

计算物理学

■ 湖南师范大学

■ 桂龙成

■ 2019-11-10

Outline

第一节 计算物理简介

第二节 linux 系统编程简介

第三节 Python 语言基础

第四节 Mathematica 基础

第五节 函数近似

第六节 数值微积分

第七节 常微分方程

第八节 矩阵的数值方法

第一节 计算物理简介

第二节 linux 系统编程简介

第三节 Python 语言基础

第四节 Mathematica 基础

第五节 函数近似

第一节

计算物理简介

1.1 计算物理学的起源与发展

1.2 计算与科学

1.3 现代计算机

1.4 程序语言与计算机算法

1.5 误差分析

1.6 浮点数相关

计算物理学的起源与发展

■ 电子计算机的发明和应用.

计算物理学的起源与发展

- 电子计算机的发明和应用.
- 学科之间的交叉渗透，使计算物理学得以蓬勃的发展.

计算物理学的起源与发展

- 电子计算机的发明和应用.
- 学科之间的交叉渗透，使计算物理学得以蓬勃的发展.
- 计算物理学对解决复杂物理问题的巨大能力，使它成为物理学的第三支柱，并在物理学研究中占有重要的位置.

计算物理学的起源与发展

- 电子计算机的发明和应用.
- 学科之间的交叉渗透，使计算物理学得以蓬勃的发展.
- 计算物理学对解决复杂物理问题的巨大能力，使它成为物理学的第三支柱，并在物理学研究中占有重要的位置.
- 计算物理学与理论物理和实验物理有着密切的联系. 计算物理学的研究内容涉及到物理学的各个领域.

计算物理学在物理学研究中的应用

■ 计算机数值分析

计算物理学在物理学研究中的应用

- 计算机数值分析
- 计算机符号处理

计算物理学在物理学研究中的应用

- 计算机数值分析
- 计算机符号处理
- 计算机模拟

计算物理学在物理学研究中的应用

- 计算机数值分析
- 计算机符号处理
- 计算机模拟
- 计算机实时控制

计算物理学的研究方法

■ 确定物理模型

计算物理学的研究方法

- 确定物理模型
- 选取数学方法

计算物理学的研究方法

- 确定物理模型
- 选取数学方法
- 分析计算结果

计算物理学的研究方法

- 确定物理模型
- 选取数学方法
- 分析计算结果
- 给出物理结论

计算物理学的研究方法

三体问题：

1 建立微分方程组（建立模型）

计算物理学的研究方法

三体问题：

- 1 建立微分方程组（建立模型）
- 2 计算机求数值解（选择算法）

计算物理学的研究方法

三体问题：

- 1 建立微分方程组（建立模型）
- 2 计算机求数值解（选择算法）
- 3 研究数值解的稳定性，给出相关物理结论.（解释结果）

第一节

计算物理简介

1.1 计算物理学的起源与发展

1.2 计算与科学

1.3 现代计算机

1.4 程序语言与计算机算法

1.5 误差分析

1.6 浮点数相关

计算与科学

- 人类历史上曾经计算过的最令人着迷的数字之一
 π

计算与科学

- 人类历史上曾经计算过的最令人着迷的数字之一
 π
- 公元三世纪，中国数学家刘徽即在其书本中记录了对 π 的计算方法.

计算与科学

- 人类历史上曾经计算过的最令人着迷的数字之一
 π
- 公元三世纪，中国数学家刘徽即在其书本中记录了对 π 的计算方法。
- 两年后，中国数学家和天文学家祖冲之进行了进一步计算，给出了七位精度的结果，领先世界一千年。

计算与科学

- 人类历史上曾经计算过的最令人着迷的数字之一
 π
- 公元三世纪，中国数学家刘徽即在其书本中记录了对 π 的计算方法。
- 两年后，中国数学家和天文学家祖冲之进行了进一步计算，给出了七位精度的结果，领先世界一千年。
- 有现代数学知识的我们是否可以做得更好？

割圆术

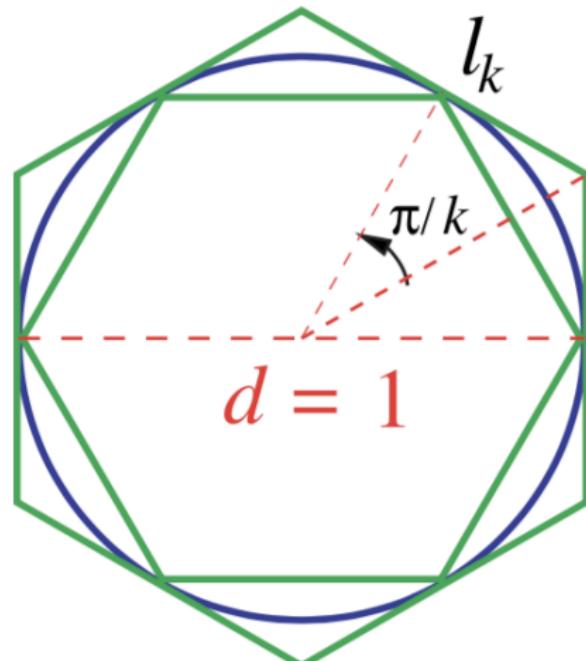
■ 刘徽的割圆术 (公元三世纪)

割圆术

- 刘徽的割圆术 (公元三世纪)
- 用正多边形去近似圆周长

割圆术

- 刘徽的割圆术 (公元三世纪)
- 用正多边形去近似圆周长



割圆术

- 两个正 192 多边形给出 $3.1410 < \pi < 3.1427$

割圆术

- 两个正 192 多边形给出 $3.1410 < \pi < 3.1427$
- 正 3072 多边形给出 $\pi \simeq 3.1416$

割圆术

- 两个正 192 多边形给出 $3.1410 < \pi < 3.1427$
- 正 3072 多边形给出 $\pi \simeq 3.1416$
- 祖冲之和他的儿子祖暅通过正 24576 边形给出 $3.1415926 < \pi < 3.1415927$. 领先世界 1000 年!

我们能否做得更好？

- 我们现在有了更多的数学知识，是否能以他们的结果为基础来改进？

我们能否做得更好?

- 我们现在有了更多的数学知识，是否能以他们的结果为基础来改进？
- 假设 k 多边形的边长为 l_k ，则 π 近似为

$$\pi_k = k l_k$$

我们能否做得更好?

- 我们现在有了更多的数学知识, 是否能以他们的结果为基础来改进?
- 假设 k 多边形的边长为 l_k , 则 π 近似为

$$\pi_k = k l_k$$

- 精确值 π 为 π_k 当 $k \rightarrow \infty$. π_k 的值可以形式的表示为

$$\pi_k = \pi_\infty + \frac{c_1}{k} + \frac{c_2}{k^2} + \frac{c_3}{k^3} + \dots$$

我们能否做得更好?

- 假设我们截断到某个 c_i . 则我们的任务即计算出所有的 c_i . 这等价于求解

$$\sum_{j=1}^n a_{ij}x_j = b_i$$

我们能否做得更好?

- 假设我们截断到某个 c_i . 则我们的任务即计算出所有的 c_i . 这等价于求解

$$\sum_{j=1}^n a_{ij}x_j = b_i$$

- 如果我们算到了正 64 边形给出

$$\pi_8 = 3.061467, \pi_{16} = 3.121445$$

$$\pi_{32} = 3.136548, \pi_{64} = 3.140331$$

我们能否做得更好?

- 假设我们截断到某个 c_i . 则我们的任务即计算出所有的 c_i . 这等价于求解

$$\sum_{j=1}^n a_{ij}x_j = b_i$$

- 如果我们算到了正 64 边形给出

$$\pi_8 = 3.061467, \pi_{16} = 3.121445$$

$$\pi_{32} = 3.136548, \pi_{64} = 3.140331$$

- 截断到 $1/k^3$, 从而得到 $\pi \simeq 3.14159265$. (大家练习)

第一节

计算物理简介

1.1 计算物理学的起源与发展

1.2 计算与科学

1.3 现代计算机

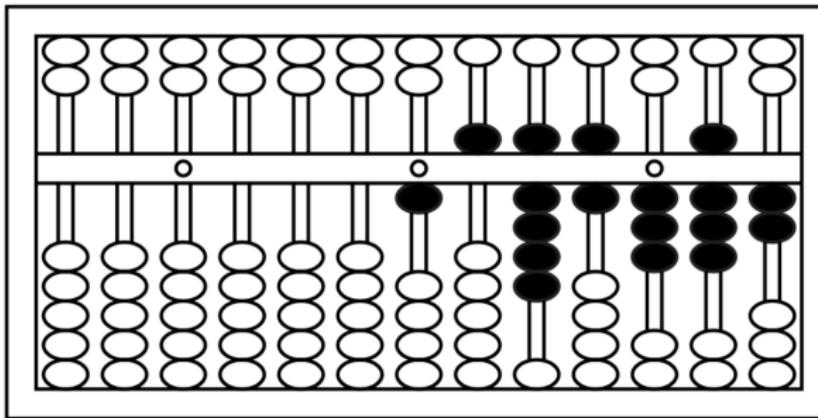
1.4 程序语言与计算机算法

1.5 误差分析

1.6 浮点数相关

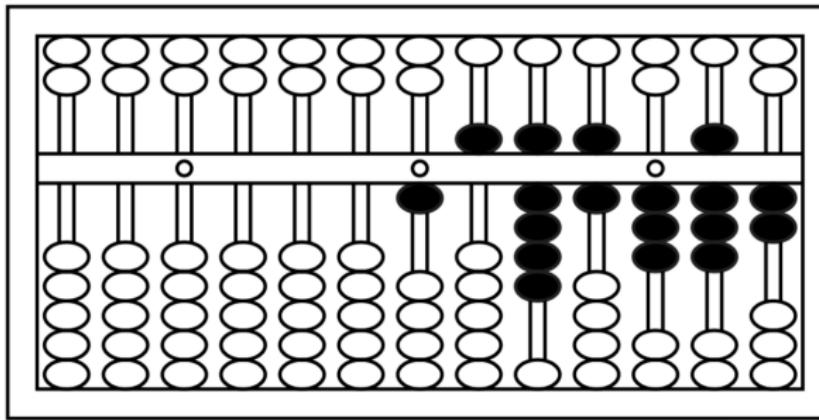
古老的计算机

■ 4000 年前的计算设备-算盘



古老的计算机

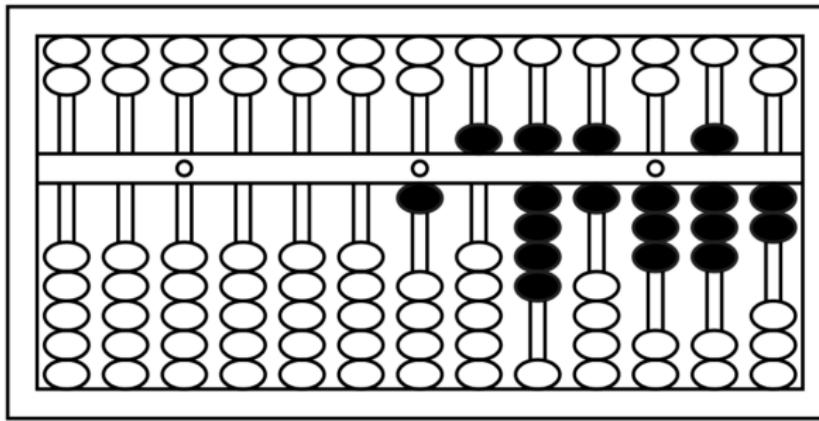
■ 4000 年前的计算设备-算盘



■ 算盘：输入设备 + 运算器 + 输出设备；

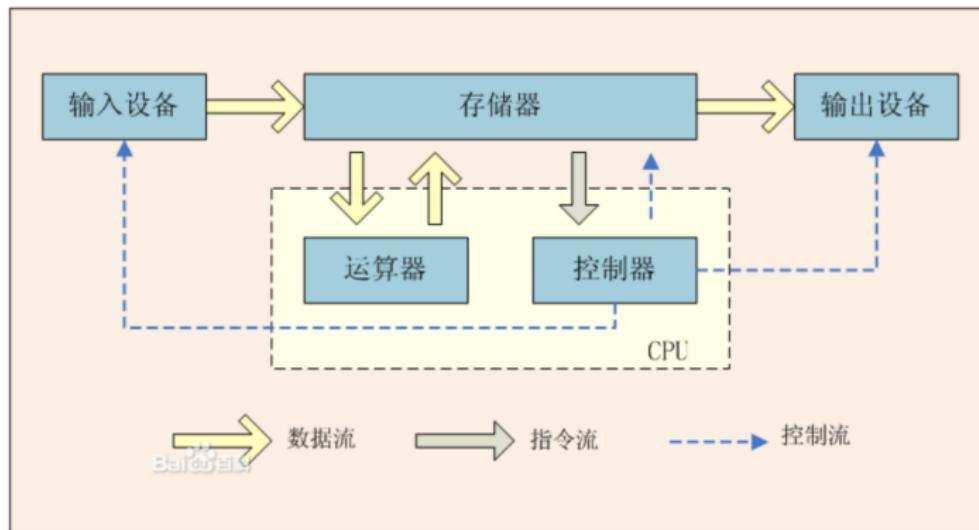
古老的计算机

■ 4000 年前的计算设备-算盘



- 算盘：输入设备 + 运算器 + 输出设备；
- 冯诺伊曼计算机：运算器 + 控制器 + 存储器 + 输入设备 + 输出设备

冯诺依曼体系结构



逻辑门

Logic Gates

Name	NOT	AND	NAND	OR	NOR	XOR	XNOR																																																																																																
Alg. Expr.	\bar{A}	AB	\bar{AB}	$A+B$	$\bar{A}+\bar{B}$	$A \oplus B$	$\bar{A} \oplus \bar{B}$																																																																																																
Symbol																																																																																																							
Truth Table	<table border="1"><thead><tr><th>A</th><th>X</th></tr></thead><tbody><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></tbody></table>	A	X	0	1	1	0	<table border="1"><thead><tr><th>B</th><th>A</th><th>X</th></tr></thead><tbody><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></tbody></table>	B	A	X	0	0	0	0	1	0	1	0	0	1	1	1	<table border="1"><thead><tr><th>B</th><th>A</th><th>X</th></tr></thead><tbody><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></tbody></table>	B	A	X	0	0	1	0	1	1	1	0	1	1	1	0	<table border="1"><thead><tr><th>B</th><th>A</th><th>X</th></tr></thead><tbody><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></tbody></table>	B	A	X	0	0	0	0	1	0	1	0	0	1	1	1	<table border="1"><thead><tr><th>B</th><th>A</th><th>X</th></tr></thead><tbody><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></tbody></table>	B	A	X	0	0	1	0	1	0	1	0	1	1	1	0	<table border="1"><thead><tr><th>B</th><th>A</th><th>X</th></tr></thead><tbody><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></tbody></table>	B	A	X	0	0	0	0	1	1	1	0	1	1	1	0	<table border="1"><thead><tr><th>B</th><th>A</th><th>X</th></tr></thead><tbody><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></tbody></table>	B	A	X	0	0	1	0	1	0	1	0	0	1	1	1
A	X																																																																																																						
0	1																																																																																																						
1	0																																																																																																						
B	A	X																																																																																																					
0	0	0																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	1																																																																																																					
B	A	X																																																																																																					
0	0	1																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	0																																																																																																					
B	A	X																																																																																																					
0	0	0																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	1																																																																																																					
B	A	X																																																																																																					
0	0	1																																																																																																					
0	1	0																																																																																																					
1	0	1																																																																																																					
1	1	0																																																																																																					
B	A	X																																																																																																					
0	0	0																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	0																																																																																																					
B	A	X																																																																																																					
0	0	1																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	1																																																																																																					

累加器

■ 5+7 换成二进制加法

$$\begin{array}{r} 0000\ 0101 \\ +0000\ 0111 \\ =0000\ 1100 \end{array}$$

累加器

■ 5+7 换成二进制加法

$$\begin{array}{r} 0000\ 0101 \\ +0000\ 0111 \\ \hline =0000\ 1100 \end{array}$$

■ 每一位的加法分解为两种动作

累加器

■ 5+7 换成二进制加法

$$\begin{array}{r} 0000\ 0101 \\ +0000\ 0111 \\ \hline =0000\ 1100 \end{array}$$

- 每一位的加法分解为两种动作
 - 同一位的两个数字相加

累加器

■ 5+7 换成二进制加法

$$\begin{array}{r} 0000\ 0101 \\ +0000\ 0111 \\ \hline =0000\ 1100 \end{array}$$

■ 每一位的加法分解为两种动作

- 同一位的两个数字相加
- 如果当前位结果大于 1，则向前进一位

累加器

- 第一个动作可能的结果（真值表）有：

累加器

- 第一个动作可能的结果（真值表）有：
 - $0 \text{ XOR } 0 = 0$

累加器

- 第一个动作可能的结果（真值表）有：
 - $0 \text{ XOR } 0 = 0$
 - $1 \text{ XOR } 0 = 1$

累加器

- 第一个动作可能的结果（真值表）有：
 - $0 \text{ XOR } 0 = 0$
 - $1 \text{ XOR } 0 = 1$
 - $0 \text{ XOR } 1 = 1$

累加器

- 第一个动作可能的结果（真值表）有：
 - $0 \text{ XOR } 0 = 0$
 - $1 \text{ XOR } 0 = 1$
 - $0 \text{ XOR } 1 = 1$
 - $1 \text{ XOR } 1 = 0$

