子 2）。这是一个示例伪代码

```
under = 1.
over = 1.
begin do N times
under = under/2.
over = over * 2.
write out: loop number, under, over
end do
```

a. 如果的初始选择不会导致下溢和溢出，请增加 N。

当 N=1024 时，over=inf。

b. 检查浮点数发生 overflow 和 underflow 的位置。

不会报错。

c. 检查最大和最大的负整数是什么。 你可以通过不断加减 1 来实现的。

2. 编写程序用因子 2 确定计算机系统的机器精度ϵ_m。示例伪代码是：

```
eps = 1.
begin do N times
eps = eps/2.
one = 1. + eps
end do
```

a. 通过实验确定浮点数的机器精度。

eps ≈ 2.220446049250313e-16

b. 通过实验确定复数的机器精度。

1.4 数值微分

虽然导数的数学定义很简单：

$$\frac{dy(t)}{dt} = \lim_{h\to 0}\frac{y(t+h)-y(t)}{h}$$

但不是一个好的算法。 随着 h 变小，分子在 0 和机器精度ϵ_m之间波动，分母趋近于零。 相反，使用一个 f(x + h) 关于 x 的泰勒级数展开式，h 保持小但有限。 在这里采用的前向差分算法：

$$\left.\frac{dy(t)}{dt}\right|_{FD} \simeq \frac{y(t+h)-y(t)}{h} + \mathcal{O}(h)$$

误差$\mathcal{O}(h)$可以用中心差分导数来消除：

$$\left.\frac{dy(t)}{dt}\right|_{\text{CD}} \frac{y(t+h/2)-y(t-h/2)}{h} + \mathcal{O}(h^2)$$

二阶导数的中心差分算法是通过使用一阶导数对应表达式的中心差分算法来获得：

$$\left.\frac{d^2y(t)}{dt^2}\right|_{CD} \simeq \frac{y'(t+h/2)-y'(t-h/2)}{h} \simeq \frac{y(t+h)+y(t-h)-2y(t)}{h^2}$$

1. 使用前向和中心差分算法求函数 cos(t)和e^t在 t = 0.1, 1., 和 100 的导数值。

a. 将导数及其相对误差$\mathcal{E}$关于 h 的函数显示出来。 减少步长 h 直到它等于机器精度$h \simeq \epsilon_m$。

b. 绘制$\log_{10}|\mathcal{E}|$关于$\log_{10} h$的图，检查小数位数是否获得与文中的估计一致。

2. 使用中心差分算法计算 cos(t)的二阶导数。

a. 测试四个周期，从$h \simeq \frac{\pi}{10}$开始并不断减小 h 直到达到机器精度。

1.5 数值积分

数学上，黎曼积分定义为：

$$\int_a^b f(x)\,dx = \lim_{h\to 0} \sum_{i=1}^{(b-a)/h} f(x_i) h$$

数值积分与之类似，但将积分近似为高度 f(x) 宽度（或权重）w_i 的矩形面积的有限和：

$$\int_a^b f(x)\,dx \simeq \sum_{i=1}^{N} f(x_i) w_i$$

上式是所有积分算法的标准形式：

函数 f(x)在区间 [a, b] 中的 N 个点处计算，并且对每个函数值$f_i \equiv f(x_i)$用w_i加权求和。 不一样的积分算法相当于选择点x_i和权重 w_i的不同方式。 如果可以自由选择积分点，那么我们建议的算法是 Gaussian quadrature。 如果这些点是均匀分布的，那么 Simpson's 规则也可以。

梯形和 Simpson's 积分规则都使用宽度为 h 的 N − 1 个框，这些框在整个积分区域[a, b]中均匀分布：

$$x_i = a + ih, \quad h = \frac{b-a}{N-1}, \quad i = 0, N-1$$

对于每个间隔，梯形规则假定梯形的宽度为 h 、高度$(f_i + f_{i+1})/2$，因此，将每个梯形的面积近似为$\frac{1}{2}hf_i + \frac{1}{2}hf_{i+1}$。将梯形规则应用于整个区域 [a, b]，我们添加了来自所有子区间的贡献：

$$\int_a^b f(x)\,dx \simeq \frac{h}{2}f_1 + hf_2 + hf_3 + \cdots + hf_{N-1} + \frac{h}{2}f_N$$

端点只计算一次，但内部点两次。 按照标准积分规则，有：

$$w_i = \left\{\frac{h}{2}, h, \ldots, h, \frac{h}{2}\right\}$$

在清单 1.15 的 `TrapMethods.py` 提供了一个简单的实现。Simpson's 规则也适用于宽度为 h 的均匀间隔点，高度由抛物线给出，该抛物线通过拟合三个相邻的被积函数值来得到。 得到近似值：

$$\int_a^b f(x)\,dx \simeq \frac{h}{3}f_1 + \frac{4h}{3}f_2 + \frac{2h}{3}f_3 + \frac{4h}{3}f_4 + \cdots + \frac{4h}{3}f_{N-1} + \frac{h}{3}f_N$$

根据积分规则，有

$$w_i = \left\{\frac{h}{3}, \frac{4h}{3}, \frac{2h}{3}, \frac{4h}{3}, \ldots, \frac{4h}{3}, \frac{h}{3}\right\}$$

因为拟合是用三个点的集合来完成的，所以要用 Simpson's 规则需要点数 N 为奇数。

一般来说，应该选择一个使用最少的积分点就能给出准确答案的积分规则。 对于梯形和 Simpson's 规则，误差变化为：

$$\mathcal{E}_t = O\left(\frac{[b-a]^3}{N^2}\right)\frac{d^2f}{dx^2}, \quad \mathcal{E}_s = O\left(\frac{[b-a]^5}{N^4}\right)\frac{d^4f}{dx^4}$$

除非被积函数的导数存在行为问题，否则 Simpson's 规则的收敛速度比梯形规则更快，误差更小。虽然看起来像可能只需要不断增加积分点的数量即可获得更好的精度，但是相对舍入误差会累积，经过 N 次积分点，增长为

$$\epsilon_{ro} \simeq \sqrt{N}\,\epsilon_m$$

其中，$\epsilon_m \simeq 10^{15}$ 是机器精度。所以尽管在算法中的误差可以任意小，但是由于舍入误差的叠加，总的误差将按 $\sqrt{N}$ 增大。

1.5.1 高斯求积法

Gauss 想出了一种选取 N 个点和 N 个权重的方法来对 [-1,1] 之间的函数进行积分。这 N 个点 x_i 必须是 N 阶勒让德多项式的 N 个零点，权重跟多项式的导数相关[LPB(15)]：

$$P_N(x_i) = 0, \quad w_i = \frac{2}{(1-x_i^2)\,[P'_N(x_i)]^2}$$

这里提供了一个程序来确定点数和权重。 如果积分范围是 [a,b] 而不是 [-1,+1]，则应缩放为

$$x'_i = \frac{b+a}{2} + \frac{b-a}{2}x_i, \quad w'_i = \frac{b-a}{2}w_i$$

一般来说，高斯积分法比梯形积分精度更高，跟 Simpson 规则需要的点数，值得推荐的。清单 1.16 中的高斯积分代码 `IntegGaussCall.py` 需要提供的点和权重以及精度 eps 值。整体精度通常通过增加点数来提高。点数和权重由 `Gaussxw` 方法生成。