累加器

- 第二步进位，只有 $1+1$ 才需要进位 1，所以真值表如下

累加器

- 第二步进位，只有 $1+1$ 才需要进位 1，所以真值表如下
 - 0 AND 0=0

累加器

- 第二步进位，只有 $1+1$ 才需要进位 1，所以真值表如下
 - 0 AND 0=0
 - 1 AND 0=0

累加器

■ 第二步进位，只有 $1+1$ 才需要进位 1，所以真值表如下

- 0 AND 0=0
- 1 AND 0=0
- 0 AND 1=0

累加器

■ 第二步进位，只有 $1+1$ 才需要进位 1，所以真值表如下

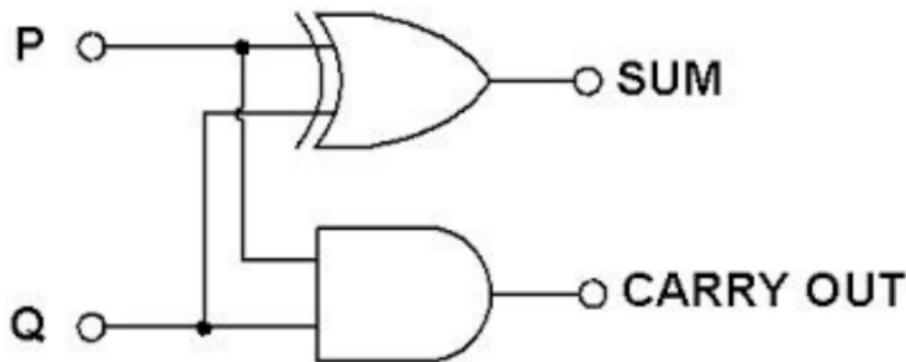
- 0 AND 0=0
- 1 AND 0=0
- 0 AND 1=0
- 1 AND 1=1

累加器

- 把一个“异或门”和一个“与门”组合到一起，就构成了一个“一位半加器”：

累加器

- 把一个“异或门”和一个“与门”组合到一起，就构成了一个“一位半加器”：

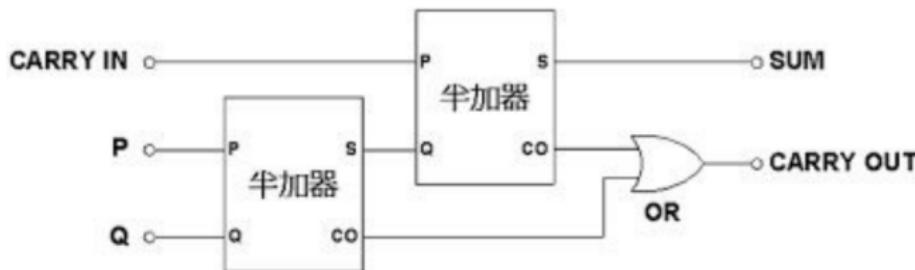


累加器

- 高位的二进制加法需要考虑 3 个输入，两个半加器组合构成一个完整的一位全加器

累加器

- 高位的二进制加法需要考虑 3 个输入，两个半加器组合构成一个完整的一位全加器

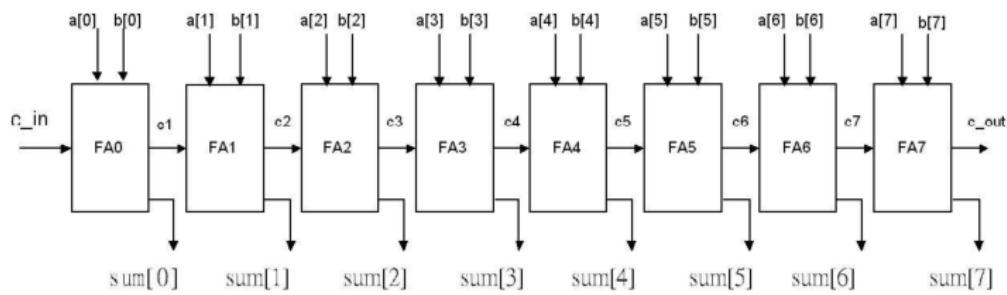


累加器

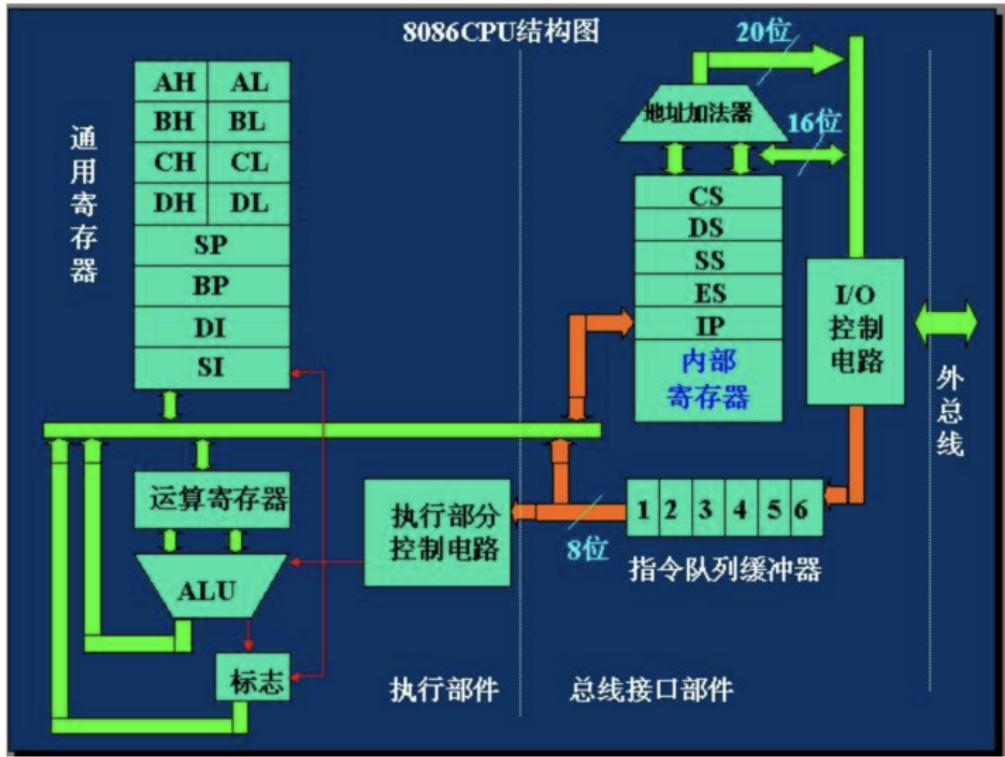
- 把 8 个这样的一位全加器组合起来，就构成了一个“八位全加器”

累加器

■ 把 8 个这样的一位全加器组合起来，就构成了一个“八位全加器”



cpu



第一节

计算物理简介

1.1 计算物理学的起源与发展

1.2 计算与科学

1.3 现代计算机

1.4 程序语言与计算机算法

1.5 误差分析

1.6 浮点数相关

CPU 指令和编程语言

- 我们的程序都会转化为二进制给 cpu, cpu 如何知道这些组合的意思?

CPU 指令和编程语言

- 我们的程序都会转化为二进制给 cpu, cpu 如何知道这些组合的意思?
- cpu 指令, 每款 CPU 在设计时就规定了一系列与其硬件电路相配合的指令系统. 有了 CPU 指令集的文档你就可以通过这个编写 CPU 认识的机器代码了

CPU 指令和编程语言

- 我们的程序都会转化为二进制给 cpu, cpu 如何知道这些组合的意思?
- cpu 指令, 每款 CPU 在设计时就规定了一系列与其硬件电路相配合的指令系统. 有了 CPU 指令集的文档你就可以通过这个编写 CPU 认识的机器代码了

指令	格式	说明
0011	[register1][register2]	加法操作

汇编语言

- 使用 0 和 1 这样的机器语言好处是 CPU 认识，可以直接执行，但是对于程序本身来说，没有可读性，难以维护，容易出错。

汇编语言

- 使用 0 和 1 这样的机器语言好处是 CPU 认识，可以直接执行，但是对于程序本身来说，没有可读性，难以维护，容易出错。
- 汇编语言，它用助记符（代替操作码指令，用地址符号代替地址码。实际是对机器语言的一种映射，可读性高。

汇编语言

- 使用 0 和 1 这样的机器语言好处是 CPU 认识，可以直接执行，但是对于程序本身来说，没有可读性，难以维护，容易出错。
- 汇编语言，它用助记符（代替操作码指令，用地址符号代替地址码。实际是对机器语言的一种映射，可读性高。

指令	汇编指令	格式	说明
0011	ADD	[var1][var2]	加法操作

高级语言

- 汇编语言的出现大大提高了编程效率，但是有一个问题就是不同 CPU 的指令集可能不同，这样就需要为不同的 CPU 编写不同的汇编程序.

高级语言

- 汇编语言的出现大大提高了编程效率，但是有一个问题就是不同 CPU 的指令集可能不同，这样就需要为不同的 CPU 编写不同的汇编程序。
- 高级语言比如 C，或者是后来的 C++, JAVA, C#，把多条汇编指令合成成为了一个表达式，并且去除了许多操作细节（比如堆栈操作，寄存器操作），而以一种更直观的方式来编写程序，

高级语言

- 汇编语言的出现大大提高了编程效率，但是有一个问题就是不同 CPU 的指令集可能不同，这样就需要为不同的 CPU 编写不同的汇编程序。
- 高级语言比如 C，或者是后来的 C++, JAVA, C#，把多条汇编指令合成成为了一个表达式，并且去除了许多操作细节（比如堆栈操作，寄存器操作），而以一种更直观的方式来编写程序，
- 面向对象的语言的出现使得程序编写更加符合我们的思维方式。我们不必把尽力放到低层的细节上，而更多的关注程序的本身的逻辑的实现。

如何写程序？

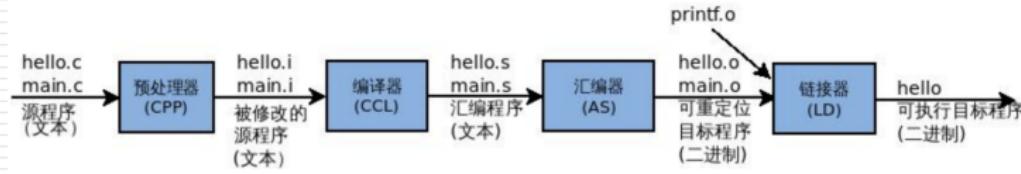
- 文本编辑器编写高级语言并存储为一个文件如，`hello.c`

如何写程序？

- 文本编辑器编写高级语言并存储为一个文件如，hello.c
- 编译器读入文件，按照编译法则将其转换成二进制文件。

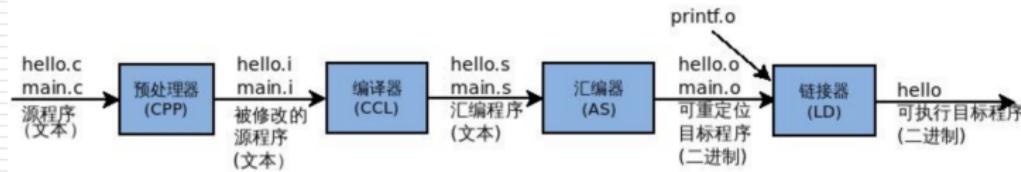
如何写程序?

- 文本编辑器编写高级语言并存储为一个文件如，hello.c
- 编译器读入文件，按照编译法则将其转换成二进制文件.



如何写程序?

- 文本编辑器编写高级语言并存储为一个文件如, hello.c
- 编译器读入文件, 按照编译法则将其转换成二进制文件.



- 练习: 写一个小程序, 编译成二进制文件.

算法

- 计算机算法 (数值算法): 对于特定计算问题的逻辑步骤的集合.

算法

- 计算机算法 (数值算法): 对于特定计算问题的逻辑步骤的集合.
- 一个典型的算法构造:

算法

- 计算机算法 (数值算法): 对于特定计算问题的逻辑步骤的集合.
- 一个典型的算法构造:
 - 假设一个有质量 m 的粒子在力 $f(x)$ 的作用下沿 x 轴运动, 根据牛顿运动方程,

$$f = ma = m \frac{dv}{dt}$$

算法

- 计算机算法 (数值算法): 对于特定计算问题的逻辑步骤的集合.
- 一个典型的算法构造:
 - 假设一个有质量 m 的粒子在力 $f(x)$ 的作用下沿 x 轴运动, 根据牛顿运动方程,

$$f = ma = m \frac{dv}{dt}$$

- 将时间划分为非常小的单元 $\tau = t_{i+1} - t_i$. 在 t_i 时刻的速度由在该小段区间里的平均速度给出:

$$v_i = \frac{x_{i+1} - x_i}{\tau}$$

算法

■ 同样的有

$$a_i \simeq \frac{v_{i+1} - v_i}{\tau}$$

算法

- 同样的有

$$a_i \simeq \frac{v_{i+1} - v_i}{\tau}$$

- 那么对于 t_{i+1} 时刻可以计算出

$$x_{i+1} = x_i + \tau v_i$$

$$v_{i+1} = v_i + \frac{\tau}{m} f(x_i)$$

算法

- 同样的有

$$a_i \simeq \frac{v_{i+1} - v_i}{\tau}$$

- 那么对于 t_{i+1} 时刻可以计算出

$$x_{i+1} = x_i + \tau v_i$$

$$v_{i+1} = v_i + \frac{\tau}{m} f(x_i)$$

- 给定初始位置和速度，可以算出任意时刻的量

第一节

计算物理简介

1.1 计算物理学的起源与发展

1.2 计算与科学

1.3 现代计算机

1.4 程序语言与计算机算法

1.5 误差分析

1.6 浮点数相关

误差分析

■ 误差：误差是近似值与准确值之差

$$E = Z^* - Z$$

误差分析

- 误差：误差是近似值与准确值之差

$$E = Z^* - Z$$

- 误差限：若存在一个小正数，使不等式

$$|E| = |Z^* - Z| \leq \epsilon$$

成立，则称 ϵ 为近似值 Z 的绝对误差限，简称误差限。

误差分析

■ 相对误差：相对误差定义为

$$E_r = \frac{|E|}{|Z^*|} \simeq \frac{|E|}{|Z|}$$

误差分析

■ 相对误差：相对误差定义为

$$E_r = \frac{|E|}{|Z^*|} \simeq \frac{|E|}{|Z|}$$

■ 相对误差限：

$$E_r \leq \frac{\epsilon}{|Z^*|} = \epsilon_r \simeq \frac{\epsilon}{|Z|}$$

误差分析

- 相对误差：相对误差定义为

$$E_r = \frac{|E|}{|Z^*|} \simeq \frac{|E|}{|Z|}$$

- 相对误差限：

$$E_r \leq \frac{\epsilon}{|Z^*|} = \epsilon_r \simeq \frac{\epsilon}{|Z|}$$

- 例：按四舍五入取 π 的近似值 3.14，试求其相对误差限。

有效数字

- 定义 1 如果近似值 Z 的误差限不超过某一位上的半个单位, 该位到 Z 的第一个非零数字共有 n 位, 我们说, Z 有 n 位有效数字.

有效数字

- 定义 1 如果近似值 Z 的误差限不超过某一位上的半个单位, 该位到 Z 的第一个非零数字共有 n 位, 我们说, Z 有 n 位有效数字.
- 定义 2 设近似数 Z 表示为 $Z = 0.x_1x_2\dots x_n \times 10^m$, x_i 取 $0 \sim 9$ 的任意数字, 但 $x_1 \neq 0$, n 为正整数, m 为整数. 若 $|Z^* - Z| \leq \frac{1}{2} \times 10^{m-n}$ 则称 Z 为 Z^* 的具有 n 位有效数字的近似值.

有效数字

■ 例 1，下面关于 π 的有效数字有多少位？

3.14, 3.141, 3.142

有效数字

■ 例 1, 下面关于 π 的有效数字有多少位?

3.14, 3.141, 3.142

■ 例 2, 下列数字各有几位有效数字, 误差限是多少?

2.0004, 0.00200, 9000

9×10^3 , 2×10^{-3}

有效数字

关于有效数字的结论：

- 由测量工具测得的数据都是有效数字.

有效数字

关于有效数字的结论：

- 1 由测量工具测得的数据都是有效数字.
- 2 用四舍五入取准确值的前 n 位都是有效数字.

有效数字

关于有效数字的结论：

- 1 由测量工具测得的数据都是有效数字.
- 2 用四舍五入取准确值的前 n 位都是有效数字.
- 3 由有效数字表示的近似数 3×10^4 和 30000 是不同的.

有效数字

关于有效数字的结论：

- 1 由测量工具测得的数据都是有效数字.
- 2 用四舍五入取准确值的前 n 位都是有效数字.
- 3 由有效数字表示的近似数 3×10^4 和 30000 是不同的.
- 4 准确值被认为有无穷位有效数字.

误差分析

误差来源主要有

- 模型误差

误差分析

误差来源主要有

- 模型误差
- 观测误差

误差分析

误差来源主要有

- 模型误差
- 观测误差
- 截断误差 *

误差分析

误差来源主要有

- 模型误差
- 观测误差
- 截断误差 *
- 舍入误差 *

数值计算应注意的问题

■ 避免相近二数相减

数值计算应注意的问题

- 避免相近二数相减
- 防止大数吃小数

数值计算应注意的问题

- 避免相近二数相减
- 防止大数吃小数
- 避免小分母溢出

数值计算应注意的问题

- 避免相近二数相减
- 防止大数吃小数
- 避免小分母溢出
- 减少运算次数

数值计算应注意的问题

- 避免相近二数相减
- 防止大数吃小数
- 避免小分母溢出
- 减少运算次数
- 正负交替级数累和计算中的问题

第一节

计算物理简介

1.1 计算物理学的起源与发展

1.2 计算与科学

1.3 现代计算机

1.4 程序语言与计算机算法

1.5 误差分析

1.6 浮点数相关

浮点数的表示

- 计算机用二进制表示浮点数 $V = (-1)^S * M * 2^E$ (E 包含偏置量)

浮点数的表示

- 计算机用二进制表示浮点数 $V = (-1)^S * M * 2^E$ (E 包含偏置量)
- 计算机表示的数是有限的. 如 32 位可以表示的数只有 2^{32} 个. 其可以表示数的范围为 (规约形式) $1.175 \times 10^{-38} \simeq 3.402 \times 10^{38}$. 分布不均匀. 有效数字 6-9 位 (十进制).

加速计算

■ 硬件

加速计算

■ 硬件

■ cpu

加速计算

■ 硬件

- cpu
- Intel knl

加速计算

■ 硬件

- cpu
- Intel knl
- Nvidia GPU

加速计算

■ 硬件

- cpu
- Intel knl
- Nvidia GPU
- Google TPU ...

加速计算

■ 硬件

- cpu
- Intel knl
- Nvidia GPU
- Google TPU ...

■ 软件

加速计算

■ 硬件

- cpu
- Intel knl
- Nvidia GPU
- Google TPU ...

■ 软件

- 矢量化运算

加速计算

■ 硬件

- cpu
- Intel knl
- Nvidia GPU
- Google TPU ...

■ 软件

- 矢量化运算
- 并行计算

本章小结

■ 本章小结

本章小结

- 本章小结
- 了解误差，误差限，相对误差，相对误差限的概念

本章小结

- 本章小结
- 了解误差，误差限，相对误差，相对误差限的概念
- 掌握有效数字的概念，知道如何计算出有效数字.

本章小结

- 本章小结
- 了解误差，误差限，相对误差，相对误差限的概念
- 掌握有效数字的概念，知道如何计算出有效数字.
- 了解数值计算中应该注意的问题.

本章小结

- 本章小结
- 了解误差，误差限，相对误差，相对误差限的概念
- 掌握有效数字的概念，知道如何计算出有效数字.
- 了解数值计算中应该注意的问题.
- 理解计算机对浮点数的处理方式及运算法则.

第一节 计算物理简介

第二节 linux 系统编程简介

第三节 Python 语言基础

第四节 Mathematica 基础

第五节 函数近似

第二节

linux 系统编程简介

2.1 Linux 系统

2.2 Linux 初步

2.3 GCC 编译

2.4 C 与 C++

操作系统

- 操作系统：管理计算机的所有活动以及驱动系统中的所有硬件的程序。

操作系统

- 操作系统：管理计算机的所有活动以及驱动系统中的所有硬件的程序.
- 应用程序只需要调用操作系统提供的各种接口就可以使用各类硬件.

操作系统

- 操作系统：管理计算机的所有活动以及驱动系统中的所有硬件的程序。
- 应用程序只需要调用操作系统提供的各种接口就可以使用各类硬件。

1 硬件

操作系统

- 操作系统：管理计算机的所有活动以及驱动系统中的所有硬件的程序.
- 应用程序只需要调用操作系统提供的各种接口就可以使用各类硬件.

- 1 硬件
- 2 操作系统核心

操作系统

- 操作系统：管理计算机的所有活动以及驱动系统中的所有硬件的程序.
- 应用程序只需要调用操作系统提供的各种接口就可以使用各类硬件.
 - 1 硬件
 - 2 操作系统核心
 - 3 系统呼叫 (System Call), 即开发接口

操作系统

- 操作系统：管理计算机的所有活动以及驱动系统中的所有硬件的程序.
- 应用程序只需要调用操作系统提供的各种接口就可以使用各类硬件.
 - 1 硬件
 - 2 操作系统核心
 - 3 系统呼叫 (System Call), 即开发接口
 - 4 应用程序

操作系统

■ 操作系统的核心功能

操作系统

- 操作系统的核心功能
 - 系统呼叫接口

操作系统

■ 操作系统的核心功能

- 系统呼叫接口
- 程序管理

操作系统

■ 操作系统的核心功能

- 系统呼叫接口
- 程序管理
- 内存管理

操作系统

■ 操作系统的核心功能

- 系统呼叫接口
- 程序管理
- 内存管理
- 文件系统管理

操作系统

■ 操作系统的核心功能

- 系统呼叫接口
- 程序管理
- 内存管理
- 文件系统管理
- 硬件的驱动

Linux 系统

- Linux 系统由 Linus Torvalds 在 1991 年开发出来，沿袭 Unix 的许多特性.

Linux 系统

- Linux 系统由 Linus Torvalds 在 1991 年开发出来，沿袭 Unix 的许多特性。
- Linux 系统只是一个操作系统，因其开源的特性，且运行效率很快，受到了大家的欢迎。

Linux 系统

- Linux 系统由 Linus Torvalds 在 1991 年开发出来，沿袭 Unix 的许多特性。
- Linux 系统只是一个操作系统，因其开源的特性，且运行效率很快，受到了大家的欢迎。
- GNU 软件：由 GNU 计划开发的开源软件

Linux 系统

- Linux 系统由 Linus Torvalds 在 1991 年开发出来，沿袭 Unix 的许多特性。
- Linux 系统只是一个操作系统，因其开源的特性，且运行效率很快，受到了大家的欢迎。
- GNU 软件：由 GNU 计划开发的开源软件
 - Emacs

Linux 系统

- Linux 系统由 Linus Torvalds 在 1991 年开发出来，沿袭 Unix 的许多特性。
- Linux 系统只是一个操作系统，因其开源的特性，且运行效率很快，受到了大家的欢迎。
- GNU 软件：由 GNU 计划开发的开源软件
 - Emacs
 - GNU C (GCC)

Linux 系统

- Linux 系统由 Linus Torvalds 在 1991 年开发出来，沿袭 Unix 的许多特性。
- Linux 系统只是一个操作系统，因其开源的特性，且运行效率很快，受到了大家的欢迎。
- GNU 软件：由 GNU 计划开发的开源软件
 - Emacs
 - GNU C (GCC)
 - GNU C Library (glibc)

Linux 系统

- Linux 系统由 Linus Torvalds 在 1991 年开发出来，沿袭 Unix 的许多特性。
- Linux 系统只是一个操作系统，因其开源的特性，且运行效率很快，受到了大家的欢迎。
- GNU 软件：由 GNU 计划开发的开源软件
 - Emacs
 - GNU C (GCC)
 - GNU C Library (glibc)
 - Bash shell

GNU 的 GPL 协议

- GNU 提出的自由软件版权 GPL

GNU 的 GPL 协议

- GNU 提出的自由软件版权 GPL
 - 取得软件与原代码;

GNU 的 GPL 协议

- GNU 提出的自由软件版权 GPL
 - 取得软件与原代码;
 - 复制;

GNU 的 GPL 协议

■ GNU 提出的自由软件版权 GPL

- 取得软件与原代码;
- 复制;
- 修改;

GNU 的 GPL 协议

■ GNU 提出的自由软件版权 GPL

- 取得软件与原代码;
- 复制;
- 修改;
- 再发行;

Linux distributions

- Linux 核心采用 GNU GPL 授权，任何人都可以取得并修改它.

Linux distributions

- Linux 核心采用 GNU GPL 授权，任何人都可以取得并修改它.
- GNU 软件大多以 Linux 为主要操作系统来开发.

Linux distributions

- Linux 核心采用 GNU GPL 授权，任何人都可以取得并修改它.
- GNU 软件大多以 Linux 为主要操作系统来开发.
- 许多其他自由软件团队也逐渐采用 Linux 系统.

Linux distributions

- Linux 核心采用 GNU GPL 授权，任何人都可以取得并修改它。
- GNU 软件大多以 Linux 为主要操作系统来开发。
- 许多其他自由软件团队也逐渐采用 Linux 系统。
- 因此许多商业公司和非营利团体将 Linux 核心和一整套的自由软件整合一起，形成一个 Linux distribution. Ubuntu, Fedora, SuSE, CentOS

Linux 系统特点

■ 稳定的系统

Linux 系统特点

- 稳定的系统
- 基本上免费

Linux 系统特点

- 稳定的系统
- 基本上免费
- 安全性、漏洞的快速修补

Linux 系统特点

- 稳定的系统
- 基本上免费
- 安全性、漏洞的快速修补
- 多任务、多使用者

Linux 系统特点

- 稳定的系统
- 基本上免费
- 安全性、漏洞的快速修补
- 多任务、多使用者
- 使用者与群组的规划

Linux 系统特点

- 稳定的系统
- 基本上免费
- 安全性、漏洞的快速修补
- 多任务、多使用者
- 使用者与群组的规划
- 相对比较不耗资源的系统

Linux 系统特点

- 稳定的系统
- 基本上免费
- 安全性、漏洞的快速修补
- 多任务、多使用者
- 使用者与群组的规划
- 相对比较不耗资源的系统
- 整合度佳且多样的图形用户接口

第二节

linux 系统编程简介

2.1 Linux 系统

2.2 Linux 初步

2.3 GCC 编译

2.4 C 与 C++

目录与路径

- 绝对路径：由根目录下写起，如`/usr/share/`

目录与路径

- 绝对路径：由根目录下写起，如/usr/share/
 - 绝对路径不容易出错，建议在脚本或程序中使用。

目录与路径

- 绝对路径：由根目录下写起，如/usr/share/
 - 绝对路径不容易出错，建议在脚本或程序中使用。
- 相对路径：相对当前的路径，如..//xxx, 相对当前目录的同级 xxx 目录。

目录与路径

- 绝对路径：由根目录下写起，如/usr/share/
 - 绝对路径不容易出错，建议在脚本或程序中使用。
- 相对路径：相对当前的路径，如..//xxx, 相对当前目录的同级 xxx 目录。
 - 方便在不同目录下切换

目录与路径

- 1 . 代表此层目录
- 2 .. 代表上一层目录
- 3 ~ 代表当前用户的家目录

常见处理目录的指令

■ cd: 变换目录

常见处理目录的指令

- cd: 变换目录
- pwd: 显示当前目录

常见处理目录的指令

- cd: 变换目录
- pwd: 显示当前目录
- mkdir: 建立一个新的目录

常见处理目录的指令

- cd: 变换目录
- pwd: 显示当前目录
- mkdir: 建立一个新的目录
- rmdir: 删除一个空的目录

常见处理目录的指令

- cd: 变换目录
- pwd: 显示当前目录
- mkdir: 建立一个新的目录
- rmdir: 删除一个空的目录
- \$PATH 里面包含了系统去寻找可执行文件的所有地址

文件管理

- ls: 显示当前目录下的文件信息

文件管理

- ls: 显示当前目录下的文件信息
- cp: 复制文件

文件管理

- ls: 显示当前目录下的文件信息
- cp: 复制文件
- rm: 删除文件

文件管理

- ls: 显示当前目录下的文件信息
- cp: 复制文件
- rm: 删除文件
- mv: 移动文件

文件管理

- ls: 显示当前目录下的文件信息
- cp: 复制文件
- rm: 删除文件
- mv: 移动文件
- cat: 将文件全部打印到屏幕

文件管理

- ls: 显示当前目录下的文件信息
- cp: 复制文件
- rm: 删除文件
- mv: 移动文件
- cat: 将文件全部打印到屏幕
- more, less: 一页一页翻动

目录与文件的权限

- 目录和文件都有带有权限信息

目录与文件的权限

- 目录和文件都有带有限权限信息
 - r 可读

目录与文件的权限

■ 目录和文件都有带有限权限信息

- r 可读
- w 可写

目录与文件的权限

■ 目录和文件都有带有限权限信息

- r 可读
- w 可写
- x 可执行

目录与文件的权限

- 目录和文件都有带有权限信息
 - r 可读
 - w 可写
 - x 可执行
- 权限分为文件所有者、所属组、其他人三部分的权限.

目录与文件的权限

- 目录和文件都有带有权限信息
 - r 可读
 - w 可写
 - x 可执行
- 权限分为文件所有者、所属组、其他人三部分的权限.
- linux 下的一些文件编辑器： vim, Emacs, 其他

应用程序的安装

- Linux 下程序的安装非常简单

应用程序的安装

- Linux 下程序的安装非常简单
- 不同发行版本有不同的程序管理套件

应用程序的安装

- Linux 下程序的安装非常简单
- 不同发行版本有不同的程序管理套件
 - apt-get (ubuntu,debian)

应用程序的安装

- Linux 下程序的安装非常简单
- 不同发行版本有不同的程序管理套件
 - apt-get (ubuntu,debian)
 - yum, dnf (centos, fedora)

应用程序的安装

- Linux 下程序的安装非常简单
- 不同发行版本有不同的程序管理套件
 - apt-get (ubuntu,debian)
 - yum, dnf (centos, fedora)
 - pacman (mingw)

第二节

linux 系统编程简介

2.1 Linux 系统

2.2 Linux 初步

2.3 GCC 编译

2.4 C 与 C++

GCC

- GCC 是由 GNU 开发的免费开源的 c 语言编译套件.

GCC

- GCC 是由 GNU 开发的免费开源的 c 语言编译套件.
- GCC 包含 gcc, g++,gfortran 等支持 c, c++ 及 fortran 语言的编译.

GCC

- GCC 是由 GNU 开发的免费开源的 c 语言编译套件.
- GCC 包含 gcc, g++,gfortran 等支持 c, c++ 及 fortran 语言的编译.
- gcc 的常用编译选项-E, -S, -c, -o, -I

GCC

- GCC 是由 GNU 开发的免费开源的 c 语言编译套件.
- GCC 包含 gcc, g++,gfortran 等支持 c, c++ 及 fortran 语言的编译.
- gcc 的常用编译选项-E, -S, -c, -o, -l
- 单个文件的编译

1 **gcc filename.c -o exefilename**

GCC

- 多个文件的编译，首先对每个文件编译成二进制文件。

```
1 gcc -c file1.c -o file1.o  
2 gcc -c file2.c -o file2.o  
3 ...  
4 gcc main.c file1.o file2.o -o  
    finalexe
```

链接库

- 常用的程序打包成一个程序库，如 lapcak, blas 等数学库。

链接库

- 常用的程序打包成一个程序库，如 lapcak, blas 等数学库。
- 动态链接库：等程序运行时，去读取，不会编译到可执行程序当中。如 libblas.so

链接库

- 常用的程序打包成一个程序库，如 lapcak, blas 等数学库。
- 动态链接库：等程序运行时，去读取，不会编译到可执行程序当中。如 libblas.so
- 静态链接库：直接编译到可执行程序当中，方便移植。如 libblas.a

链接库

- 常用的程序打包成一个程序库，如 lapcak, blas 等数学库。
- 动态链接库：等程序运行时，去读取，不会编译到可执行程序当中。如 libblas.so
- 静态链接库：直接编译到可执行程序当中，方便移植。如 libblas.a
- 链接库需要指定寻找的目录，设在 lib 目录下有 libfoo.so 库，则

```
1 gcc foo.c -L /home/lib -lfoo  
      -o foo
```

生成链接库

- 静态链接库，假设已经将文件编译成.o 文件

```
1 ar -crv libstaticmath.a  
StaticMath.o
```

生成链接库

- 静态链接库，假设已经将文件编译成.o 文件

```
1 ar -crv libstaticmath.a  
      StaticMath.o
```

- 动态链接库的生成

```
1 gcc -shared -fPCI -o libxxx.  
      so xxx1.o xxx2.o ...
```

库地址

- 在编译时使用-L 是为 gcc 指定寻找这些库的地址.
也可以通过系统的环境变量 `$LIBRARY_PATH` 来指定

库地址

- 在编译时使用-L 是为 gcc 指定寻找这些库的地址.
也可以通过系统的环境变量 `$LIBRARY_PATH` 来指定
- 但是在运行时，很可能找不到库. 解决办法：

库地址

- 在编译时使用-L 是为 gcc 指定寻找这些库的地址.
也可以通过系统的环境变量 `$LIBRARY_PATH` 来指定
- 但是在运行时，很可能找不到库. 解决办法：
 - 编译时加入-Wl,rpath=/xx/xxx 选项

库地址

- 在编译时使用-L 是为 gcc 指定寻找这些库的地址.
也可以通过系统的环境变量 `$LIBRARY_PATH` 来指定
- 但是在运行时，很可能找不到库. 解决办法：
 - 编译时加入-Wl,rpath=/xx/xxx 选项
 - 或加入 `$LD_LIBRARY_PATH` 环境变量

Makefile

■ makefile: 自动化编译

```
1 main: main.o add.o sub.o
2 main.o: main.c
3         gcc -c main.c -o main.o
4 add.o: add.c
5         gcc -c add.c -o add.o #
6             加-c 指定生成为可重链接
7             .o文件
8 sub.o: sub.c
9         gcc -c sub.c -o sub.o
10 .PHONY: clean
11 clean:
12         -rm -rf *.o
```

第二节

linux 系统编程简介

2.1 Linux 系统

2.2 Linux 初步

2.3 GCC 编译

2.4 C 与 C++

C 语言回顾

■ C 语言算术运算符: +, -, *, /, %, ++, -

C 语言回顾

- C 语言算术运算符: +, -, *, /, %, ++, -
- 关系运算符: ==, !=, >, <, >=, <=

C 语言回顾

- C 语言算术运算符: +, -, *, /, %, ++, -
- 关系运算符: ==, !=, >, <, >=, <=
- 逻辑运算符: &&, ||, !

C 语言回顾

- C 语言算术运算符: +, -, *, /, %, ++, -
- 关系运算符: ==, !=, >, <, >=, <=
- 逻辑运算符: &&, ||, !
- 运算符的优先级: 尽量用 () 来给出需要的运算优先级.

C 语言回顾

- C 语言算术运算符: +, -, *, /, %, ++, -
- 关系运算符: ==, !=, >, <, >=, <=
- 逻辑运算符: &&, ||, !
- 运算符的优先级: 尽量用 () 来给出需要的运算优先级.
- 指针与取地址

C 语言回顾

■ 判断语句：

C 语言回顾

- 判断语句：
 - if 语句

C 语言回顾

■ 判断语句：

- if 语句
- if...else

C 语言回顾

■ 判断语句：

- if 语句
- if...else
- switch 语句

C 语言回顾

■ 判断语句：

- if 语句
- if...else
- switch 语句
- ?: 运算符

C 语言回顾

■ 循环语句

C 语言回顾

■ 循环语句

■ while 循环

C 语言回顾

■ 循环语句

- while 循环
- for 循环

C 语言回顾

■ 循环语句

- while 循环
- for 循环
- do...wihle 循环

C 语言回顾

■ 循环语句

- while 循环
- for 循环
- do...while 循环
- break

C 语言回顾

■ 循环语句

- while 循环
- for 循环
- do...while 循环
- break
- continue

C 语言回顾

■ 循环语句

- while 循环
- for 循环
- do...while 循环
- break
- continue
- goto

C 语言回顾

■ 函数：函数返回类型、函数名称、参数、函数主体

C 语言回顾

- 函数：函数返回类型、函数名称、参数、函数主体
- 关键字 static 定义的函数只能在本文件中被其他函数调用

C 语言回顾

- 函数：函数返回类型、函数名称、参数、函数主体
- 关键字 static 定义的函数只能在本文件中被其他函数调用
- 函数的传值调用和引用调用

C 语言回顾

- 函数：函数返回类型、函数名称、参数、函数主体
- 关键字 static 定义的函数只能在本文件中被其他函数调用
- 函数的传值调用和引用调用
 - 传值调用：函数内对参数的改变，不会传入的参数。参数为变量

C 语言回顾

- 函数：函数返回类型、函数名称、参数、函数主体
- 关键字 static 定义的函数只能在本文件中被其他函数调用
- 函数的传值调用和引用调用
 - 传值调用：函数内对参数的改变，不会传入的参数。参数为变量
 - 引用调用：传入参数为变量的地址，会改变传入的变量的值。参数为指针

C++ 类

- C++ 引入了命名空间，来减少同名变量的冲突.