1.5.2 蒙特卡洛积分

蒙特卡洛积分通常很简单，但不是特别有效。 这只是一个中值定理的直接应用：

$$I = \int_a^b dx f(x) = (b-a)\langle f \rangle$$

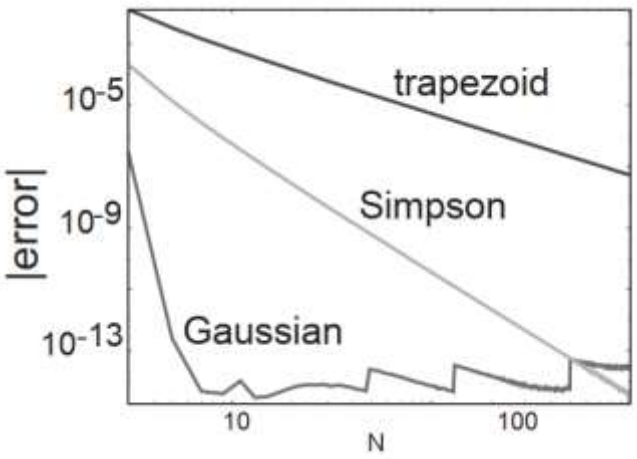

图 1.9 使用梯形规则，Simpson 规则以及高斯积分的相对误差与积分点数 N 的关系的双 log 图。

均值是通过对函数 f(x) 在积分区间内进行随机采样来确定的：

$$\langle f\rangle \simeq \frac{1}{N}\sum_{i=1}^{N} f\left(x_i\right) \quad \Rightarrow \quad \int_a^b dx f(x) \simeq (b-a)\frac{1}{N}\sum_{i=1}^{N} f\left(x_i\right)$$

由 f(x) 的 N 个样本得到的积分 I 的值的不确定性由标准偏差σ_I衡量。 假设σ_f是被积函数 f 在抽样过程的标准差，那么对于正态分布的随机数，有

$$\sigma_I \simeq \frac{1}{\sqrt{N}}\sigma_f$$

因此，对于大的 N，误差随$1/\sqrt{N}$减小。 图 1.20 展示了一个用 Monte Carlo 积分计算（清单 1.7 的 `PondMapPlot.py`）的散点图。

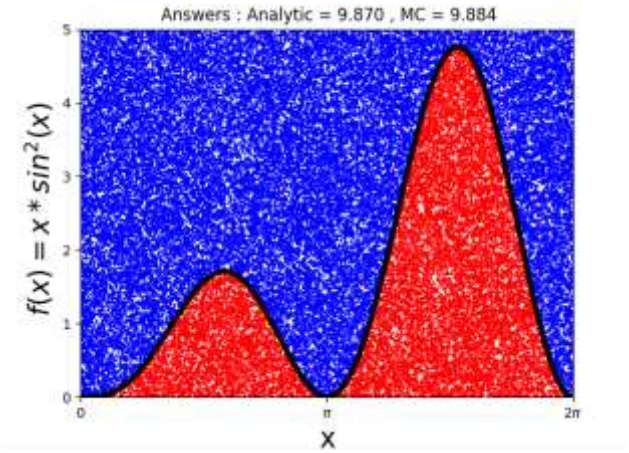

图 1.20 清单 1.7 用蒙特卡洛算法计算面积

在实际使用随机数计算积分之前，建议通过 §1.6.2 来确保随机数生成器工作正常。

为了确定池塘的面积，可以通过向空中扔石头来确定这种不规则图形的面积（生成随机 (x, y) 值），并计算进入池塘 Npond 的数量以及躺在地上的石头数量 Nbox。池塘的面积由下式给出简单的比例：

$$A_{\text{pond}} = \frac{N_{\text{pond}}}{N_{\text{pond}} + N_{\text{box}}} A_{\text{box}} .$$

1. 编写一个程序来对一个知道解析答案的函数的积分。分别用

a. 矩形规则

b. Simpson 规则

c. 高斯积分法

d. 蒙特卡罗

2. 对于每种情况计算相对误差 $\epsilon = |numerical - exact|/exact$，以及作相对误差关于 N 的对数图，如图 1.9 所示。可以观察到误差按幂次下降，表明特征算法误差随着 N 的增加而减小。请注意 plot 中的纵坐标是积分中精度的小数位数的负数。

3. 当舍入误差开始占优势时，表现为误差的随机涨落，算法应该停止收敛。 估计三个规则中精度的小数位数。

4. 使用蒙特卡洛算法计算 π（单位圆内的命中数/总命中数= π /盒子面积）。

5. 计算以下积分：

$a.\ \int_0^1 e^{\sqrt{x^3+5x}}dx$

$b.\ \int \frac{1}{x^2+2x+4}dx$

$c.\ \int_0^\infty e^{(-x^2)}dx$

1.6 随机数生成器

随机性或偶然性发生在物理学的不同领域。例如，量子力学和统计力学本质上是统计的，因此随机性作为统计学的关键假设之一。或者换个角度看，随机很早就从分子运动等过程观察到了。除了自然界的随机性，许多程序采用蒙特卡罗方法，其中包括模拟随机物理过程，例如热运动或放射性衰变，或数学计算如积分。在数学上，定义了一个序列 r1, r2, ...如果数字之间没有短期或长期相关性，则视为随机。这并不一定意味着该序列中所有数字发生的可能性相同（所谓的均匀性）。举个例子，0, 2, 4, 6, ...是均匀的，但可能不是随机的。如果 P(r) dr 是在区间 [r, r + dr] 中找到 r 的概率，那么均匀分布满足 P(r) = 常数。

计算机具有确定性，无法生成真正的随机数。 但可以生成伪随机数，内置的生成器通常非常擅长这个。 Python 中的 `random` 模块产生一个随机数序列，通过 `import random` 之后使用。 可以使用简单命令 `random.random()` 返回一个均匀分布在 [0.0, 1.0) 范围内的随机浮点数。线性同余或幂余法也常用于生成伪随机数序列：

$$r_{i+1} = (ar_i + c) \text{mod } M = \text{ remainder } \left(\frac{ar_i + c}{M}\right)$$

维基百科有一个常用选项表，例如，$m=2^{48}$, a = 25214903917, c =11。这里的 mod 是一个用于取模或求余的函数（Python 中的%符号），本质上是一种位移操作，返回了输入的最低有效部分，这个值依赖于舍入误差的随机性。