C++ 类

- C++ 引入了命名空间，来减少同名变量的冲突.
- C++ 增加了类这一用户定义的类型，使得其具有面向对象的特性.

C++ 类

- C++ 引入了命名空间，来减少同名变量的冲突.
- C++ 增加了类这一用户定义的类型，使得其具有面向对象的特性.
- 类定义是以关键字 class 开头:

```
1 class Box {  
2 public:  
3     double length; // 盒子的长度  
4     double breadth; // 盒子的宽  
      度  
5     double height; // 盒子的高度  
6 };
```

C++ 类对象

- 有了类的定义，就可以声明类的对象

```
1 Box Box1;  
2 Box Box2;
```

C++ 类对象

- 有了类的定义，就可以声明类的对象

1 **Box Box1;**

2 **Box Box2;**

- 通过对象可以访问其公共数据成员

1 **Box1.height=7;**

2 **Box2.height=6;**

C++ 访问对象的成员

- 类除了有公共成员外，还有类成员函数，成员函数可以操作类的任意对象，可以访问对象中的所有成员。

C++ 访问对象的成员

- 类除了有公共成员外，还有类成员函数，成员函数可以操作类的任意对象，可以访问对象中的所有成员。
- 使用成员函数，通过对对象的. 运算符来调用

```
1 class Box{  
2     public:  
3         double length;  
4         double getLength(){  
5             return length; } ; } ;  
6 Box box1;  
7 box1.getLength();
```

C++ 构造函数

- 类中的几个特殊的函数：类的构造函数、析构函数、拷贝构造函数.

C++ 构造函数

- 类中的几个特殊的函数：类的构造函数、析构函数、拷贝构造函数。
- 构造函数在类实例化对象时，被调用。构造函数名称与类名称相同，不会返回任何类型，可以在构造函数中为成员函数设置初始值。

C++ 构造函数

```
1 class Line {  
2     public:  
3         Line(double len); // 这是构造  
4             函数  
5     private:  
6         double length; };  
7 Line::Line( double len ) {  
8     cout << "Object is being  
         created, length = " << len  
         << endl;  
9     length = len; }
```

C++ 析构函数

- 类的析构函数是类的一种特殊的成员函数，它会在每次删除所创建的对象时执行.

C++ 析构函数

- 类的析构函数是类的一种特殊的成员函数，它会在每次删除所创建的对象时执行.
- 析构函数的名称与类的名称是完全相同的，只是在前面加了个波浪号 (~) 作为前缀，它不会返回任何值，也不能带有任何参数. 析构函数有助于在跳出程序（比如关闭文件、释放内存等）前释放资源.

C++ 拷贝构造函数

- 拷贝构造函数是一种特殊的构造函数，它在创建对象时，是使用同一类中之前创建的对象来初始化新创建的对象。拷贝构造函数通常用于：

C++ 拷贝构造函数

- 拷贝构造函数是一种特殊的构造函数，它在创建对象时，是使用同一类中之前创建的对象来初始化新创建的对象。拷贝构造函数通常用于：
 - 通过使用另一个同类型的对象来初始化新创建的对象。

C++ 拷贝构造函数

- 拷贝构造函数是一种特殊的构造函数，它在创建对象时，是使用同一类中之前创建的对象来初始化新创建的对象。拷贝构造函数通常用于：
 - 通过使用另一个同类型的对象来初始化新创建的对象。
 - 复制对象把它作为参数传递给函数。

C++ 拷贝构造函数

- 拷贝构造函数是一种特殊的构造函数，它在创建对象时，是使用同一类中之前创建的对象来初始化新创建的对象。拷贝构造函数通常用于：
 - 通过使用另一个同类型的对象来初始化新创建的对象。
 - 复制对象把它作为参数传递给函数。
 - 复制对象，并从函数返回这个对象。

```
1 classname (const classname &obj)  
2 {      // 构造函数的主体  
3 }
```

C++ 类的派生

- 面向对象程序设计中最重要的一个概念是继承。继承允许我们依据另一个类来定义一个类

C++ 类的派生

- 面向对象程序设计中最重要的一个概念是继承。继承允许我们依据另一个类来定义一个类
- 新建的类继承了一个已有的类的成员，这个已有的类称为基类，新建的类称为派生类。

C++ 类的派生

- 面向对象程序设计中最重要的一个概念是继承。继承允许我们依据另一个类来定义一个类
- 新建的类继承了一个已有的类的成员，这个已有的类称为基类，新建的类称为派生类。
- 一个类可以派生自多个类

1 **class** <派生类名>:<继承方式1><
基类名1>,<继承方式2><基类名
2>, ...

C++ 函数重载

- C++ 允许在同一作用域中的某个函数和运算符指定多个定义，分别称为函数重载和运算符重载。

```
1 class printData
2 {public:
3     void print(int i) {
4         cout << "整数为：" << i
5             << endl;}
6     void print(double f) {
7         cout << "浮点数为：" <<
8             f << endl;}
9     void print(char c[]) {
10        cout << "字符串为：" <<
11            c << endl;} }
```

C++ 运算符重载

- 可以重定义或重载大部分 C++ 内置的运算符.

```
1 Box operator+(const Box& b){  
2     Box box;  
3     box.length = this->length +  
4         b.length;  
5     box.breadth = this->breadth  
6         + b.breadth;  
7     box.height = this->height +  
8         b.height;  
9     return box; }
```

C++ 多态

- 当类之间存在层次结构，并且类之间是通过继承关联时，就会用到多态.

C++ 多态

- 当类之间存在层次结构，并且类之间是通过继承关联时，就会用到多态.
- C++ 多态意味着调用成员函数时，会根据调用函数的对象的类型来执行不同的函数.

C++ 多态

- 当类之间存在层次结构，并且类之间是通过继承关联时，就会用到多态.
- C++ 多态意味着调用成员函数时，会根据调用函数的对象的类型来执行不同的函数.
- 虚函数是在基类中使用关键字 `virtual` 声明的函数. 在派生类中重新定义基类中定义的虚函数时，会告诉编译器不要静态链接到该函数.

C++ 多态

- 当类之间存在层次结构，并且类之间是通过继承关联时，就会用到多态.
- C++ 多态意味着调用成员函数时，会根据调用函数的对象的类型来执行不同的函数.
- 虚函数是在基类中使用关键字 `virtual` 声明的函数. 在派生类中重新定义基类中定义的虚函数时，会告诉编译器不要静态链接到该函数.
- = 0 告诉编译器，函数没有主体，上面的虚函数是纯虚函数.

C++ 模板

- 模板是泛型编程的基础，泛型编程即以一种独立于任何特定类型的方式编写代码.

C++ 模板

- 模板是泛型编程的基础，泛型编程即以一种独立于任何特定类型的方式编写代码.
- 模板函数定义的一般形式如下所示：

```
1 template <class type> ret-
    type func-name(parameter
list)

2 {
3     // 函数的主体
4 }
```

C++ 模板

- 正如我们定义函数模板一样，我们也可以定义类模板。

```
1 template <class type> class  
    class-name {  
2     .  
3     .  
4     .  
5 }
```

第一节 计算物理简介

第二节 linux 系统编程简介

第三节 Python 语言基础

第四节 Mathematica 基础

第五节 函数近似

第三节

Python 语言基础

3.1

基本语法

3.2

列表简介

3.3

函数

3.4

numpy 包的使用

基本语法

- Python 是一门解释型语言，无需编译和链接

基本语法

- Python 是一门解释型语言，无需编译和链接
- 语句组使用缩进替代开始和结束大括号来组织

基本语法

- Python 是一门解释型语言，无需编译和链接
- 语句组使用缩进替代开始和结束大括号来组织
- 变量或参数无需声明

基本语法

- Python 是一门解释型语言，无需编译和链接
- 语句组使用缩进替代开始和结束大括号来组织
- 变量或参数无需声明
- 有更强的表达能力

基本语法

■ C

```
1 #include<stdio.h>
2 #include "math.h"
3 void main()
4 {double x,y;
5 scanf("%f %f",&x,&y)
6 printf("Sin(x) + Sin(y) is %f\n",sin(x)+sin(y))}
```

基本语法

■ C

```
1 #include<stdio.h>
2 #include "math.h"
3 void main()
4 {double x,y;
5 scanf("%f %f",&x,&y)
6 printf("Sin(x) + Sin(y) is %f\n",sin(x)+sin(y))}
```

■ python

```
1 import math
2 a,b = eval(input())
3 print("Sin(x) + Sin(y) is ",math.sin(x) + math.sin(y))
```

变量

■ 变量命名和使用

变量

■ 变量命名和使用

- 包含字母、数字和下划线. 不能以数字开头

变量

■ 变量命名和使用

- 包含字母、数字和下划线. 不能以数字开头
- 不能用 python 关键字和函数名用作变量名

变量

■ 变量命名和使用

- 包含字母、数字和下划线. 不能以数字开头
- 不能用 python 关键字和函数名用作变量名
- 变量命名应简短又具有描述性

变量

■ 变量命名和使用

- 包含字母、数字和下划线. 不能以数字开头
- 不能用 python 关键字和函数名用作变量名
- 变量命名应简短又具有描述性
- 慎用小写字母 l 和大写字母 O

变量

■ 变量命名和使用

- 包含字母、数字和下划线. 不能以数字开头
- 不能用 python 关键字和函数名用作变量名
- 变量命名应简短又具有描述性
- 慎用小写字母 l 和大写字母 O

■ 可能出现的错误

变量

■ 变量命名和使用

- 包含字母、数字和下划线. 不能以数字开头
- 不能用 python 关键字和函数名用作变量名
- 变量命名应简短又具有描述性
- 慎用小写字母 l 和大写字母 O

■ 可能出现的错误

- 定义变量后，使用时拼写错误

变量

■ 变量命名和使用

- 包含字母、数字和下划线. 不能以数字开头
- 不能用 python 关键字和函数名用作变量名
- 变量命名应简短又具有描述性
- 慎用小写字母 l 和大写字母 O

■ 可能出现的错误

- 定义变量后，使用时拼写错误
- 试一试：将一条消息赋值给一个变量，并打印出来. 更改后再次打印.

字符串

- 用引号括起来的都是字符串，可以是单引号或双引号

字符串

- 用引号括起来的都是字符串，可以是单引号或双引号
- 用 + 号来合并字符串

字符串

- 用引号括起来的都是字符串，可以是单引号或双引号
- 用 + 号来合并字符串
- 小技巧：通过不同字符串变量来组合想要打印的消息

字符串

- 用引号括起来的都是字符串，可以是单引号或双引号
- 用 + 号来合并字符串
- 小技巧：通过不同字符串变量来组合想要打印的消息
- python2 与 python3 的 print 语句有区别

字符串

- 用引号括起来的都是字符串，可以是单引号或双引号
- 用 + 号来合并字符串
- 小技巧：通过不同字符串变量来组合想要打印的消息
- python2 与 python3 的 print 语句有区别
- 试一试：试试不同引号的 print 语句

数字

- 整数的加减乘除乘方操作 (python2 与 python3 对/号处理有区别)

数字

- 整数的加减乘除乘方操作 (python2 与 python3 对/号处理有区别)
- 浮点数：带小数点，注意结果可能并不精确

数字

- 整数的加减乘除乘方操作 (python2 与 python3 对/号处理有区别)
- 浮点数：带小数点，注意结果可能并不精确
- 当心类型错误，如

```
1 age=23  
2 print("Happy" + age + "rd Birthday!")
```

数字

- 整数的加减乘除乘方操作 (python2 与 python3 对/号处理有区别)
- 浮点数：带小数点，注意结果可能并不精确
- 当心类型错误，如

```
1 age=23  
2 print("Happy" + age + "rd Birthday!")
```

- 数据类型转化函数：str(), int(), float()

第三节

Python 语言基础

3.1 基本语法

3.2 列表简介

3.3 函数

3.4 numpy 包的使用

列表是什么

- 列表由一系列按特定顺序排列的元素组成，用方括号 [] 来表示列表，逗号来分隔元素

```
1 students = [ 'zhangsan', 'lisi', 'wangwu', 'zhaoliu' ]
```

列表是什么

- 列表由一系列按特定顺序排列的元素组成，用方括号 [] 来表示列表，逗号来分隔元素

```
1 students = [ 'zhangsan', 'lisi', 'wangwu', 'zhaoliu' ]
```

- 通过索引来访问列表中的元素，元素指标从 0 开始

列表是什么

- 列表由一系列按特定顺序排列的元素组成，用方括号 [] 来表示列表，逗号来分隔元素

```
1 students = [ 'zhangsan', 'lisi', 'wangwu', 'zhaoliu' ]
```

- 通过索引来访问列表中的元素，元素指标从 0 开始
- 试试看指标输入负值会是什么样子？

列表是什么

- 列表由一系列按特定顺序排列的元素组成，用方括号 [] 来表示列表，逗号来分隔元素

```
1 students = [ 'zhangsan', 'lisi', 'wangwu', 'zhaoliu' ]
```

- 通过索引来访问列表中的元素，元素指标从 0 开始
- 试试看指标输入负值会是什么样子？
- 列表的元素可以像其它变量一样使用

列表是什么

- 列表由一系列按特定顺序排列的元素组成，用方括号 [] 来表示列表，逗号来分隔元素

```
1 students = [ 'zhangsan', 'lisi', 'wangwu', 'zhaoliu' ]
```

- 通过索引来访问列表中的元素，元素指标从 0 开始
- 试试看指标输入负值会是什么样子？
- 列表的元素可以像其它变量一样使用
- 修改列表：将新的值覆盖需要改变的元素

增减元素

- 在列表末尾添加元素：append() 方法

增减元素

- 在列表末尾添加元素：append() 方法
 - 试一试，对一个空列表添加各类元素

增减元素

- 在列表末尾添加元素：append() 方法
 - 试一试，对一个空列表添加各类元素
- 在列表中插入元素：insert() 方法

增减元素

- 在列表末尾添加元素：append() 方法
 - 试一试，对一个空列表添加各类元素
- 在列表中插入元素：insert() 方法
 - 需要指定元素的索引和值，试一试插入元素的操作

增减元素

- 在列表末尾添加元素：append() 方法
 - 试一试，对一个空列表添加各类元素
- 在列表中插入元素：insert() 方法
 - 需要指定元素的索引和值，试一试插入元素的操作
- 在列表中删除元素：del 语句

```
1 del bicycles[0]
```

增减元素

- 在列表末尾添加元素：append() 方法
 - 试一试，对一个空列表添加各类元素
 - 在列表中插入元素：insert() 方法
 - 需要指定元素的索引和值，试一试插入元素的操作
 - 在列表中删除元素：del 语句
- 1 `del bicycles[0]`
- 可选择的使用：pop() 方法或 remove() 方法

增减元素

- 在列表末尾添加元素：append() 方法
 - 试一试，对一个空列表添加各类元素
- 在列表中插入元素：insert() 方法
 - 需要指定元素的索引和值，试一试插入元素的操作
- 在列表中删除元素：del 语句
 - 1 `del bicycles[0]`
- 可选择的使用：pop() 方法或 remove() 方法
 - pop 与 del 的区别是你可以拿到删除的元素

增减元素

- 在列表末尾添加元素：append() 方法
 - 试一试，对一个空列表添加各类元素
- 在列表中插入元素：insert() 方法
 - 需要指定元素的索引和值，试一试插入元素的操作
- 在列表中删除元素：del 语句
 - 1 `del bicycles[0]`
- 可选择的使用：pop() 方法或 remove() 方法
 - pop 与 del 的区别是你可以拿到删除的元素
 - remove 通过传入值来删除（只删除第一个找到的元素）

列表排序

■ 排序

列表排序

■ 排序

- `sort()` 方法永久对列表排序, `reverse=True` 反向排序

列表排序

■ 排序

- `sort()` 方法永久对列表排序, `reverse=True` 反向排序
- `sorted()` 函数临时排序

列表排序

■ 排序

- `sort()` 方法永久对列表排序, `reverse=True` 反向排序
- `sorted()` 函数临时排序
- `reverse()`方法与原顺序反向排列

列表排序

■ 排序

- `sort()` 方法永久对列表排序, `reverse=True` 反向排序
- `sorted()` 函数临时排序
- `reverse()`方法与原顺序反向排列

■ `len()` 函数获取列表长度

列表排序

■ 排序

- `sort()` 方法永久对列表排序, `reverse=True` 反向排序
- `sorted()` 函数临时排序
- `reverse()`方法与原顺序反向排列

■ `len()` 函数获取列表长度

■ 列表访问容易出现索引错误

历遍列表

■ 使用 for 循环历遍列表

```
1 for student in students:  
2     print(student)
```

历遍列表

■ 使用 for 循环历遍列表

```
1 for student in students:  
2     print(student)
```

■ 方便对列表每个元素进行操作

历遍列表

■ 使用 for 循环历遍列表

```
1 for student in students:  
2     print(student)
```

- 方便对列表每个元素进行操作
- 注意缩进划分代码块，放入循环中的代码需要缩进，循环外的代码不用缩进

历遍列表

■ 使用 for 循环历遍列表

```
1 for student in students:  
2     print(student)
```

- 方便对列表每个元素进行操作
- 注意缩进划分代码块，放入循环中的代码需要缩进，循环外的代码不用缩进
- 不要忘记 for 语句末尾的冒号！

操作列表

■ 列表解析:

```
1 squares = [value**2 for value in range(1,11)]
```

操作列表

■ 列表解析:

```
1 squares = [value**2 for value in range(1,11)]
```

■ 切片: List[k:m] 取出只包含选定索引的列表

```
1 squares[1:2]
2 squares[:2]
3 squares[-3:-1]
```

操作列表

■ 列表解析:

```
1 squares = [value**2 for value in range(1,11)]
```

■ 切片: List[k:m] 取出只包含选定索引的列表

```
1 squares[1:2]
2 squares[:2]
3 squares[-3:-1]
```

■ 复制列表: $listA = listB$ 和 $listA = listB[:]$ 的区别

其它可迭代对象

■ 元组

其它可迭代对象

■ 元组

- 不可变的列表，用圆括号来表示

其它可迭代对象

■ 元组

- 不可变的列表，用圆括号来表示
- 如果一组值在程序整个生命周期不变，可使用元组

其它可迭代对象

■ 元组

- 不可变的列表，用圆括号来表示
- 如果一组值在程序整个生命周期不变，可使用元组

■ 字典

其它可迭代对象

■ 元组

- 不可变的列表，用圆括号来表示
- 如果一组值在程序整个生命周期不变，可使用元组

■ 字典

- 由放在花括号的键值对组成

```
1 alien_0 = { 'color': 'green', 'points': 5}
```

其它可迭代对象

■ 元组

- 不可变的列表，用圆括号来表示
- 如果一组值在程序整个生命周期不变，可使用元组

■ 字典

- 由放在花括号的键值对组成

```
1 alien_0 = { 'color': 'green', 'points': 5}
```

- 通过键访问对应值，也通过键来修改其值

其它可迭代对象

■ 元组

- 不可变的列表，用圆括号来表示
- 如果一组值在程序整个生命周期不变，可使用元组

■ 字典

- 由放在花括号的键值对组成

```
1 alien_0 = { 'color': 'green', 'points': 5}
```

- 通过键访问对应值，也通过键来修改其值
- 如何添加键-值对

```
1 alien_0[ 'x_position' ] = 0
```

其它可迭代对象

■ 元组

- 不可变的列表，用圆括号来表示
- 如果一组值在程序整个生命周期不变，可使用元组

■ 字典

- 由放在花括号的键值对组成

```
1 alien_0 = { 'color': 'green', 'points': 5}
```

- 通过键访问对应值，也通过键来修改其值
- 如何添加键-值对

```
1 alien_0[ 'x_position' ] = 0
```

- del 命令来删除键-值对

第三节

Python 语言基础

3.1 基本语法

3.2 列表简介

3.3 函数

3.4 numpy 包的使用

定义函数

- 关键字 def 来定义函数，后接函数名 + 括号加冒号，注意缩进

```
1 def greet_user():
2     """显示简单的问候语"""
3     print("Hello!")
4 greet_user()
```

定义函数

- 关键字 def 来定义函数，后接函数名 + 括号加冒号，注意缩进

```
1 def greet_user():
2     """显示简单的问候语"""
3     print("Hello!")
4 greet_user()
```

- 传入参数 (回忆 C 语言中的形参和实参)

```
1 def greet_user(username):
2     """显示简单的问候语"""
3     print("Hello " + username + "!")
4 greet_user("zhangsan")
```

定义函数

- 关键字 def 来定义函数，后接函数名 + 括号加冒号，注意缩进

```
1 def greet_user():
2     """显示简单的问候语"""
3     print("Hello!")
4 greet_user()
```

- 传入参数 (回忆 C 语言中的形参和实参)

```
1 def greet_user(username):
2     """显示简单的问候语"""
3     print("Hello " + username + "!")
4 greet_user("zhangsan")
```

- return 关键字后面跟函数返回值

函数参数

■ 位置实参：按形参顺序对应传入实参

```
1 def describe_pet(animal_type, pet_name):  
2     """显示宠物的信息"""  
3     print("\nI have a " + animal_type + " . Its name  
        is " + pet_name)  
4 describe_pet('dog', 'wangwang')
```

函数参数

■ 位置实参：按形参顺序对应传入实参

```
1 def describe_pet(animal_type, pet_name):  
2     """显示宠物的信息"""  
3     print("\nI have a " + animal_type + " . Its name  
is " + pet_name)  
4 describe_pet('dog', 'wangwang')
```

■ 关键字实参：对形参指定实参

```
1 describe_pet(pet_name= 'wangwang', animal_type = 'dog')
```

函数参数

■ 默认值

```
1 def describe_pet( pet_name, animal_type="dog"):  
2     """显示宠物的信息"""  
3     print("\nI have a " + animal_type +  
4         ". " + " Its name is " + pet_name)  
5 describe_pet( 'wangwang')  
6 describe_pet( 'miaomiao', 'cat')
```

函数参数

■ 默认值

```
1 def describe_pet( pet_name, animal_type="dog"):  
2     """显示宠物的信息"""  
3     print("\nI have a " + animal_type +  
4         ". " + " Its name is " + pet_name)  
5 describe_pet( 'wangwang')  
6 describe_pet( 'miaomiao', 'cat')
```

■ 可以混合使用，注意顺序，默认参数放在后面.

函数参数

- 事先不知道应该放入多少参数？

函数参数

- 事先不知道应该放入多少参数？
- 使用 `*args` 可以传入不定数目的参数，此时函数会建立一个空的元组 `args`，然后将所有实参放进该元组

```
1 def choice_courses(*courses):  
2     """选择课程"""  
3     print("\nYou choice courses as :")  
4     for course in courses:  
5         print("- " + course)  
6 choice_courses('mathematics')  
7 choice_courses('mathematics', 'physics', 'chemistry')
```

函数参数

■ 任意数量的关键字参数？

函数参数

- 任意数量的关键字参数？
- 使用 `**args` 传入指定关键字的参数，关键字与实参组成键值对存入字典 args 中

```
1 def show_scores(**courses):  
2     """选择课程"""  
3     print("\nYou choice courses as :")  
4     for course,score in courses:  
5         print("\n course "+course + "'s score is " + str(  
              score))  
6 show_scores(mathematics=90)  
7 show_scores(mathematics=90, physics=95, chemistry=92)
```

函数编写建议

- 总体来说函数参数的排列为：

```
def func(arg1, arg2, ..., keywords1=arg11,  
..., *tuble, **dicts)
```

函数编写建议

- 总体来说函数参数的排列为：

```
def func(arg1, arg2, ..., keywords1=arg11,  
..., *tuble, **dicts)
```

- 编写程序时，可以以各种方式混合使用各种实参。

函数编写建议

- 总体来说函数参数的排列为：

```
def func(arg1, arg2, ..., keywords1=arg11,  
..., *tuble, **dicts)
```

- 编写程序时，可以以各种方式混合使用各种实参。
- 编写指南（只是建议）

函数编写建议

- 总体来说函数参数的排列为：

```
def func(arg1, arg2, ..., keywords1=arg11,  
..., *tuble, **dicts)
```

- 编写程序时，可以以各种方式混合使用各种实参。
- 编写指南（只是建议）

- 函数都以小写字母和下划线命名，名字应易懂。

函数编写建议

- 总体来说函数参数的排列为：

```
def func(arg1, arg2, ..., keywords1=arg11,  
..., *tuble, **dicts)
```

- 编写程序时，可以以各种方式混合使用各种实参。
- 编写指南（只是建议）

- 函数都以小写字母和下划线命名，名字应易懂。
- 函数后面跟着注释，说明函数的功能，更详细的可以说明参数的作用。

函数编写建议

- 总体来说函数参数的排列为：

```
def func(arg1, arg2, ..., keywords1=arg11,  
..., *tuble, **dicts)
```

- 编写程序时，可以以各种方式混合使用各种实参。
- 编写指南（只是建议）

- 函数都以小写字母和下划线命名，名字应易懂。
- 函数后面跟着注释，说明函数的功能，更详细的可以说明参数的作用。
- 函数和函数之间空两行

将函数存储在模块中

- 函数的优点是将一般性的方法分离，方便重复调用。通常将一定功能的函数放入同一个文件中，该文件被称为**模块**

将函数存储在模块中

- 函数的优点是将一般性的方法分离，方便重复调用。通常将一定功能的函数放入同一个文件中，该文件被称为**模块**
- 在其它程序调用模块中的函数时，使用 `import` 语句

将函数存储在模块中

- 函数的优点是将一般性的方法分离，方便重复调用。通常将一定功能的函数放入同一个文件中，该文件被称为**模块**
- 在其它程序调用模块中的函数时，使用 import 语句
 - import 模块名：导入整个模块

将函数存储在模块中

- 函数的优点是将一般性的方法分离，方便重复调用。通常将一定功能的函数放入同一个文件中，该文件被称为**模块**
- 在其它程序调用模块中的函数时，使用 import 语句
 - import 模块名：导入整个模块
 - 调用该模块中的函数：模块名.函数名

将函数存储在模块中

- 函数的优点是将一般性的方法分离，方便重复调用。通常将一定功能的函数放入同一个文件中，该文件被称为**模块**
- 在其它程序调用模块中的函数时，使用 import 语句
 - import 模块名：导入整个模块
 - 调用该模块中的函数：模块名.函数名
 - 只导入特定函数：from module_name import func1, func2。此时调用只用函数名即可

将函数存储在模块中

- 函数的优点是将一般性的方法分离，方便重复调用。通常将一定功能的函数放入同一个文件中，该文件被称为**模块**
- 在其它程序调用模块中的函数时，使用 import 语句
 - import 模块名：导入整个模块
 - 调用该模块中的函数：模块名.函数名
 - 只导入特定函数：from module_name import func1, func2。此时调用只用函数名即可
 - 导入模块中所有的函数：from module_name import *. 所有函数都被导入，可直接使用函数名调用

将函数存储在模块中

■ 使用 as 给模块指定别名

```
1 import numpy as np
```

将函数存储在模块中

■ 使用 as 给模块指定别名

```
1 import numpy as np
```

■ 使用 as 给函数指定别名

```
1 from scipy.optimize import curve_fit as fit
```

第三节

Python 语言基础

3.1 基本语法

3.2 列表简介

3.3 函数

3.4 numpy 包的使用

numpy

简介

- 科学计算离不开大量的循环计算操作，而 python
自己的循环效率很低.

numpy

简介

- 科学计算离不开大量的循环计算操作，而 python
自己的循环效率很低.
- numpy 为 python 提供了快速的多维数组处理
的能力

numpy

简介

- 科学计算离不开大量的循环计算操作，而 python 自己的循环效率很低.
- numpy 为 python 提供了快速的多维数组处理的能力
- 它和 scipy 一起提供了众多科学计算所需要的工具包，sympy 则提供了在 python 环境下进行符号计算的软件包. 详细介绍 <http://www.scipy.org>

numpy

简介

- 科学计算离不开大量的循环计算操作，而 python 自己的循环效率很低.
- numpy 为 python 提供了快速的多维数组处理的能力
- 它和 scipy 一起提供了众多科学计算所需要的工具包，sympy 则提供了在 python 环境下进行符号计算的软件包. 详细介绍 <http://www.scipy.org>
- 常用的一些数值计算库

numpy

简介

- 科学计算离不开大量的循环计算操作，而 python 自己的循环效率很低.
- numpy 为 python 提供了快速的多维数组处理的能力
- 它和 scipy 一起提供了众多科学计算所需要的工具包，sympy 则提供了在 python 环境下进行符号计算的软件包. 详细介绍 <http://www.scipy.org>
- 常用的一些数值计算库
 - LAPACK, FFTPACK, ODEPACK, MINPACK

创建多维数组

- 首先 import numpy

创建多维数组

- 首先 import numpy
- 给 array 函数传入 python 序列对象创建数组，
如果是多层嵌套序列，则创建多维数组

```
1 import numpy as np  
2 a = np.array([1,2,3])  
3 b = np.array([[1,2,3],[4,5,6],[7,8,9],[10,11,12]])  
4 a.shape  
5 b.shape
```

创建多维数组

- `shape` 属性可以更改, 只是改变每个轴的大小 (-1, 表示自动计算该维度大小)

```
1 b.shape = 4, -1
```

创建多维数组

- `shape` 属性可以更改, 只是改变每个轴的大小 (-1, 表示自动计算该维度大小)

```
1 b.shape = 4, -1
```

- 数组的 `reshape` 方法可以创建一个改变尺寸的新数组, 原数组的 `shape` 保持不变

```
1 c = b.reshape((2,6))
```

创建多维数组

- `shape` 属性可以更改, 只是改变每个轴的大小 (-1, 表示自动计算该维度大小)

```
1 b.shape = 4, -1
```

- 数组的 `reshape` 方法可以创建一个改变尺寸的新数组, 原数组的 `shape` 保持不变

```
1 c = b.reshape((2,6))
```

- 注意 `c` 和 `b` 数组按这种方式其内存共享, 改变 `b` 中的元素同样会影响 `c`

创建多维数组

■ 一些内置函数来快速创建数组

创建多维数组

- 一些内置函数来快速创建数组
 - `numpy.arange([start,]stop, [step,]dtype=None)`
不包含终值

创建多维数组

■ 一些内置函数来快速创建数组

- `numpy.arange([start,]stop, [step,]dtype=None)`
不包含终值
- `numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)`, 缺省包含终值

创建多维数组

■ 一些内置函数来快速创建数组

- `numpy.arange([start,]stop, [step,]dtype=None)`
不包含终值
- `numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)`, 缺省包含终值
- `numpy.diag(v, k=0),`
`numpy.ones(shape, dtype=None, order='C')`

创建多维数组

■ 一些内置函数来快速创建数组

- `numpy.arange([start,]stop, [step,]dtype=None)`
不包含终值
- `numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)`, 缺省包含终值
- `numpy.diag(v, k=0),`
`numpy.ones(shape, dtype=None, order='C')`
- `numpy.empty(shape, dtype=float, order='C')`,
`numpy.zeros(shape, dtype=float, order='C')`

创建多维数组

■ 一些内置函数来快速创建数组

- `numpy.arange([start,]stop, [step,]dtype=None)`
不包含终值
- `numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)`, 缺省包含终值
- `numpy.diag(v, k=0),`
`numpy.ones(shape, dtype=None, order='C')`
- `numpy.empty(shape, dtype=float, order='C')`,
`numpy.zeros(shape, dtype=float, order='C')`
- `numpy.fromfunction(function, shape, **kwargs)`

存取元素

- 数组元素的存取方法和 Python 的标准方法相同

存取元素

- 数组元素的存取方法和 Python 的标准方法相同
- 和 Python 的列表序列不同，通过下标范围获取的新的数组是原始数组的一个视图。它与原始数组共享同一块数据空间

存取元素

- 数组元素的存取方法和 Python 的标准方法相同
- 和 Python 的列表序列不同，通过下标范围获取的新的数组是原始数组的一个视图。它与原始数组共享同一块数据空间
- 两种存取元素的高级方法

存取元素

- 数组元素的存取方法和 Python 的标准方法相同
- 和 Python 的列表序列不同，通过下标范围获取的新的数组是原始数组的一个视图。它与原始数组共享同一块数据空间
- 两种存取元素的高级方法
 - 使用整数序列

存取元素

- 数组元素的存取方法和 Python 的标准方法相同
- 和 Python 的列表序列不同，通过下标范围获取的新的数组是原始数组的一个视图。它与原始数组共享同一块数据空间
- 两种存取元素的高级方法
 - 使用整数序列
 - 使用布尔数组

ufunc 运算

- ufunc 是一种能对数组每个元素进行操作的函数。内置函数大多都是在 C 语言级别实现的，运算速度非常快

ufunc 运算

- ufunc 是一种能对数组每个元素进行操作的函数。内置函数大多都是在 C 语言级别实现的，运算速度非常快
- 各类常见的数学函数：sum, sin, exp, average,

ufunc 运算

- ufunc 是一种能对数组每个元素进行操作的函数。内置函数大多都是在 C 语言级别实现的，运算速度非常快
- 各类常见的数学函数：sum, sin, exp, average,
- 复杂的操作可能会产生大量的中间结果而降低程序的运行效率

```
1 x = a*b+c  
2 x = a*b  
3 x+ = c
```

ufunc 运算规则

- 1. 让所有输入数组都向其中 shape 最长的数组看齐, shape 中不足的部分都通过在前面加 1 补齐

ufunc 运算规则

- 1. 让所有输入数组都向其中 shape 最长的数组看齐, shape 中不足的部分都通过在前面加 1 补齐
- 2. 输出数组的 shape 是输入数组 shape 的各个轴上的最大值

ufunc 运算规则

- 1. 让所有输入数组都向其中 shape 最长的数组看齐, shape 中不足的部分都通过在前面加 1 补齐
- 2. 输出数组的 shape 是输入数组 shape 的各个轴上的最大值
- 3. 如果输入数组的某个轴和输出数组的对应轴的长度相同或者其长度为 1 时, 这个数组能够用来计算, 否则出错

ufunc 运算规则

- 1. 让所有输入数组都向其中 shape 最长的数组看齐, shape 中不足的部分都通过在前面加 1 补齐
- 2. 输出数组的 shape 是输入数组 shape 的各个轴上的最大值
- 3. 如果输入数组的某个轴和输出数组的对应轴的长度相同或者其长度为 1 时, 这个数组能够用来计算, 否则出错
- 4. 当输入数组的某个轴的长度为 1 时, 沿着此轴运算时都用此轴上的第一组值

```
1 a = np.arange(0, 60, 10).reshape(-1, 1)
2 b = np.arange(0, 5)
3 c = a + b
```

自定义 ufunc

- `numpy.frompyfunc(func, nin, nout)` 将任意的 python 函数转变成 numpy 的 ufunc

自定义 ufunc

- `numpy.frompyfunc(func, nin, nout)` 将任意的 python 函数转变成 numpy 的 ufunc
- `class numpy.vectorize(pyfunc, otypes=None, doc=None, excluded=None, cache=False, signature=None)[source]`

```
1 def myfunc(a, b):  
2     "Return a-b if a>b, otherwise return a+b"  
3     if a > b:  
4         return a - b  
5     else:  
6         return a + b  
7 vfunc = np.vectorize(myfunc)  
8 vfunc([1,2,3,4],2)
```

调用外部函数

■ 调用已有的 fortran 函数，推荐用 f2py

1 `f2py -c -m fib3 fib3.f`

调用外部函数

■ 调用已有的 fortran 函数，推荐用 f2py

```
1 f2py -c -m fib3 fib3.f
```

■ 调用已有的 c 函数，ctypes,
numpy.ctypeslib.load_library, cython 等

```
1 gcc -shared -o filename.so -fPIC filename.c
```

```
1 import filename  
2 filename.func()
```