在 Python 中，使用 `random.random()`，Mersenne Twister 发生器。 要初始化一个随机序列，需要种一粒种子 (r0)，或者在 Python 中写 `random.seed(None)`，这将播种生成器与系统时间，重复执行的结果有所不同。 如果随机需要 [A, B] 范围内的数字，只需要缩放，例如：

$$x_i = A + (B - A)\, r_i, \quad 0 \le r_i \le 1, \Rightarrow A \le x_i \le B$$

1.6.1 随机数生成器测试

在开始完整计算之前，一个好的习惯是检查随机数生成器。 这里有一些方法：

- 查看 print 出来的数字是否不同且在所需的范围内。
- ri 与 i 的简单关系图无法证明随机性，但可以用来说明不随机以及超过范围。
- 绘制$(x_i, y_i) = (r_{2i}, r_{2i+1})$的 x-y 图。 如果的点具有明显的规律性，则该序列不是随机的。
- 通过评估 k 阶矩的分布来评估均匀性（而非随机性）

$$\langle x^k \rangle = \frac{1}{N}\sum_{i=1}^{N} x_i^k \simeq \int_0^1 dx x^k P(x) \simeq \frac{1}{k+1} + O\left(\frac{1}{\sqrt{N}}\right)$$

上式适用于连续的均匀分布。 如果结果与上式的偏差按$1/\sqrt{N}$ 变化，那么分布是随机的。

另一个简单的测试是确定序列的 k 阶相关性：

$$C(k) = \frac{1}{N}\sum_{i=1}^{N} x_i x_{i+k}, \simeq \int_0^1 dx \int_0^1 dy xy = \frac{1}{4}, \quad (k = 1, 2, ...)$$

如果随机数具有恒定的联合概率，那近似值将保持不变，将其设为 1。如果结果

与上式的偏差按$1/\sqrt{N}$变化，那么分布是随机的。

1. 分析为何推荐使用随机数生成器，请尝试使用线性同余方法比较。

a. 从一个不明智的选择开始：(a, c, M, r1) = (57, 1, 256, 10)。

b. 确定伪随机序列的周期。

c. 通过在这个不明智的选择$(x_i, y_i) = (r_{2i-1}, r_{2i}), i = 1, 2, ...$中观察点的团聚来分析相关性

d. 绘制 ri 与 i 的关系图。

e. 现在认真对待并使用其中一组维基百科给出的常数测试线性同余方法。

2. 类似上面讨论，使用各种方法测试计算机上的内置随机数生成器。

3. 比较使用内置随机数生成器与线性同余方法获得的散点图。

1.6.2 中心极限定理

在开始在计算中使用随机数之前，最好先验证生成的随机数满足所需的统计。 一种策略是测试生成的数字是否符合中心极限定理。这个定理告诉我们，当大量 N 个独立的随机变量相加，它们的总和趋于正态（高斯）分配：

$$\rho(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\langle x\rangle)^2/2\sigma^2}$$

其中$\langle x\rangle$是 x 的平均值，σ是标准偏差：

$$\langle x\rangle = \frac{1}{N}\sum_{i=1}^{N} x_i, \quad \sigma = \frac{1}{N}\sum_{i=1}^{N}(x - \langle x\rangle)^2$$

除了检查的随机数生成器之外，该定理还提供了一种简单的算法来生成具有高斯分布的随机数。

1. 生成 1000 个均匀分布的随机数并在[0,1] 内求和。

2. 创建一个列表，并将的总和作为列表中的第一个元素。创建许多个和并用它们填充列表，检查它们的平均值是否为 1/2，并且总和值满足均值为 1/2 的正态分布。

3. 对 N = 9999 重复步骤 1 和 2。

4. 计算列表中所有总和的平均值。

5. 通过上面的式子计算σ。

6. 绘制总和的直方图。在绘图选项使分布标准化（通过 density=Ture）。

7. 使用在上一步中计算的标准偏差在同一图上绘制正态分布。

8. 运行程序 100 次，注意每次运行直方图如何围绕正态分布曲线波动。

9. 绘制 100 次运行的平均值并将其与上上式进行比较。清单 1.11 中的程序 `CentralValue.py` 产生的结果如图 1.13 所示。

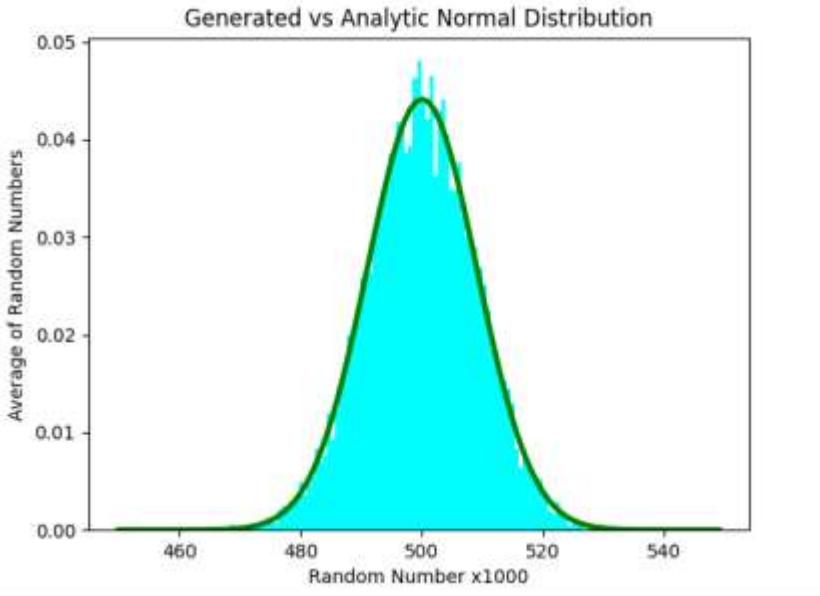

图 1.21 清单 1.11 均匀分布随机变量之和产生正态分布

1.7 常微分方程

本书中的许多问题需要常微分方程 ODE 的解。 用一两个简单的算法可以很容易地解决几乎所有的 ODE，而且由于相同的算法适用于非线性 ODE，不限于具有小位移的线性系统。 可以用于经典动力学，单个 ODE 或一组同时的 ODE。任意 N 阶的常微分方程可以表示为 N 个未知$y^{(0)}$–$y^{(N-1)}$的一阶 ODE。 未知数被组合成一个 N 维向量 y，其中

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

$$\boldsymbol{y} = \begin{pmatrix} y^{(0)}(t) \\ y^{(1)}(t) \\ \cdots \\ y^{(N-1)}(t) \end{pmatrix}, \quad \boldsymbol{f} = \begin{pmatrix} f^{(0)}(t,\boldsymbol{y}) \\ f^{(1)}(t,\boldsymbol{y}) \\ \cdots \\ f^{(N-1)}(t,\boldsymbol{y}) \end{pmatrix}$$

上式表明的规则是函数$\boldsymbol{f}(t,\boldsymbol{y})$可能不包含任何显式导数，尽管$y^{(i)}$的各个分量可能等于导数。

为了解这是如何工作的，从具有任意力的牛顿定律开始，然后将每一阶导数定义为一个新变量：

$$\frac{d^2x}{dt^2} = \frac{1}{m}F\left(t,x,\frac{dx}{dt}\right)$$

$$y^{(0)}(t) = x(t), \quad y^{(1)}(t) = \frac{dx}{dt} = \frac{dy^{(0)}(t)}{dt}$$

$$\Rightarrow \quad \frac{dy^{(0)}}{dt} = y^{(1)}(t) = f^{(0)}, \quad \frac{dy^{(1)}}{dt} = \frac{1}{m}F(t,y^{(0)},y^{(1)}) = f^{(1)}$$

1.7.1 欧拉-龙格库塔规则

如图 1.14 所示，通过从初始值 y(t = 0) ≡ y0 开始，对 ODE 进行数值求解，并使用导数函数 f(t, y)将 y0 在时间上向前推进一小步 h 到 y(t = h) ≡ y1。 然后算法只是不断重复，将 y 的新值视为新的初始条件进行下一步。 欧拉规则通过直接应用导数的前向差分算法来实现这一点：

$$\mathbf{y}_{n+1} \simeq \mathbf{y}_n + h\mathbf{f}(t_n,\mathbf{y}_n)$$

其中 yn ≡ y(tn) 是 tn 时刻 y 的值。除了用于初始化其他算法，欧拉的方法对于科学工作来说不够准确。与前向差分导数一样，欧拉规则中的误差为 O(h2)，相当于忽略加速度对弹丸位置的影响，但包含了弹丸的速度的影响。

与欧拉规则相反，四阶 Runge-Kutta 算法 rk4 具有被证明是鲁棒的并且能够进行工业强度的工作。这是推荐的求解 ODE 的方法。该算法基于微分方程：

$$\frac{dy}{dt} = f(t,y) \quad \Rightarrow \quad y(t) = \int f(t,y)\,dt \quad \Rightarrow \quad y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} f(t,y)\,dt.$$

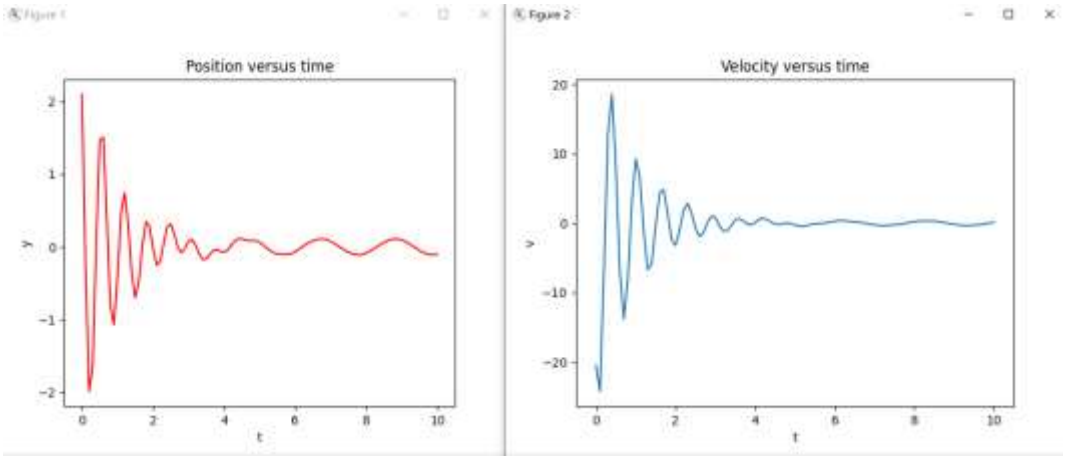

图 1.22 清单 1.12 用 rk4 算法求解二阶偏微分方程

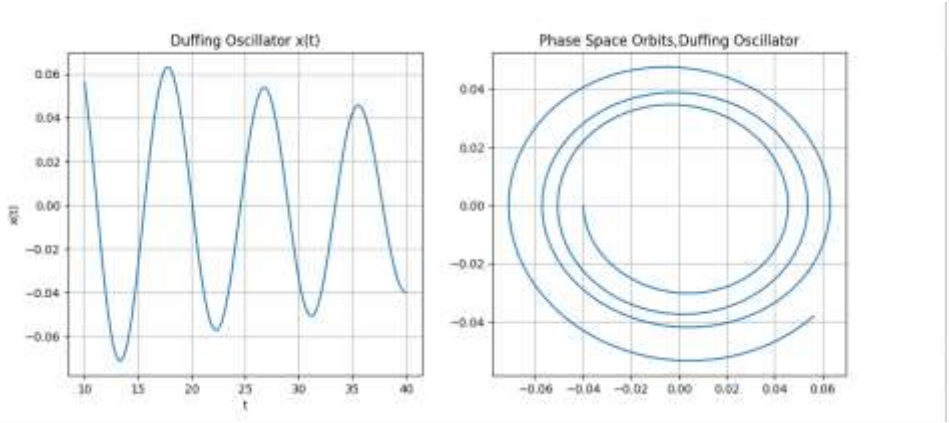

图 1.23 清单 1.13 用 rk4 算法求解二阶偏微分方程

这里的关键思想是围绕积分的中点展开，从而通过抵消 h 和 h3 项获得 O(h4) 精度。为提高精度付出的代价是必须近似三个导数以及区间中间的未知数 y

$$\mathbf{y}_{n+1} = \simeq \mathbf{y}_n + \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4)$$
$$\mathbf{k}_1 = h\mathbf{f}(t_n, \mathbf{y}_n), \quad \mathbf{k}_2 = h\mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{\mathbf{k}_1}{2}\right)$$
$$\mathbf{k}_3 = h\mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{\mathbf{k}_2}{2}\right), \quad \mathbf{k}_4 = h\mathbf{f}(t_n + h, \mathbf{y}_n + \mathbf{k}_3)$$

清单 1.12 中的程序 `rk4Call.py` 使用 rk4 来求解$y'' = -100y - 2y' + 100\sin(3t)$。作为替代方案，该程序代码清单 1.13 中的 `rk4Duffing.py` 求解$y'' = -2\gamma y' - \alpha y - \beta y^3 + F\cos\omega t$，使用 Matplotlib 绘制结果。注意，在第一次调用 rk4Algor.py 之后， `rk4Algor.pyc` 的编译版本将留在工作目录中。 在 rk4Call.py 开头 `from rk4Algor import rk4Algor` 的声明包含预编译方法。这是 rk4Call.py 的伪代码版本：

```
# Pseudocode for rk4Call.py for ODE y" = -100y-2y'+ 100 sin(3t)
import packages
Initialize variables, y(0) = position, y(1) = velocity
h = (Tend-Tstart)/N # Time step
Define f(t, y) # Function with RHS's
while (t < Tend)
call rk4Algor # y_new = y_old + Delta
t = t + h
plot each new point
```

给出 rk4Duffing.py 的伪代码版本：

```
# Pseudocode for rk4Duffing.py
import packages
Declare yy = all positions, vy = all velocities, tt = all t's
Define f(t, y) # RHS Force Function
i = 0, initialize y[0], y[1]
for 0 < t < 100
store tt[i], yy[i] = y[0], vy[i] = y[1]
call rk4Algor # rk4: y_new = y_old + Delta y
i = i + 1
plot yy, vy vectors
```