调用外部函数

■ 调用已有的 fortran 函数，推荐用 f2py

```
1 f2py -c -m fib3 fib3.f
```

■ 调用已有的 c 函数，ctypes, numpy.ctypeslib.load_library, cython 等

```
1 gcc -shared -o filename.so -fPIC filename.c
```

```
1 import filename  
2 filename.func()
```

■ C++ : Boost.Python

第三节

Python 语言基础

3.1 基本语法

3.2 列表简介

3.3 函数

3.4 numpy 包的使用

练习

- 1 编写一个猜数字的小程序，要求每次输入一个数，程序反馈这个数太小，或是太大，然后再次输入，直到猜出正确的数字为止。

练习

- 1 编写一个猜数字的小程序，要求每次输入一个数，程序反馈这个数太小，或是太大，然后再次输入，直到猜出正确的数字为止。
- 2 假定有下面这样的列表：

```
1 author = [ 'Georg P. Engel', 'C. B. Lang', 'Daniel Mohler',  
             'Andreas Schafer' ]
```

编写一个函数，它以一个列表值作为参数，返回一个字符串。例如，将前面的 author 列表传递给函数，将返回

```
1 'G.~P.~Engel, C.~B.~Lang, D.~Mohler and A.~Schafer'
```

练习

1 归一化，将矩阵规格化到 0~1，即最小的变成 0，最大的变成 1，最小与最大之间的等比缩放。

1 [[3, 2, 9], [4, 7, 3], [11, 8, 5]]

练习

1 归一化，将矩阵规格化到 0~1，即最小的变成 0，最大的变成 1，最小与最大之间的等比缩放。

1 [[3, 2, 9], [4, 7, 3], [11, 8, 5]]

2 用 for 循环写两个矩阵的乘法，并和 np.dot() 的结果对比

1 [[3, 4, 5], [2, 7, 9], [1, 6, 8]]

2 [[2, 9, 1], [1, 5, 2], [3, 7, 4]]

练习

1 归一化，将矩阵规格化到 0~1，即最小的变成 0，最大的变成 1，最小与最大之间的等比缩放。

1 [[3, 2, 9], [4, 7, 3], [11, 8, 5]]

2 用 for 循环写两个矩阵的乘法，并和 np.dot() 的结果对比

1 [[3, 4, 5], [2, 7, 9], [1, 6, 8]]

2 [[2, 9, 1], [1, 5, 2], [3, 7, 4]]

3 对一组数据计算其平均值，并给出测量结果的标准差和标准误。

1 [1.233, 1.242, 1.215, 1.237, 1.282, 1.243, 1.223, 1.219]

第二节 linux 系统编程简介

第三节 Python 语言基础

第四节 Mathematica 基础

第五节 函数近似

第六节 数值微积分

第四节

Mathematica 基础

4.1 基本语法

4.2 基本概念

4.3 解方程

4.4 微积分

4.5 微分方程

4.6 线性代数

4.7 练习

基本语法

- 加减乘除 (+, -, *, /). 两个符号间空格代表着乘法.

基本语法

- 加减乘除 (+, -, *, /). 两个符号间空格代表着乘法.
- 字母区分大小写, 所有内置函数都是以大写字母开头. 因此用户自定义变量及函数都用小写字母开头.

基本语法

- 加减乘除 (+, -, *, /). 两个符号间空格代表着乘法.
- 字母区分大小写, 所有内置函数都是以大写字母开头. 因此用户自定义变量及函数都用小写字母开头.
- 不同括号的用途

基本语法

- 加减乘除 (+, -, *, /). 两个符号间空格代表着乘法.
- 字母区分大小写, 所有内置函数都是以大写字母开头. 因此用户自定义变量及函数都用小写字母开头.
- 不同括号的用途
 - 方括号用在函数参数指定中: $\text{Sin}[x]$ (不是 $\text{Sin}(x)$)

基本语法

- 加减乘除 (+, -, *, /). 两个符号间空格代表着乘法.
- 字母区分大小写, 所有内置函数都是以大写字母开头. 因此用户自定义变量及函数都用小写字母开头.
- 不同括号的用途
 - 方括号用在函数参数指定中: $\text{Sin}[x]$ (不是 $\text{Sin}(x)$)
 - 圆括号表示分组.

基本语法

- 加减乘除 (+, -, *, /). 两个符号间空格代表着乘法.
- 字母区分大小写, 所有内置函数都是以大写字母开头. 因此用户自定义变量及函数都用小写字母开头.
- 不同括号的用途
 - 方括号用在函数参数指定中: $\text{Sin}[x]$ (不是 $\text{Sin}(x)$)
 - 圆括号表示分组.
 - 打括号表示列表: {1,2,3,4}

基本语法

- 加减乘除 (+, -, *, /). 两个符号间空格代表着乘法.
- 字母区分大小写, 所有内置函数都是以大写字母开头. 因此用户自定义变量及函数都用小写字母开头.
- 不同括号的用途
 - 方括号用在函数参数指定中: $\text{Sin}[x]$ (不是 $\text{Sin}(x)$)
 - 圆括号表示分组.
 - 打括号表示列表: {1,2,3,4}
- 自然数和虚数的符号是 E,I

基本语法

- 加减乘除 (+, -, *, /). 两个符号间空格代表着乘法.
- 字母区分大小写, 所有内置函数都是以大写字母开头. 因此用户自定义变量及函数都用小写字母开头.
- 不同括号的用途
 - 方括号用在函数参数指定中: $\text{Sin}[x]$ (不是 $\text{Sin}(x)$)
 - 圆括号表示分组.
 - 打括号表示列表: {1,2,3,4}
- 自然数和虚数的符号是 E,I
- 如不做指定, mathematica 总是给出准确的数学表达式.

基本语法

- 变量名可以由字母数字及一些特定字符构成，但首字母不能是数字

基本语法

- 变量名可以由字母数字及一些特定字符构成，但首字母不能是数字
- 运算指令后加；号，可以不显示出运算结果

基本语法

- 变量名可以由字母数字及一些特定字符构成，但首字母不能是数字
- 运算指令后加；号，可以不显示出运算结果
- 优先级：括号 > 指数 > 乘除 > 加减法

基本语法

- 变量名可以由字母数字及一些特定字符构成，但首字母不能是数字
- 运算指令后加；号，可以不显示出运算结果
- 优先级：括号 > 指数 > 乘除 > 加减法
- 用 = 进行赋值，用 Clear[符号名] 或 Remove[符号名] 来删除符号

基本语法

- 变量名可以由字母数字及一些特定字符构成，但首字母不能是数字
- 运算指令后加；号，可以不显示出运算结果
- 优先级：括号 > 指数 > 乘除 > 加减法
- 用 = 进行赋值, 用 Clear[符号名] 或 Remove[符号名] 来删除符号
- 表达式//命令等价于命令 [表达式]

基本语法

- 变量名可以由字母数字及一些特定字符构成，但首字母不能是数字
- 运算指令后加；号，可以不显示出运算结果
- 优先级：括号 > 指数 > 乘除 > 加减法
- 用 = 进行赋值, 用 Clear[符号名] 或 Remove[符号名] 来删除符号
- 表达式//命令等价于命令 [表达式]
- %...(k) 返回前第 k 次计算结果, %n 返回第 n 次的输出结果

基本语法

- 变量名可以由字母数字及一些特定字符构成，但首字母不能是数字
- 运算指令后加；号，可以不显示出运算结果
- 优先级：括号 > 指数 > 乘除 > 加减法
- 用 = 进行赋值, 用 Clear[符号名] 或 Remove[符号名] 来删除符号
- 表达式//命令等价于命令 [表达式]
- %...(k) 返回前第 k 次计算结果, %n 返回第 n 次的输出结果

- 计算 $1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{2}}}}$

第四节

Mathematica 基础

4.1 基本语法

4.2 基本概念

4.3 解方程

4.4 微积分

4.5 微分方程

4.6 线性代数

4.7 练习

内置函数

■ 一些常见的内置函数

内置函数

■ 一些常见的内置函数

- `Sqrt[]`,`Abs[]`,`Sign[]`,`Factorial[]`

内置函数

■ 一些常见的内置函数

- `Sqrt[]`,`Abs[]`,`Sign[]`,`Factorial[]`
- `RandomReal[]`,`Mod[]`,`Quotient[]`,`Log[]`

内置函数

■ 一些常见的内置函数

- `Sqrt[]`,`Abs[]`,`Sign[]`,`Factorial[]`
- `RandomReal[]`,`Mod[]`,`Quotient[]`,`Log[]`
- `Sin[]`,`Cos[]`,`Tan[]`,`Cot[]`,`Sec[]`,`Csc[]`

内置函数

■ 一些常见的内置函数

- `Sqrt[]`,`Abs[]`,`Sign[]`,`Factorial[]`
- `RandomReal[]`,`Mod[]`,`Quotient[]`,`Log[]`
- `Sin[]`,`Cos[]`,`Tan[]`,`Cot[]`,`Sec[]`,`Csc[]`
- `Print[]`

内置函数

■ 一些常见的内置函数

- `Sqrt[]`,`Abs[]`,`Sign[]`,`Factorial[]`
- `RandomReal[]`,`Mod[]`,`Quotient[]`,`Log[]`
- `Sin[]`,`Cos[]`,`Tan[]`,`Cot[]`,`Sec[]`,`Csc[]`
- `Print[]`
- `StringJoin[]`,`StringReplace[]`

赋值、替换、逻辑判断

■ 赋值

赋值、替换、逻辑判断

■ 赋值

- $=$, 即时赋值

赋值、替换、逻辑判断

■ 赋值

- $=$, 即时赋值
- $:=$, 延时赋值

赋值、替换、逻辑判断

■ 赋值

- `=`, 即时赋值
- `:=`, 延时赋值

■ 替换

赋值、替换、逻辑判断

■ 赋值

- $=$, 即时赋值
- $:=$, 延时赋值

■ 替换

- 表达式/.{ $a \rightarrow b$ }

赋值、替换、逻辑判断

■ 赋值

- $=$, 即时赋值
- $:=$, 延时赋值

■ 替换

- 表达式/.{ $a \rightarrow b$ }
- a 和 b 可以是单个变量也可以是表达式

赋值、替换、逻辑判断

■ 赋值

- $=$, 即时赋值
- $:=$, 延时赋值

■ 替换

- 表达式/.{ $a \rightarrow b$ }
- a 和 b 可以是单个变量也可以是表达式

■ 逻辑判断

赋值、替换、逻辑判断

■ 赋值

- $=$, 即时赋值
- $:=$, 延时赋值

■ 替换

- 表达式/.{ $a \rightarrow b$ }
- a 和 b 可以是单个变量也可以是表达式

■ 逻辑判断

- Equal[x,y] 等价于 $x == y$

赋值、替换、逻辑判断

■ 赋值

- $=$, 即时赋值
- $:=$, 延时赋值

■ 替换

- 表达式/.{ $a \rightarrow b$ }
- a 和 b 可以是单个变量也可以是表达式

■ 逻辑判断

- Equal[x,y] 等价于 $x == y$
- Unequal[x,y] 等价于 $x != y$

赋值、替换、逻辑判断

■ 赋值

- $=$, 即时赋值
- $:=$, 延时赋值

■ 替换

- 表达式/.{ $a \rightarrow b$ }
- a 和 b 可以是单个变量也可以是表达式

■ 逻辑判断

- Equal[x,y] 等价于 $x == y$
- Unequal[x,y] 等价于 $x != y$
- Less, Greater, LessEqual, GreaterEqual

赋值、替换、逻辑判断

■ 赋值

- $=$, 即时赋值
- $:=$, 延时赋值

■ 替换

- 表达式/.{ $a \rightarrow b$ }
- a 和 b 可以是单个变量也可以是表达式

■ 逻辑判断

- Equal[x,y] 等价于 $x == y$
- Unequal[x,y] 等价于 $x != y$
- Less, Greater, LessEqual, GreaterEqual
- And, Or, Xor, Not

和与积、循环、画图、自定义函数

■ 和与积

和与积、循环、画图、自定义函数

■ 和与积

■ Sum, Product

和与积、循环、画图、自定义函数

■ 和与积

- Sum, Product

■ 循环

和与积、循环、画图、自定义函数

■ 和与积

- Sum, Product

■ 循环

- Do[表达式, {i,imin,imax}], While[条件, 表达式],
For[初始化, 条件, 增量, 表达式]

和与积、循环、画图、自定义函数

■ 和与积

- Sum, Product

■ 循环

- Do[表达式, {i,imin,imax}], While[条件, 表达式],
For[初始化, 条件, 增量, 表达式]
- If[条件, 真, 假, 空]

和与积、循环、画图、自定义函数

■ 和与积

- Sum, Product

■ 循环

- Do[表达式, {i,imin,imax}], While[条件, 表达式],
For[初始化, 条件, 增量, 表达式]
- If[条件, 真, 假, 空]

■ 画图

和与积、循环、画图、自定义函数

■ 和与积

- Sum, Product

■ 循环

- Do[表达式, {i,imin,imax}], While[条件, 表达式],
For[初始化, 条件, 增量, 表达式]
- If[条件, 真, 假, 空]

■ 画图

- Plot[] 参数很多, 多看帮助文档

和与积、循环、画图、自定义函数

■ 和与积

- Sum, Product

■ 循环

- Do[表达式, {i,imin,imax}], While[条件, 表达式], For[初始化, 条件, 增量, 表达式]
- If[条件, 真, 假, 空]

■ 画图

- Plot[] 参数很多, 多看帮助文档

■ 自定义函数 (定义方式, 分段函数, 复合函数, 递归函数)

和与积、循环、画图、自定义函数

■ 和与积

- Sum, Product

■ 循环

- Do[表达式, {i,imin,imax}], While[条件, 表达式], For[初始化, 条件, 增量, 表达式]
- If[条件, 真, 假, 空]

■ 画图

- Plot[] 参数很多, 多看帮助文档

■ 自定义函数 (定义方式, 分段函数, 复合函数, 递归函数)

- $f[x_]=\dots$

和与积、循环、画图、自定义函数

■ 和与积

- Sum, Product

■ 循环

- Do[表达式, {i,imin,imax}], While[条件, 表达式], For[初始化, 条件, 增量, 表达式]
- If[条件, 真, 假, 空]

■ 画图

- Plot[] 参数很多, 多看帮助文档

■ 自定义函数 (定义方式, 分段函数, 复合函数, 递归函数)

- $f[x_]=\dots$
- $f[x_]:=\dots$

列表

- **List[元素]**, 元素由逗号分开的各个列表对象, 等价于 {元素}.

列表

- **List[元素]**, 元素由逗号分开的各个列表对象, 等价于 {元素}.
- 生成列表

列表

- **List[元素]**, 元素由逗号分开的各个列表对象, 等价于 {元素}.
- 生成列表
 - Range[m,n,d]

列表

- **List[元素]**, 元素由逗号分开的各个列表对象, 等价于 {元素}.
- 生成列表
 - Range[m,n,d]
 - Table[表达式, {k,m,n,d}]

列表

- **List[元素]**, 元素由逗号分开的各个列表对象, 等价于 {元素}.
- 生成列表
 - Range[m,n,d]
 - Table[表达式, {k,m,n,d}]
 - Array[f,n,r]

列表

- **List[元素]**, 元素由逗号分开的各个列表对象, 等价于 {元素}.
- 生成列表
 - Range[m,n,d]
 - Table[表达式, {k,m,n,d}]
 - Array[f,n,r]
- 列表操作

列表

- **List[元素]**, 元素由逗号分开的各个列表对象, 等价于 {元素}.
- 生成列表
 - Range[m,n,d]
 - Table[表达式, {k,m,n,d}]
 - Array[f,n,r]
- 列表操作
 - Length[]

列表

- **List[元素]**, 元素由逗号分开的各个列表对象, 等价于 {元素}.
- 生成列表
 - Range[m,n,d]
 - Table[表达式, {k,m,n,d}]
 - Array[f,n,r]
- 列表操作
 - Length[]
 - First[], Last[], Part[lst,k], Part[lst,-k]

列表

- **List[元素]**, 元素由逗号分开的各个列表对象, 等价于 {元素}.
- 生成列表
 - Range[m,n,d]
 - Tabel[表达式, {k,m,n,d}]
 - Array[f,n,r]
- 列表操作
 - Length[]
 - First[],Last[],Part[lst,k],Part[lst,-k]
 - Sort[],Reverse[]

第四节

Mathematica 基础

4.1 基本语法

4.2 基本概念

4.3 解方程

4.4 微积分

4.5 微分方程

4.6 线性代数

4.7 练习

求解代数方程

- 求解一些简单方程的主要函数 `Solve[方程, 变量]`

求解代数方程

- 求解一些简单方程的主要函数 `Solve[方程, 变量]`
- 得到的解其实是一个规则即 `Rule[]`

求解代数方程

- 求解一些简单方程的主要函数 `Solve[方程, 变量]`
- 得到的解其实是一个规则即 `Rule[]`
- `NSolve[]` 用数值的方法求解方程

求解代数方程

- 求解一些简单方程的主要函数 `Solve[方程, 变量]`
- 得到的解其实是一个规则即 `Rule[]`
- `NSolve[]` 用数值的方法求解方程
- `Reduce[]` 函数也可以用来求解方程. 给出表达式等价的约化简化表示

求解代数方程

- 求解一些简单方程的主要函数 `Solve[方程, 变量]`
- 得到的解其实是一个规则即 `Rule[]`
- `NSolve[]` 用数值的方法求解方程
- `Reduce[]` 函数也可以用来求解方程. 给出表达式等价的约化简化表示
- 对于一些超越方程, 可以用 `FindRoot[]`

代数与三角

■ 处理多项式的一些主要函数有

代数与三角

■ 处理多项式的一些主要函数有

- `Coefficient[多项式, 项]`, `CoefficientList[多项式, 变量]`

代数与三角

■ 处理多项式的一些主要函数有

- `Coefficient[多项式, 项]`, `CoefficientList[多项式, 变量]`
- `Expand[], Factor, Collect[]`

代数与三角

- 处理多项式的一些主要函数有
 - Coefficient[多项式, 项], CoefficientList[多项式, 变量]
 - Expand[], Factor, Collect[]
- 有理函数和代数函数

代数与三角

- 处理多项式的一些主要函数有
 - Coefficient[多项式, 项], CoefficientList[多项式, 变量]
 - Expand[], Factor, Collect[]
- 有理函数和代数函数
 - Apart[], 把分式表示为部分分式的和

代数与三角

- 处理多项式的一些主要函数有
 - Coefficient[多项式, 项], CoefficientList[多项式, 变量]
 - Expand[], Factor, Collect[]
- 有理函数和代数函数
 - Apart[], 把分式表示为部分分式的和
 - Together[], 利用公分母把表达式中的项合并起来，并消去公因子.