1.8 偏微分方程

没有一种算法可以应用于所有不同类型的偏微分方程。 尽管我们讨论的所有 PDE 解决方案都应用有限差分导数的近似值，细节取决于方程和边界条件。因此，下文介绍了不同的物理用

途：拉普拉斯和泊松方程的 relaxation，第 5.2.1 节和热和波动方程的时间步长，第 7.2 节、第 4.2 节。 作为一个例子，我们将在 §7.2 中更全面地介绍，热方程：

$$\frac{\partial T(x,t)}{\partial t} = \kappa \frac{\partial^2 T(x,t)}{\partial x^2}$$

是关于空间和时间的偏微分方程。 当时间和空间导数用有限差分近似，偏微分方程变为有限差分方程

$$\frac{T(x,t+\Delta t) - T(x,t)}{\Delta t} = \alpha \frac{T(x+\Delta x,t) + T(x-\Delta x,t) - 2T(x,t)}{\Delta x^2}$$

通过重新排序这个方程来形成一个算法，使得温度在较早的时间 j 可以向前推进以在较晚的时间产生温度 j+1：

$$T_{i,j+1} = T_{i,j} + \eta[T_{i+1,j} + T_{i-1,j} - 2T_{i,j}]$$

1.9 代码清单

```
from vpython import * # Import Vpython
graph1=graph ( align='left', width =400 , height=400 ,background=color.white , foreground = color.black )
Plot1=gcurve (color= color.black ) # gcurve method
for x in arange ( 0 , 8.1 , 0.1 ): # x range
    Plot1.plot(pos=(x,5*cos(2*x)*exp(-0.4*x)))
graph2=graph ( align="right" , width =400 , height=400 ,background=color.white , foreground=color.black ,
title="2-D Plot" , xtitle="x" , ytitle="f(x) " )
Plot2=gdots ( color=color.black ) # Dots
for x in arange ( -5 ,5 ,0.1) :
    Plot2.plot( pos=(x , cos(x) ) ) # plot dots
```

清单 1.1. EasyVisualVP.py 使用 VPython 获得图 1.1。

```
from vpython import *
string=" blue : sin^2( x ) , black = cos^2( x ) , cyan : sin(x) * cos ( x ) "
graph1=graph ( title=string , xtitle='x' , ytitle='y',background=color.white , foreground=color.black )
y1 = gcurve( color=color.blue ) # curve
y2 = gvbars( color=color.black ) # vertical bars
y3 = gdots( color=color.cyan ) # dots
```

```
for x in arange ( -5 ,5 ,0.1 ) : # arange for plots
    y1.plot ( pos=(x , sin(x) **2) )
    y2.plot ( pos=(x , cos(x) * cos(x) / 3.))
    y3.plot ( pos=(x , sin(x) * cos(x) ) )
```

清单 1.2. 3GraphVP.py 用 VPython 包产生一个 2-D x-y plot。

```
# EasyMatPlot . py : Simple use of matplotlib ''' splot command
from pylab import * # Load Matplotlib
Xmin = -5.; Xmax = +5.; Npoints= 500
DelX = (Xmax - Xmin) / Npoints
x = arange (Xmin , Xmax, DelX )
y = sin(x) * sin(x*x) # Function of array
print( 'arange => x[0] , x[1] , x[499]=%8.2f %8.2f %8.2f '%(x [ 0 ] , x [ 1 ] , x [ 499 ] ) )
print( 'arange => y[0] , y[1] , y[499]=%8.2f %8.2f %8.2f'%(y [ 0 ] , y [ 1 ] , y [ 499 ] ) )
print( " \n Doing plotting , look for Figure 1 " )
xlabel ( 'x' ) ; ylabel( 'f ( x )' ) ; title ( 'f ( x ) vs x' )
text ( -1.75 , 0.75 , ' MatPlotLib \n Example' ) # Text on plot
plot ( x , y , '-' , lw=2)
grid ( True ) # Form grid
show ( )
```

清单 1.3. EasyMatPlot.py 使用 Matplotlib 包产生一个 2-D x-y plot。

```
# GradeMatPlot.py : Matplotlib plot multi-data sets
import matplotlib.pyplot as plt # Matplotlib
from numpy import *
plt.title ( ' Grade Inflation ' ) # title and labels
plt.xlabel ( ' Years in College ' )
plt.ylabel ( ' GPA ' )
xa = array ([ -1 , 5 ] ) # For horinzontalline
ya = array ( [ 0 , 0 ] ) # " "
plt.plot ( xa , ya ) # Draw horinzontalline
x0 = array ( [ 0 , 1 , 2 , 3 , 4 ] ) # Data set 0 point
y0 = array ([ -1.4 , +1.1 , 2.2 , 3.3 , 4.0 ] )
plt.plot ( x0 , y0 , 'bo' ) # Data set 0 = bluecircle
plt.plot ( x0 , y0 , 'g' ) # Data set 0 =line
x1 = arange ( 0 , 5 , 1) # Data set 1 point
y1 = array ( [ 4.0 , 2.7 , -1.8 , -0.9 , 2.6 ] )
plt.plot ( x1 , y1 , 'r ' )
errTop = array ( [ 1.0 , 0.3 , 1.2 , 0.4 , 0.1 ] ) # Asymmetric error bars
errBot = array ( [ 2.0 , 0.6 , 2.3 , 1.8 , 0.4 ] )
```

```
plt.errorbar ( x1 , y1 , [ errBot , errTop ] , fmt = 'o ' ) # Plot error bars
plt.grid ( True ) # Gridline
plt.show ( )
```

清单 1.4 GradesMatPlot.py 用 Matplotlib 包产生 x-y plot

```
# MatPlot2figs.py : plot of 2 subplots on 1 fig , 2 separate figs
import matplotlib.pyplot as plt # Matplotlib
from numpy import *
Xmin = -5.0; Xmax = 5.0 ; Npoints= 500
DelX= (Xmax-Xmin) / Npoints # Delta x
x1 = arange (Xmin , Xmax, DelX ) # x1 range
x2 = arange (Xmin , Xmax, DelX /20 ) # different x2 range
y1 = -sin( x1 ) * cos( x1*x1 ) # Function 1
y2 = exp(-x2/4. ) * sin ( x2 ) # Function 2
print ( " \n Now plotting , look for Figures 1 & 2 on desktop " )

plt.figure ( 1 ) # Fig 1 如果不加 figure 则画在一个 figure 中
plt.subplot ( 2 , 1 , 1 ) # 1st subplot in first figure
plt.plot ( x1 , y1 , "r" , lw=2)
plt.xlabel ( "x" ) ; plt.ylabel ( "f(x)" ) ; plt.title ( "$\sin(x)*\cos(x^2)$ " )
plt.grid ( True ) # Form grid
plt.subplot ( 2 , 1 , 2 ) # 2nd subplot in first figure
plt.plot ( x2 , y2 , "-" , lw=2)
plt.xlabel ( "x" ) # Axes labels
plt.ylabel ( "f(x)" )
plt.title ( "$\exp(-x/4)*\sin(x)$" )

plt.figure ( 2 ) # Fig 2
plt.subplot ( 2 , 1 , 1 ) # 1st subplot in 2nd figure
plt.plot ( x1 , y1*y1 , "r" , lw=2)
plt.xlabel ( "x" ) ; plt.ylabel ( "f(x)" ) ; plt.title ( "$\sin^2(x)*\cos^2(x^2)$" )
plt.subplot ( 2 , 1 , 2 ) # 2nd subplot in 2nd figure
plt.plot ( x2 , y2*y2 , "-" , lw=2)
plt.xlabel ( "x" ) ; plt.ylabel ( "f(x)" ) ; plt.title ( "$\exp(-x/2)*\sin^2(x)$" )
plt.grid ( True ) # form grid
plt.show ( )
```