代数与三角

- 处理多项式的一些主要函数有
 - Coefficient[多项式, 项], CoefficientList[多项式, 变量]
 - Expand[], Factor, Collect[]
- 有理函数和代数函数
 - Apart[], 把分式表示为部分分式的和
 - Together[], 利用公分母把表达式中的项合并起来，并消去公因子.
- 化简. 不同的目的有不同的简化形式，灵活组合各个函数.

代数与三角

- 处理多项式的一些主要函数有
 - Coefficient[多项式, 项], CoefficientList[多项式, 变量]
 - Expand[], Factor, Collect[]
- 有理函数和代数函数
 - Apart[], 把分式表示为部分分式的和
 - Together[], 利用公分母把表达式中的项合并起来，并消去公因子.
- 化简. 不同的目的有不同的简化形式，灵活组合各个函数.
 - Simplify[], FullSimplify[]. 默认展开, 因式分解及三角简化等.

第四节

Mathematica 基础

4.1 基本语法

4.2 基本概念

4.3 解方程

4.4 微积分

4.5 微分方程

4.6 线性代数

4.7 练习

极限的计算

- $\text{Limit}[f[x], x \rightarrow a]$ 计算 $\lim_{x \rightarrow a} f(x)$ 的值

极限的计算

- $\text{Limit}[f[x], x \rightarrow a]$ 计算 $\lim_{x \rightarrow a} f(x)$ 的值
- 对于不连续的函数, Direction 参数指定逼近方向

极限的计算

- $\text{Limit}[f[x], x \rightarrow a]$ 计算 $\lim_{x \rightarrow a} f(x)$ 的值
- 对于不连续的函数, Direction 参数指定逼近方向
- 可以计算趋向于无穷的极限

极限的计算

- $\text{Limit}[f[x], x \rightarrow a]$ 计算 $\lim_{x \rightarrow a} f(x)$ 的值
- 对于不连续的函数, Direction 参数指定逼近方向
- 可以计算趋向于无穷的极限
- 极限可以是无穷

导数的计算

- $D[]$ 或者 $f'[x]$ 表示对函数 $f[x]$ 求导

导数的计算

- $D[\cdot]$ 或者 $f'[x]$ 表示对函数 $f[x]$ 求导
- $D[f[x], x]$ 返回 f 对 x 的一阶导数

导数的计算

- $D[]$ 或者 $f'[x]$ 表示对函数 $f[x]$ 求导
- $D[f[x],x]$ 返回 f 对 x 的一阶导数
- $D[f[x],\{x,n\}]$ 返回 f 对 x 的 n 阶导数

导数的计算

- $D[\cdot]$ 或者 $f'[x]$ 表示对函数 $f[x]$ 求导
- $D[f[x], x]$ 返回 f 对 x 的一阶导数
- $D[f[x], \{x, n\}]$ 返回 f 对 x 的 n 阶导数
- $D[f[x, y], \{\{x, y\}\}]$ 向量导数

导数的计算

- $D[]$ 或者 $f'[x]$ 表示对函数 $f[x]$ 求导
- $D[f[x],x]$ 返回 f 对 x 的一阶导数
- $D[f[x],\{x,n\}]$ 返回 f 对 x 的 n 阶导数
- $D[f[x,y],\{\{x,y\}\}]$ 向量导数
- $Dt[f]$ 给出全微分

导数的计算

- $D[]$ 或者 $f'[x]$ 表示对函数 $f[x]$ 求导
- $D[f[x],x]$ 返回 f 对 x 的一阶导数
- $D[f[x],\{x,n\}]$ 返回 f 对 x 的 n 阶导数
- $D[f[x,y],\{\{x,y\}\}]$ 向量导数
- $Dt[f]$ 给出全微分
- $Series[]$ 进行泰勒展开, $Normal[]$ 给出可计算的多项式表示.

导数的计算

例 1 证明函数 $f(x) = (x^3 + 2x^2 + 15x + 2) \sin \pi x$ 在区间 $[0, 1]$ 上满足罗尔定理，并求出定理中所声称的点 C.

积分的计算

- `Integrate[f[x],x]` 计算不定积分 $\int f(x)dx$.

积分的计算

- `Integrate[f[x],x]` 计算不定积分 $\int f(x)dx$.
- `Integrate[f[x],{x,a,b}]` 计算定积分 $\int_a^b f(x)dx$

积分的计算

- `Integrate[f[x],x]` 计算不定积分 $\int f(x)dx$.
- `Integrate[f[x],{x,a,b}]` 计算定积分 $\int_a^b f(x)dx$
- `Integrate[f[x,y],{x,a,b},{y,c,d}]` 计算多重积分，最外层为 x

积分的计算

- `Integrate[f[x],x]` 计算不定积分 $\int f(x)dx$.
- `Integrate[f[x],{x,a,b}]` 计算定积分 $\int_a^b f(x)dx$
- `Integrate[f[x,y],{x,a,b},{y,c,d}]` 计算多重积分，最外层为 x
- `NIntegrate[]` 给出数值积分

第四节

Mathematica 基础

4.1 基本语法

4.2 基本概念

4.3 解方程

4.4 微积分

4.5 微分方程

4.6 线性代数

4.7 练习

微分方程

- DSolve[方程, $y[x]$, x] 给出独立变量为 x 的微分方程的一般解 (注意得到的解只是 $y[x]$ 这个符合的形式)

微分方程

- DSolve[方程, $y[x]$, x] 给出独立变量为 x 的微分方程的一般解 (注意得到的解只是 $y[x]$ 这个符合的形式)
- DSolve[方程, y , x] 给出纯函数解, 一般来说会更方面后期使用.

微分方程

- DSolve[方程, $y[x], x]$ 给出独立变量为 x 的微分方程的一般解 (注意得到的解只是 $y[x]$ 这个符合的形式)
- DSolve[方程, $y, x]$ 给出纯函数解, 一般来说会更方面后期使用.
- DSolve[方程 && 边界条件, $y[x], x]$ 给出在边界条件下的解.

微分方程

- DSolve[方程, $y[x]$, x] 给出独立变量为 x 的微分方程的一般解 (注意得到的解只是 $y[x]$ 这个符合的形式)
- DSolve[方程, y , x] 给出纯函数解, 一般来说会更方面后期使用.
- DSolve[方程 $\&\&$ 边界条件, $y[x]$, x] 给出在边界条件下的解.
- NDSolve[方程, y , $\{x, xmin, xmax\}$] 给出方程的数值解, 其中 x 满足给定条件.

微分方程

- DSolve[方程, $y[x], x]$ 给出独立变量为 x 的微分方程的一般解 (注意得到的解只是 $y[x]$ 这个符合的形式)
- DSolve[方程, $y, x]$ 给出纯函数解, 一般来说会更方面后期使用.
- DSolve[方程 && 边界条件, $y[x], x]$ 给出在边界条件下的解.
- NDSolve[方程, $y, \{x, xmin, xmax\}$] 给出方程的数值解, 其中 x 满足给定条件.
- 对于一些方程, 可以先做拉普拉斯变换或傅立叶变换再求解. LaplaceTransform[], FourierTransform[]

微分方程

- DSolve[方程, $y[x]$, x] 给出独立变量为 x 的微分方程的一般解 (注意得到的解只是 $y[x]$ 这个符合的形式)
- DSolve[方程, y , x] 给出纯函数解, 一般来说会更方面后期使用.
- DSolve[方程 $\&\&$ 边界条件, $y[x]$, x] 给出在边界条件下的解.
- NDSolve[方程, y , $\{x, xmin, xmax\}$] 给出方程的数值解, 其中 x 满足给定条件.
- 对于一些方程, 可以先做拉普拉斯变换或傅立叶变换再求解. LaplaceTransform[], FourierTransform[]
- 更多关于各类微分方程的求解参看关于 DSolve 帮助文档的应用章节

第四节

Mathematica 基础

4.1 基本语法

4.2 基本概念

4.3 解方程

4.4 微积分

4.5 微分方程

4.6 线性代数

4.7 练习

向量与矩阵

■ 向量：构建 n 维向量

向量与矩阵

■ 向量：构建 n 维向量

- `Table[表达式, {i,n}]`

向量与矩阵

■ 向量：构建 n 维向量

- `Table[表达式, {i,n}]`
- `Array[f,n]`

向量与矩阵

- 向量: 构建 n 维向量
 - `Table[表达式, {i,n}]`
 - `Array[f,n]`
- 矩阵: 构造一个 $m \times n$ 阶矩阵

向量与矩阵

- 向量：构建 n 维向量
 - `Table[表达式, {i,n}]`
 - `Array[f,n]`
- 矩阵：构造一个 $m \times n$ 阶矩阵
 - `Table[表达式, {i,m}, {j,n}]`

向量与矩阵

- 向量：构建 n 维向量
 - `Table[表达式, {i,n}]`
 - `Array[f,n]`
- 矩阵：构造一个 $m \times n$ 阶矩阵
 - `Table[表达式, {i,m}, {j,n}]`
 - `Array[f, {m,n}]`

向量与矩阵

■ 向量：构建 n 维向量

- `Table[表达式, {i,n}]`
- `Array[f,n]`

■ 矩阵：构造一个 $m \times n$ 阶矩阵

- `Table[表达式, {i,m}, {j,n}]`
- `Array[f, {m,n}]`
- `DiagonalMatrix[列表]`

向量与矩阵

- 向量：构建 n 维向量
 - `Table[表达式, {i,n}]`
 - `Array[f,n]`
- 矩阵：构造一个 $m \times n$ 阶矩阵
 - `Table[表达式, {i,m}, {j,n}]`
 - `Array[f, {m,n}]`
 - `DiagonalMatrix[列表]`
 - `IdentityMatrix[n]`

矩阵运算

■ $m1 \pm m2$ 矩阵加减法

矩阵运算

- $m1 \pm m2$ 矩阵加减法
- $c m$ 矩阵每个元素乘上常数 c

矩阵运算

- $m1 \pm m2$ 矩阵加减法
- $c m$ 矩阵每个元素乘上常数 c
- $m1 \cdot m2$ 两个矩阵 (矢量) 进行点乘

矩阵运算

- $m1 \pm m2$ 矩阵加减法
- $c m$ 矩阵每个元素乘上常数 c
- $m1 \cdot m2$ 两个矩阵 (矢量) 进行点乘
- $\text{Cross}[v1, v2]$ 两个矢量叉乘

矩阵运算

■ Inverse[矩阵] 计算矩阵的逆矩阵

矩阵运算

- Inverse[矩阵] 计算矩阵的逆矩阵
- Det[矩阵] 计算矩阵的行列式

矩阵运算

- Inverse[矩阵] 计算矩阵的逆矩阵
- Det[矩阵] 计算矩阵的行列式
- Transpose[矩阵] 计算矩阵的转置

矩阵运算

- Inverse[矩阵] 计算矩阵的逆矩阵
- Det[矩阵] 计算矩阵的行列式
- Transpose[矩阵] 计算矩阵的转置
- Tr[矩阵] 计算矩阵的迹

矩阵运算

- Inverse[矩阵] 计算矩阵的逆矩阵
- Det[矩阵] 计算矩阵的行列式
- Transpose[矩阵] 计算矩阵的转置
- Tr[矩阵] 计算矩阵的迹
- MatrixPower[矩阵, n] 计算矩阵的 n 次方

矩阵运算

- Inverse[矩阵] 计算矩阵的逆矩阵
- Det[矩阵] 计算矩阵的行列式
- Transpose[矩阵] 计算矩阵的转置
- Tr[矩阵] 计算矩阵的迹
- MatrixPower[矩阵, n] 计算矩阵的 n 次方
- Minors[矩阵, k] 生成一个矩阵，其元素由矩阵所有的 k 阶子矩阵的行列式组成.

矩阵运算

- Inverse[矩阵] 计算矩阵的逆矩阵
- Det[矩阵] 计算矩阵的行列式
- Transpose[矩阵] 计算矩阵的转置
- Tr[矩阵] 计算矩阵的迹
- MatrixPower[矩阵, n] 计算矩阵的 n 次方
- Minors[矩阵, k] 生成一个矩阵，其元素由矩阵所有的 k 阶子矩阵的行列式组成.
- RowReduce[矩阵] 把矩阵化为行梯型形式

矩阵运算

- Inverse[矩阵] 计算矩阵的逆矩阵
- Det[矩阵] 计算矩阵的行列式
- Transpose[矩阵] 计算矩阵的转置
- Tr[矩阵] 计算矩阵的迹
- MatrixPower[矩阵, n] 计算矩阵的 n 次方
- Minors[矩阵, k] 生成一个矩阵，其元素由矩阵所有的 k 阶子矩阵的行列式组成.
- RowReduce[矩阵] 把矩阵化为行梯型形式
- Eigensystem[矩阵] 给出矩阵的 {特征值, 特征向量}

第四节

Mathematica 基础

4.1 基本语法

4.2 基本概念

4.3 解方程

4.4 微积分

4.5 微分方程

4.6 线性代数

4.7 练习

单摆

例 2 研究无阻尼下的单摆运动，假设质量为 m ，线长为 L ，讨论小角近似下的解析解及完全的数值解。

电力线

例3 在二维空间中，电力线的方程为 $\frac{dy}{dx} = \frac{E_y(x,y)}{E_x(x,y)}$ ，若已知电场强度的分布函数，根据方程求解电力线方程 $f(x,y)=0$. 即可得到电力线. 分别画出单个点电荷，两个点电荷的电力线.

第三节 Python 语言基础

第四节 Mathematica 基础

第五节 函数近似

第六节 数值微积分

第七节 常微分方程

第五节

函数近似

5.1

插值法

5.2

最小二乘近似

5.3

密立根油滴实验

5.4

样条插值

Interpolation

- 数值分析中，计算结果通常是对准确值的近似，带有一定的误差。类似于物理实验。

Interpolation

- 数值分析中，计算结果通常是对准确值的近似，带有一定的误差。类似于物理实验。
- 通常我们需要从一组离散的数据点取获取函数的信息，以此来得到新的理论预言。

Interpolation

- 数值分析中，计算结果通常是对准确值的近似，带有一定的误差。类似于物理实验。
- 通常我们需要从一组离散的数据点取获取函数的信息，以此来得到新的理论预言。
- 需要从不完整的信息推断局部的信息，需要插值。

Interpolation

- 数值分析中，计算结果通常是对准确值的近似，带有一定的误差。类似于物理实验。
- 通常我们需要从一组离散的数据点取获取函数的信息，以此来得到新的理论预言。
- 需要从不完整的信息推断局部的信息，需要插值。
- 例如，我们可以记录每隔 0.01 秒一个棒球的速度，从而给出其速度函数。

Linear interpolation

- 考虑一组离散的数据点 y_i . 最简单给出该数据的函数近似为,

$$f(x) = y_i + \frac{x - x_i}{x_{i+1} - x_i} (y_{i+1} - y_i) + \Delta f(x)$$

Linear interpolation

- 考虑一组离散的数据点 y_i . 最简单给出该数据的函数近似为,

$$f(x) = y_i + \frac{x - x_i}{x_{i+1} - x_i} (y_{i+1} - y_i) + \Delta f(x)$$

- 其中线性插值的误差 $\Delta f(x)$ 为

$$\Delta f(x) = \frac{\gamma}{2} (x - x_i)(x - x_{i+1})$$

Linear interpolation

- 考虑一组离散的数据点 y_i . 最简单给出该数据的函数近似为,

$$f(x) = y_i + \frac{x - x_i}{x_{i+1} - x_i} (y_{i+1} - y_i) + \Delta f(x)$$

- 其中线性插值的误差 $\Delta f(x)$ 为

$$\Delta f(x) = \frac{\gamma}{2} (x - x_i)(x - x_{i+1})$$

- 其中 $\gamma = f''(\xi)$, $|\Delta f(x)| \leq \frac{\gamma_1}{8} (x_{i+1} - x_i)^2$

Linear interpolation

- 虽然我们可以加少 $h_i = x_{i+1} - x_i$, 来增加精度, 但并不实际.

Linear interpolation

- 虽然我们可以加少 $h_i = x_{i+1} - x_i$, 来增加精度, 但并不实际.
- 例题: 取 $f(x) = \sin x$. 设 $x_i = \pi/4, x_{i+1} = \pi/2$, 我们有 $f_i = 0.707$ 和 $f_{i+1} = 1.000$. 试试利用线性插值计算 $f(3\pi/8)$ 的结果?

Linear interpolation

- 虽然我们可以加少 $h_i = x_{i+1} - x_i$, 来增加精度, 但并不实际.
- 例题: 取 $f(x) = \sin x$. 设 $x_i = \pi/4, x_{i+1} = \pi/2$, 我们有 $f_i = 0.707$ 和 $f_{i+1} = 1.000$. 试试利用线性插值计算 $f(3\pi/8)$ 的结果?
- 该例子虽然简单但给出了函数插值的一般想法.