清单 1.5 MatPlot2figs.py 如图 1.4 产生两个 figure，每个 figure 有两个 subplot

```
# Simple3Dplot.py : matplotlib 3D plot , rotate & scale with mouse
import matplotlib.pyplot as plt
```

```
import numpy as np
from mpl_toolkits. mplot3d import Axes3D
print ( " Please be patient while I do importing & plotting " )
delta = 0.1
x = np.arange ( -3. , 3. , delta )
y = np.arange ( -3. , 3. , delta )
X , Y = np.meshgrid ( x , y )
Z = np.sin (X) * np.cos (Y) # Surface height
fig = plt.figure ( ) # Create figure
ax = Axes3D (fig) # Plots axes
ax.plot_surface (X, Y, Z) # Surface
ax.plot_wireframe (X, Y, Z , color = "r" ) # Add wireframe
ax.set_xlabel ( "X" )
ax.set_ylabel ( "Y" )
ax.set_zlabel ( "Z" )
plt.show ( ) # Output figure
```

清单 1.6 Simple3Dplot.py 产生 Matplotlib 的 3-D 表面图，如图 1.5

```
# PondMatPlot.py : Monte_Carlo intergration via von Neumann rejction
import numpy as np , matplotlib.pyplot as plt
N, Npts = 100 , 50000;
analyt = np.pi**2 # f(x)=x*sin^2(x)在[0,pi]的积分结果
def fx(x) : return x*np.sin(x)**2 # Integrand
xx = 2.* np.pi * np.random.rand ( Npts ) # 0 =< x <= 2pi : rand 实现从[0,1]的均匀分布
yy = 5*np.random.rand ( Npts ) # 0 =< y <= 5
boxarea = 2.* np.pi * 5.# Box area 总面积
xi = [ ] ; yi = [ ] ; xo = [ ] ; yo = [ ]
j = 0 # Inside curve counter
for i in range ( Npts ) :
    if  yy [i] <= fx ( xx [i] ) : # Below curve
        xi.append ( xx [i] )
        yi.append ( yy [i] )
        j +=1
    else :
        yo.append ( yy [i] )
        xo.append ( xx [i] )
    area = boxarea * j / Npts # Area under curve
plt.plot ( xo , yo ," bo" , markersize =1) # markersize 标记大小
plt.plot ( xi , yi ," ro" , markersize =1)
x1 = np.arange ( 0 , 2*np.pi, 2*np.pi/N) # 区域[0,2pi)
y1 = x1 * np.sin ( x1 ) **2 # Integrand
```

```
plt.plot ( x1 , y1 ,"k" , lw=4)
plt.xlim ( ( 0 , 2*np.pi ) )
plt.ylim ( ( 0 , 5 ) )
plt.xticks ([0 , np.pi , 2*np.pi], ["0" ,"$\pi$" ,"2$\pi$"] )
plt.ylabel (" $f(x) = x*sin^2(x)$" , fontsize =20)
plt.xlabel ("x" , fontsize =20)
plt.title (" Answers : Analytic = %5.3f , MC = %5.3f" %(analyt , area ) )
plt.show ( )
```

清单 1.7. PondMatPlot.py 产生图 1.6 左侧的散点图和曲线图

```
# Scatter3dPlot.py : MatplotlibScatterplot example
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
n = 100
def randrange (n , vmin , vmax ):
    return (vmax - vmin ) * np.random.rand (n) + vmin
xs = randrange (n , 23 , 32 ) # 在[23,32]均匀取 100 个点
ys = randrange (n , 0 , 100 )
zs1 = randrange (n , -50, -25 )
zs2 = randrange (n , -30, -5 )
fig= plt.figure ( )
ax = fig.add_subplot (111 , projection="3d" )
ax.scatter ( xs , ys , zs1 , c="r" , marker="o")
ax.scatter ( xs , ys , zs2 , c="b" , marker="^")
ax.set_xlabel ("X Label" )
ax.set_ylabel ("Y Label" )
ax.set_zlabel ("Z Label" )
plt.show( )
```

清单 1.8 Scatter3dPlot.py 用 Matplotlib3D 工具产生 3-D 散点图

```
#TwoForces.py Forces on two moving strings
from vpython import *
posy = 100 ; Lcord = 250         #basic height , cord length
Hweight = 50 ; W = 10         #cylinder height , weight
magF = W/2.0          # 每个学生的初始拉力
v = 2.0          #(m/s) 学生的初始速度
x1 = 0.0          #初始位置
scalefactor = 8 # 力的缩放因子
```

```
canvas(width=700,height=500,background=color.black) # 幕布：3D 应该用 canvas；2D 用 graph
curve(pos = [vector(-300 , posy , 0) , vector(300 , posy , 0)]) # 折线：把所在点都连起来
curve(pos = [vector(0 , posy , 0) , vector(0 , -150 , 0)])
label(pos = vector(0 , 240 , 0) , text = "angle(deg):" , box = 0)
label(pos = vector(200 , 240 , 0) , text = "Force:" , box = 0)
label(pos = vector(-200 , 240 , 0) , text = "Force:" , box = 0)
local_light(pos = vector(0 , 0 , 20) , color = color.yellow)         # 光源的位置和颜色

kilogr = cylinder(pos = vector(0 , posy-Lcord , 0) , radius = 20 , axis = vector(0 , -
Hweight , 0) , color = color.red)         #kgasacylinder
cord1 = cylinder(pos = vector(0 , posy , 0) , axis = vector(0 , -Lcord , 0) , color = color.yellow , radius = 2)
cord2 = cylinder(pos = vector(0 , posy , 0) , axis = vector(0 , -Lcord , 0) , color = color.yellow , radius = 2)
arrow1 = arrow(pos = vector(0 , posy , 0) , color = color.orange)         #Tension cord1
arrow2 = arrow(pos = vector(0 , posy , 0) , color = color.orange)         #Tension cord2
angultext = label(pos = vector(20 , 210 , 0) , box = 0)
Ftext1 = label(pos = vector(200 , 210 , 0) , box = 0)
Ftext2 = label(pos = vector(-200 , 210 , 0) , box = 0)

for t in arange(0. , 120.0 , 0.2):
    rate(100)         #播放速度：1 秒 100 帧=间隔 0.01s
    x1 = v*t         #1 to right , 2 to left
    theta = asin(x1/Lcord)         # angle cord 相对于 y 轴的夹角
    poscil = posy-Lcord*cos(theta)         #cylinder height
    kilogr.pos = vector(0 , poscil , 0)         # 更新重物的高度
    force = W/(2.*cos(theta))         #Cord tension
    angle = 180.*theta/pi
    cord1.pos = vector(x1 , posy , 0)         #position cord end
    cord1.axis = vector(-Lcord*sin(theta) , -Lcord*cos(theta) , 0)
    cord2.pos = vector(-x1 , posy , 0)         #position end cord
    cord2.axis = vector(Lcord*sin(theta) , -Lcord*cos(theta) , 0)
    arrow1.pos = cord1.pos         #axis arrow
    arrow1.axis = vector(scalefactor*force*sin(theta) , scalefactor*force*cos(theta) , 0) # 力的大小用 8 倍缩放，可选任意倍
    arrow2.pos = cord2.pos
    arrow2.axis = vector(-scalefactor*force*sin(theta) , scalefactor*force*cos(theta) , 0)
    angultext.text = '{:.2f}'.format(angle)
    Ftext1.text = '{:.2f}'.format(force)       #Tension
    Ftext2.text = '{:.2f}'.format(force)
```

清单 1.9. TwoForces.py 产生 3-D 动画，显示两个学生拉着重物。

```
#  SlidingBox.py : 3-Danimation of forces on a beam as box slides
from vpython import *
```

```
Hsupport = 30 ; d =100  #  height , distance supports
Lbeam=500 ; Wbeam=80 ; thickness=10  #  beam dimensions
W=200 ; WeightBox=400  #  weight of table , box
Lbox=60 ; Wbox=60 ; Hbox=60  #  Box Dimensions
v=4.0  #  box speed
xt=-d  #  box initial position
Mg=WeightBox+W  #  weight box+beam
Fl=(2*Wbox+W)/2.0 # 初始支持力
Fr=Mg-Fl  #  right force

#  Graphics
scene=canvas(width=750 , height=500 , range=300)
scene.forward=vector(0.5 , -0.2 , -1)  #  to change point of view
support1=cone(pos=vector(-d , 0 , 0) , axis=vector(0 , Hsupport , 0) , color=color.yellow , radius=20)
support2=cone(pos=vector(d , 0 , 0) , axis=vector(0 , Hsupport , 0) , color=color.yellow , radius=20)
beam=box(pos=vector(0 , Hsupport+thickness/2 , 0) , color=color.orange ,length=Lbeam , width=Wbeam , height=thickness)
cube=box(pos=vector(-d , Hsupport+Hbox/2+thickness , 0) , length=Lbox , width=Wbox , height=Hbox)
label(pos=vector(-100 , 150 , 0) , text="Fl=" , box=0)
label(pos=vector(100 , 150 , 0) , text="Fr=" , box=0)
arrowFl=arrow(color=color.red , pos=vector(-d , Hsupport+thickness/2 , 0) , axis=vector(0 , 0.15*Fl , 0))
arrowFr=arrow(color=color.red , pos=vector(d , Hsupport+thickness/2 , 0) , axis=vector(0 , 0.15*Fr , 0))
Ftext1=label(pos=vector(-50 , 153 , 0) , box=0)
Ftext2=label(pos=vector(150 , 153 , 0) , box=0)
weightcube=arrow(color=color.orange , axis=vector(0 , -0.15*Wbox , 0)) # 滑块所受重力的向量
piso=curve(pos=[(-300 , 0 , 0) , (300 , 0 , 0)] , color=color.green , radius=1)
weightbeam=arrow(pos=vector(0 , Hsupport+thickness/2 , 0),color=color.orange , axis=vector(0 , -0.15*W , 0))

for t in arange(0.0 , 65.0 , 0.5):
    rate(10)
    xt=-d+v*t # 滑块的坐标
    cube.pos.x=xt #  position cube
    weightcube.pos.x=xt
    if Fl>0:
        Fl=(d*Mg-xt*WeightBox)/(2.0*d)
        Fr=Mg-Fl
        cube.pos.x=xt
        weightcube.pos.x=xt
        arrowFl.axis=vector(0 , 0.15*Fl , 0)
        arrowFr.axis=vector(0 , 0.15*Fr , 0)
        Ftext1.text='{:.2f}'.format(Fl)  #  Left force
        Ftext2.text='{:.2f}'.format(Fr)  #  Right force
    elif Fl==0:
```

```
        # 滑块掉落后，横梁绕过右支点的 z 轴顺时针旋转 0.2 度
        beam.rotate(angle=-0.2 , axis=vector(0 , 0 , 1) , origin=vector(d , Hsupport+thickness/2 , 0))
        cube.pos=vector(300 , Hbox/2 , 0)
        weightcube.pos=vector(300 , 0 , 0)
        break
arrowFl.axis=vector(0 , 0.15*W/2 , 0)  #  滑块掉落之后的力
arrowFr.axis=arrowFl.axis
beam.rotate(angle=0.2 , axis=vector(0 , 0 , 1) , origin=vector(d , Hsupport+thickness/2 , 0))
Fl=100.0 # 100 N
Ftext1.text='{:.2f}'.format(Fl)
Ftext2.text='{:.2f}'.format(Fl)
```

清单 1.10. SlidingBox.py 滑块问题 3D 动画

```
#  CentralValue.py:Gaussian distribution from sum of randoms
import random
import numpy as np
import matplotlib.pyplot as plt
N = 1000;NR = 10000  #  SumN variables,distribution of sums
mu = 0
SumList = []  #  empty list
def SumRandoms():  #  SumN randoms in [0,1]
    sum = 0.0
    for i in range(0,N):
        sum = sum+random.random()
    return sum
def normal_dist_param():
    add = sum2 = 0
    for i in range(0,NR):
        add = add+SumList[i] # 1w 个数之和
    mu = add/NR  #  Average distribution 大概 500
    for i in range(0,NR):
        sum2 = sum2+(SumList[i]-mu)**2
    sigma = np.sqrt(sum2/NR) # 方差
    return mu,sigma
for i in range(0,NR):
    dist = SumRandoms()
    SumList.append(dist)  #  Fill list with NR sums

mu,sigma = normal_dist_param()
x = np.arange(450,550)
rho = np.exp(-(x-mu)**2/(2*sigma**2))/(np.sqrt(2*np.pi*sigma**2)) # 正态分布函数
```

```
plt.hist(SumList,bins = 100,color = "cyan",density=True)  #  density 表示面积是 1 ; Bins 代表最小值和最大值之间分割成多少份
plt.plot(x,rho,"g-",linewidth = 3.0)  #  Normal distrib 分析的
plt.xlabel("Random Number x1000") # 1000 个数之和
plt.ylabel("Average of Random Numbers") # 应该是和的出现概率
plt.title("Generated vs Analytic Normal Distribution")
plt.show()
```

清单 1.11. CentralValue.py 均匀分布的随机数之和跟正态分布对比

```
# rk4Call.py:4th-O Runge-Kutta that calls rk4Algor
# Here for ODE y" = -100y-2y'+100sin(3t)
# 用 y'积分得到 y，用 y"积分得到 y'
import numpy as np, matplotlib.pyplot as plt
def rk4Algor(t, h, N, y, f):
    k1=np.zeros(N); k2=np.zeros(N); k3=np.zeros(N); k4=np.zeros(N)
    k1 = h*f(t,y)
    k2 = h*f(t+h/2.,y+k1/2.)
    k3 = h*f(t+h/2.,y+k2/2.)
    k4 = h*f(t+h,y+k3)
    y=y+(k1+2*(k2+k3)+k4)/6.
    return y # y[0] = y, y[1] = y', y[2] = y"
def f(t,y):  # Force(RHS) function
    fvector = np.zeros((2))
    fvector[0] = y[1] # y 的导数是 y'
    fvector[1] = -100.*y[0]-2.*y[1]+10.*np.sin(3.*t) # y'的导数是 y"=...
    return fvector

Tstart = 0. ; Tend = 10. ; Nsteps = 100  #Initialization
tt = [] ; yy = [] ; yv = []
y = [3.,5.]  #Initial position & velocity
t = Tstart ; h = (Tend-Tstart)/Nsteps ; # 10s 分成 100 步，每步 0.1s

while(t<Tend):
    tt.append(t)  #Time loop 时间从 0 加到 10s，记为 tt
    y = rk4Algor(t,h,2,y,f) # rk4 迭代法：传入当前时间 t，当前的位置和速度向量 y，步长 h，方程阶数 N=2，力函数 f
    yy.append(y[0]) # 每次更新时间和 y，获取新的位置和速度。
    yv.append(y[1])
    t = t+h # 由于数据误差，只能加到 9.999..

plt.plot(tt,yy,'r')
plt.title('Position versus time')
plt.xlabel('t')
```

```
plt.ylabel('y')
plt.figure() # 再打开一张图
plt.plot(tt,yv)
plt.title('Velocity versus time')
plt.xlabel('t')
plt.ylabel('v')
plt.show()
```

清单 1.12&1.14. rk4Call.py 用四阶 Runge-Kutta 算法解决常微分方程，rk4Algor 是 rk4 算法的细节。

```
#rk4Duffing.py solve ODE for Duffing Osc via rk4 & Matplotlib
import numpy as np,matplotlib.pylab as plt
def rk4Algor(t, h, N, y, f):
    k1 = np.zeros(N); k2=np.zeros(N); k3=np.zeros(N); k4=np.zeros(N)
    k1 = h*f(t,y)
    k2 = h*f(t+h/2.,y+k1/2.)
    k3 = h*f(t+h/2.,y+k2/2.)
    k4 = h*f(t+h,y+k3)
    y = y+(k1+2*(k2+k3)+k4)/6.
    return y # y[0] = y, y[1] = y', y[2] = y"

tt=[];yy=[];vy=[]
y=np.zeros(2)
a=0.5;b=-0.5;g=0.02;
A,w,h=0.0008,1.,0.01
y[0]=0.09;y[1]=0.  #Initial x,velocity
def f(t,y):
    rhs=np.zeros(2)
    rhs[0]=y[1]
    rhs[1]=-2*g*y[1]-a*y[0]-b*y[0]**3+A*np.cos(w*t)
    return rhs

for t in np.arange(0,40,h):  #TimeLoop 40/0.01=4000 步
    y=rk4Algor(t,h,2,y,f)  #Callrk4
    tt.append(t)
    yy.append(y[0])  #x(t)
    vy.append(y[1])  #v(t)

fig,axes=plt.subplots(1,2,figsize=(12,5)) # 1 行 2 列的图，也可以用 subplot
axes[0].plot(tt[1000:],yy[1000:]) #1000 avoids transients
```

```
axes[0].grid()
axes[0].set_title("Duffing Oscillator x(t)")
axes[0].set_xlabel("t")
axes[0].set_ylabel("x(t)")
axes[1].plot(yy[1000:],vy[1000:])
axes[1].grid()
axes[1].set_title("Phase Space Orbits,Duffing Oscillator")
plt.show()
```

清单 1.13 rk4Duffing.py 用 rk4 计算 ODE 方程，跟 1.12 差别只在于画图。

```
#TrapMethods.py:trapezoid integrnt,a<x<b,Npts,N-1 intervals
import numpy as np
from scipy import integrate
def func(x):
    return 5*(np.sin(8*x))**2*np.exp(-x*x)-13*np.cos(3*x)
def trapezoid(A,B,N):
    h=(B-A)/N  # stepsize
    sum=(func(A)+func(B))*h/2  #(1st+last)/2，求梯形面积
    for i in range(1,N):
        sum+=func(A+i*h)*h
    return sum
A=0.5 ; B=2.3 # 积分区间
N=1200
print(trapezoid(A,B,N)) # 梯形法
print(integrate.quad(func,A,B)[0]) # 借用 scipy 的积分来比较
```

清单 1.15 TrapMethods.py 用矩形规则积分 f(y)注意到步长 h 依赖于分割数目 N。

```
# GaussPoints.py:N point Gaussian quadrature pts & Wts generation
# 本质上就是因为函数在[-1,1]区间可以进行多项式展开，用 Npt 个点取拟合多项式的积分
# N 个参数点可以拟合到 2N 阶展开.
import numpy as np
def Gaussxw(N,eps):
    # Initial approximation to roots of the Legendre polynomial
    a = np.linspace(3,4*N-1,N)/(4*N+2)
    x = np.cos(np.pi*a+1/(8*N*N*np.tan(a)))
    # Find roots using Newton's method
    epsilon = eps
    delta = 1.0
```

```
    while delta>epsilon:
        p0 = np.ones(N,float)
        p1 = np.copy(x)
        for k in range(1,N):
            p0,p1 = p1,((2*k+1)*x*p1-k*p0)/(k+1)
        dp = (N+1)*(p0-x*p1)/(1-x*x)
        dx = p1/dp
        x -= dx
        delta = max(abs(dx))
    # Calculate the weights
    w = 2*(N+1)*(N+1)/(N*N*(1-x*x)*dp*dp)
    return x,w

Npts=3;Ans=0;a=0.;b=1.;eps=3.E-14
def f(x):return np.exp(x)  # Integrand
x,w = Gaussxw(Npts,eps)  # 返回根和权重
for j in range(0,Npts):
    x[j]=x[j]*(b-a)/2.+(b+a)/2.
    w[j]=w[j]*(b-a)/2.
for i in range(0,Npts):Ans+=f(x[i])*w[i]  #Sum integrands
print("Npts=",Npts,",Ans=",Ans)
print("eps=",eps,",Error=",Ans-(np.exp(1)-1))
```

清单 1.16&1.17 GaussPoints.py 为 Gaussian quadrature 产生点和权重，被 IntegGaussCall.py 调用。