The Lagrange interpolation

- 如果我们有三个数据点，我们就可以构建一个二次函数来穿过这三个点.

$$f(x) = ax^2 + bx + c$$

The Lagrange interpolation

- 如果我们有三个数据点，我们就可以构建一个二次函数来穿过这三个点.

$$f(x) = ax^2 + bx + c$$

- 需要三个插值条件：插值区间 $[x_{i-1}, x_{i+1}]$ 的已知端点值等于函数值

The Lagrange interpolation

- 如果我们有三个数据点，我们就可以构建一个二次函数来穿过这三个点.

$$f(x) = ax^2 + bx + c$$

- 需要三个插值条件：插值区间 $[x_{i-1}, x_{i+1}]$ 的已知端点值等于函数值
- 求解 a, b, c 代入插值函数得：

$$f(x) = l_{i-1}(x)y_{i-1} + l_i(x)y_i + l_{i+1}(x)y_{i+1} \quad x_{i-1} \leq x \leq x_{i+1}$$

The Lagrange interpolation



$$l_{i-1}(x) = \frac{(x - x_i)(x - x_{i+1})}{(x_{i-1} - x_i)(x_{i-1} - x_{i+1})}$$

$$l_i(x) = \frac{(x - x_{i-1})(x - x_{i+1})}{(x_i - x_{i-1})(x_i - x_{i+1})}$$

$$l_{i+1}(x) = \frac{(x - x_{i-1})(x - x_i)}{(x_{i+1} - x_{i-1})(x_{i+1} - x_i)}$$

The Lagrange interpolation

■ 我们尝试推广到更多. 首先我们把线性插值的表达式重新写一下

$$\begin{aligned}f(x) &= \frac{x - x_{i+1}}{x_i - x_{i+1}} f_i + \frac{x - x_i}{x_{i+1} - x_i} f_{i+1} + \Delta f(x) \\&= \sum_{j=i}^{i+1} f_j p_{1j}(x) + \Delta f(x)\end{aligned}$$

$$\text{其中 } p_{1j}(x) = \frac{x - x_k}{x_j - x_k}$$

The Lagrange interpolation

- 现在我们可以很容易推广到高阶曲线的表示

$$f(x) = \sum_{j=0}^n f_j p_{nj}(x) + \Delta f(x)$$

$$p_{nj}(x) = \prod_{k \neq j}^n \frac{x - x_k}{x_j - x_k}$$

The Lagrange interpolation

- 现在我们可以很容易推广到高阶曲线的表示

$$f(x) = \sum_{j=0}^n f_j p_{nj}(x) + \Delta f(x)$$

$$p_{nj}(x) = \prod_{k \neq j}^{n-1} \frac{x - x_k}{x_j - x_k}$$

- 练一练：证明二阶可以表示成这种形式

The Lagrange interpolation

- 显然 $\Delta f(x)$ 在每个数据点等于零. 利用泰勒展开可以证明

$$\Delta f(x) = \frac{\gamma}{(n+1)!} (x - x_0)(x - x_1) \cdots (x - x_n)$$
$$\gamma = f^{(n+1)}(a), \quad a \in [x_0, x_n]$$

The Lagrange interpolation

- 显然 $\Delta f(x)$ 在每个数据点等于零. 利用泰勒展开可以证明

$$\Delta f(x) = \frac{\gamma}{(n+1)!} (x - x_0)(x - x_1) \cdots (x - x_n)$$
$$\gamma = f^{(n+1)}(a), a \in [x_0, x_n]$$

- 同样可以估计出最大误差为

$$|\Delta f(x)| \leq \frac{\gamma_n}{4(n+1)} h^{n+1} \quad \gamma_n = \max[|f^{n+1}(x)|]$$

The Aitken method

- Aitken 最早给出了一种利用连续的线性插值来实现拉格朗日插值.

The Aitken method

- Aitken 最早给出了一种利用连续的线性插值来实现拉格朗日插值.
- 首先对每两个点之间采用线性插值，得到 n 个线性插值函数.

The Aitken method

- Aitken 最早给出了一种利用连续的线性插值来实现拉格朗日插值.
- 首先对每两个点之间采用线性插值，得到 n 个线性插值函数.
- 然后，我们使用这 N 个插值数据点和下一个相邻点 x_i 来实现另 $n - 1$ 个线性插值

The Aitken method

- Aitken 最早给出了一种利用连续的线性插值来实现拉格朗日插值.
- 首先对每两个点之间采用线性插值，得到 n 个线性插值函数.
- 然后，我们使用这 N 个插值数据点和下一个相邻点 x_i 来实现另 $n - 1$ 个线性插值
- 重复这个过程，直到我们得到 n 阶的插值函数.

The Aitken method

■ 下面的公式总结了这个方案的过程

$$f_{i \dots j} = \frac{x - x_j}{x_i - x_j} f_{i \dots j-1} + \frac{x - x_i}{x_j - x_i} f_{i+1 \dots j}$$

$$f_i = y_i$$

The Aitken method

f_0

f_{01}

f_1

...

f_{12}

$f_{0\dots n-1}$

f_2

...

$f_{01\dots n}$

:

$f_{12\dots n}$

:

...

f_{n-1n}

f_n

The Aitken method

■ 例如

$$f_{012} = \frac{x - x_2}{x_0 - x_2} f_{01} + \frac{x - x_0}{x_2 - x_0} f_{12}$$

The Aitken method

■ 例如

$$f_{012} = \frac{x - x_2}{x_0 - x_2} f_{01} + \frac{x - x_0}{x_2 - x_0} f_{12}$$

■

$$f_{01234} = \frac{x - x_4}{x_0 - x_4} f_{0123} + \frac{x - x_0}{x_4 - x_0} f_{1234}$$

The Aitken method

■ 例如

$$f_{012} = \frac{x - x_2}{x_0 - x_2} f_{01} + \frac{x - x_0}{x_2 - x_0} f_{12}$$

■

$$f_{01234} = \frac{x - x_4}{x_0 - x_4} f_{0123} + \frac{x - x_0}{x_4 - x_0} f_{1234}$$

■ 练习：验证 f_{012} 等价于三点插值的二次函数.

The Aitken method

- Aitken 方法同时可以给出拉格朗日插值的误差估计.

The Aitken method

- Aitken 方法同时可以给出拉格朗日插值的误差估计.
- 如果我们用 5 点插值，则其误差估计为

$$\Delta f(x) \approx \frac{|f_{01234} - f_{0123}| + |f_{01234} - f_{1234}|}{2}$$

The Aitken method

■ 练习，给定五个数据点计算出 $f(0.9)$ 的 Aitken 插值各项，并给出误差估计.

x_i	f_i	f_{ij}	f_{ijk}	f_{ijkl}	f_{ijklm}
0.0	1.000 000				
		0.889 246			
0.5	0.938 470		0.808 792		
		0.799 852		0.807 272	
1.0	0.765 198		0.806 260		0.807 473
		0.815 872		0.807 717	
1.5	0.511 828		0.811 725		
		0.857 352			
2.0	0.223 891				

The Aitken method

- 练习，给定五个数据点计算出 $f(0.9)$ 的 Aitken 插值各项，并给出误差估计.

x_i	f_i	f_{ij}	f_{ijk}	f_{ijkl}	f_{ijklm}
0.0	1.000 000				
		0.889 246			
0.5	0.938 470		0.808 792		
		0.799 852		0.807 272	
1.0	0.765 198		0.806 260		0.807 473
		0.815 872		0.807 717	
1.5	0.511 828		0.811 725		
		0.857 352			
2.0	0.223 891				

- $f(0.9) = 0.807524$, 零阶 Bessel 函数.

The Aitken method

- 虽然 Aitken 方法在许多情况下给出的精度都不错，但是某些情况下，插值值与实际值变化非常小，数据点非常多时，带来的舍入误差，累积较大。

The Aitken method

- 虽然 Aitken 方法在许多情况下给出的精度都不错，但是某些情况下，插值值与实际值变化非常小，数据点非常多时，带来的舍入误差，累积较大。
- 一个改进的方案是，通过更新相邻列中插值的值的差异来改进每一步的插值值。该方法与上下法相结合，即利用沿三角形的上下两个方向的修正：

$$\Delta_{ij}^+ = f_{j,\dots,j+i} - f_{j+1,\dots,j+i}$$

$$\Delta_{ij}^- = f_{j,\dots,j+i} - f_{j,\dots,j+i-1}$$

The Aitken method

- Δ_{ij}^- 代表下修正, Δ_{ij}^+ 代表上修正. i 指标代表着修正的阶, j 指标代表该阶的元素.

The Aitken method

- Δ_{ij}^- 代表下修正, Δ_{ij}^+ 代表上修正. i 指标代表着修正的阶, j 指标代表该阶的元素.
- 从他们的定义可以发现其满足

$$\Delta_{ij}^+ = \frac{x_{i+j} - x}{x_{i+j} - x_j} (\Delta_{i-1j}^+ - \Delta_{i-1j+1}^-)$$

$$\Delta_{ij}^- = \frac{x_j - x}{x_{i+j} - x_j} (\Delta_{i-1j}^+ - \Delta_{i-1j+1}^-)$$

The Aitken method

- Δ_{ij}^- 代表下修正, Δ_{ij}^+ 代表上修正. i 指标代表着修正的阶, j 指标代表该阶的元素.
- 从他们的定义可以发现其满足

$$\Delta_{ij}^+ = \frac{x_{i+j} - x}{x_{i+j} - x_j} (\Delta_{i-1j}^+ - \Delta_{i-1j+1}^-)$$

$$\Delta_{ij}^- = \frac{x_j - x}{x_{i+j} - x_j} (\Delta_{i-1j}^+ - \Delta_{i-1j+1}^-)$$

- 其中 $\Delta_{oj}^\pm = f_j$. 这里 x_j 选择为最靠近 x 的数据点.

The Aitken method

Δ_{00}

Δ_{10}

Δ_{01}

...

Δ_{11}

Δ_{n-10}

Δ_{02}

...

Δ_{n0}

:

Δ_{n-11}

:

...

Δ_{1n-1}

Δ_{0n}

The Aitken method

- 同样的以 Bessel 函数为例. 起始点选择 $x_j = 1.0$.
因此零阶近似为 $f(x) \simeq f(1.0)$.

The Aitken method

- 同样的以 Bessel 函数为例. 起始点选择 $x_j = 1.0$.
因此零阶近似为 $f(x) \simeq f(1.0)$.
- $f(x)$ 的一阶修正为 Δ_{11}^+ . 接下来, 我们选择向下修正, 则应为 Δ_{21}^- .

The Aitken method

- 同样的以 Bessel 函数为例. 起始点选择 $x_j = 1.0$.
因此零阶近似为 $f(x) \simeq f(1.0)$.
- $f(x)$ 的一阶修正为 Δ_{11}^+ . 接下来, 我们选择向下修正, 则应为 Δ_{21}^- .
- 重复该过程, 再利用向上修正, 接下来再向下修正,
一直到最终结果.

The Aitken method

- 同样的以 Bessel 函数为例. 起始点选择 $x_j = 1.0$.
因此零阶近似为 $f(x) \simeq f(1.0)$.
- $f(x)$ 的一阶修正为 Δ_{11}^+ . 接下来, 我们选择向下修正, 则应为 Δ_{21}^- .
- 重复该过程, 再利用向上修正, 接下来再向下修正,
一直到最终结果.
- 练习: 写代码实现该算法. 并给出 $f(0.9)$.

拉格朗日插值

- 拉格朗日插值公式的特点是插值公式的系数依赖于全部插值基点，每增加一个基点，所有的系数都要重算.（这时可以考虑使用牛顿插值法. 见《计算物理学》-顾昌鑫 p15）

拉格朗日插值

- 拉格朗日插值公式的特点是插值公式的系数依赖于全部插值基点，每增加一个基点，所有的系数都要重算.（这时可以考虑使用牛顿插值法. 见《计算物理学》-顾昌鑫 p15）
- 拉格朗日多项式插值并不是插值点越多越好. 因为插值点过多时，会出现插值基函数正负剧烈振荡的情况

拉格朗日插值

- 拉格朗日插值公式的特点是插值公式的系数依赖于全部插值基点，每增加一个基点，所有的系数都要重算.（这时可以考虑使用牛顿插值法. 见《计算物理学》-顾昌鑫 p15）
- 拉格朗日多项式插值并不是插值点越多越好. 因为插值点过多时，会出现插值基函数正负剧烈振荡的情况
- 使用插值函数时，最好进行分段插值，而不是全区间插值.

第五节

函数近似

5.1

插值法

5.2

最小二乘近似

5.3

密立根油滴实验

5.4

样条插值

Least-squares approximation

- 插值主要用于寻找给定数据点的局部近似.

Least-squares approximation

- 插值主要用于寻找给定数据点的局部近似.
- 在许多例子中，我们需要知道数据的整体行为，以便了解整个观测的趋势.

Least-squares approximation

- 插值主要用于寻找给定数据点的局部近似.
- 在许多例子中，我们需要知道数据的整体行为，以便了解整个观测的趋势.
- 典型的例子如用多项式函数去拟合一组带有误差条的实验数据.

Least-squares approximation

- 最常见的近似方案则是基于数据与近似函数间的最小二乘法.

Least-squares approximation

- 最常见的近似方案则是基于数据与近似函数间的最小二乘法.
- 假设 $f(x)$ 为一组数据, 近似的函数为 m 阶多项式函数

$$p_m(x) = \sum_{k=0}^m a_k x^k$$

Least-squares approximation

■ 构建关于 a_k 的函数, 如果 $f(x)$ 为连续函数则:

$$\chi^2 [a_k] = \int_a^b [p_m(x) - f(x)]^2 dx$$

Least-squares approximation

- 构建关于 a_k 的函数, 如果 $f(x)$ 为连续函数则:

$$\chi^2 [a_k] = \int_a^b [p_m(x) - f(x)]^2 dx$$

- 若 $f(x)$ 为离散的数据点则

$$\chi^2 [a_k] = \sum_{i=0}^n [p_m(x_i) - f(x_i)]^2$$

Least-squares approximation

- 最小二乘近似，则是取 $\chi^2[a_k]$ 关于 a_k 取极小值的情形

$$\frac{\partial \chi^2[a_k]}{\partial a_l} = 0$$

Least-squares approximation

- 最小二乘近似，则是取 $\chi^2[a_k]$ 关于 a_k 取极小值的情形

$$\frac{\partial \chi^2[a_k]}{\partial a_l} = 0$$

- 任务则转化为求解 $m + 1$ 个线性方程来给出 a_l . 后面我们会给出更一般的求解线性方程的方法.

Least-squares approximation

■ 我们考虑 $m = 1$ 的情况，即

$$p_1(x) = a_0 + a_1 x$$

Least-squares approximation

- 我们考虑 $m = 1$ 的情况，即

$$p_1(x) = a_0 + a_1 x$$

- 此时

$$\chi^2 [a_k] = \sum_{i=0}^n (a_0 + a_1 x_i - f_i)^2$$

Least-squares approximation

■ 根据最小二乘的定义，给出参数方程为

$$(n + 1)a_0 + c_1 a_1 - c_3 = 0$$

$$c_1 a_0 + c_2 a_1 - c_4 = 0$$

Least-squares approximation

- 根据最小二乘的定义，给出参数方程为

$$(n + 1)a_0 + c_1 a_1 - c_3 = 0$$

$$c_1 a_0 + c_2 a_1 - c_4 = 0$$

- 其中 $c_1 = \sum_{i=0}^n x_i$, $c_2 = \sum_{i=0}^n x_i^2$, $c_3 = \sum_{i=0}^n f_i$, $c_4 = \sum_{i=0}^n x_i f_i$

Least-squares approximation

- 我们改变一下策略，选用正交多项式来求解。理论上，我们总是可以将多项式写成一组正交多项式

$$p_m(x) = \sum_{k=0}^m \alpha_k u_k(x)$$

Least-squares approximation

- 我们改变一下策略，选用正交多项式来求解。理论上，我们总是可以将多项式写成一组正交多项式

$$p_m(x) = \sum_{k=0}^m \alpha_k u_k(x)$$

- $u_k(x)$ 为一组正交多项式，满足

$$\int_a^b u_k(x) w(x) u_l(x) dx = \langle u_k | u_l \rangle = \delta_{kl} N_k$$

其中 $w(x)$ 为权重，取决于正交多项式的类型。 N_k 为归一化常数。

Least-squares approximation

- 系数 a_k 形式上可以通过对 a_l 经过一个矩阵变换得到.

Least-squares approximation

- 系数 a_k 形式上可以通过对 a_l 经过一个矩阵变换得到.
- 另外，对于正交函数，我们可以通过如下的方法选取

$$u_{k+1}(x) = (x - g_k)u_k(x) - h_k u_{k-1}(x)$$

Least-squares approximation

- 系数 a_k 形式上可以通过对 a_l 经过一个矩阵变换得到.
- 另外，对于正交函数，我们可以通过如下的方法选取

$$u_{k+1}(x) = (x - g_k)u_k(x) - h_k u_{k-1}(x)$$

- 其中我们定义

$$g_k = \frac{\langle xu_k | u_k \rangle}{\langle u_k | u_k \rangle}$$

$$h_k = \frac{\langle xu_k | u_{k-1} \rangle}{\langle u_{k-1} | u_{k-1} \rangle}$$

Least-squares approximation

- 我们取 $u_0(x) = 1$ 以及 $h_0 = 0$, 从而可以产生出所需要的正交多项式.

Least-squares approximation

- 我们取 $u_0(x) = 1$ 以及 $h_0 = 0$, 从而可以产生出所需要的正交多项式.
- 该方法产生的多项式无论其是连续情况还是离散情况, 都是满足正交性.

Least-squares approximation

- 我们取 $u_0(x) = 1$ 以及 $h_0 = 0$, 从而可以产生出所需要的正交多项式.
- 该方法产生的多项式无论其是连续情况还是离散情况, 都是满足正交性.
- 同样的, 我们最小化 $\chi^2[a_k]$ 函数, 寻找参数集满足

$$\frac{\partial \chi^2[a_k]}{\partial a_j} = 0$$

且二次导数大于零.

Least-squares approximation

- $\chi^2[\alpha_k]$ 的一阶导数很容易可以得出. 通过交换求和顺序并对 $\partial\chi^2[\alpha_k]/\partial\alpha_j = 0$ 进行积分, 可以得到

$$\alpha_j = \frac{\langle u_j | f \rangle}{\langle u_j | u_j \rangle}$$

Least-squares approximation

- $\chi^2[\alpha_k]$ 的一阶导数很容易可以得出. 通过交换求和顺序并对 $\partial\chi^2[\alpha_k]/\partial\alpha_j = 0$ 进行积分, 可以得到

$$\alpha_j = \frac{\langle u_j | f \rangle}{\langle u_j | u_j \rangle}$$

- 由于 $\partial^2\chi^2[\alpha_k]/\partial\alpha_j^2 = 2\langle u_j | u_j \rangle$, 保证我们找到的参数满足极小化 χ^2 .

Least-squares approximation

- $\chi^2[\alpha_k]$ 的一阶导数很容易可以得出. 通过交换求和顺序并对 $\partial\chi^2[\alpha_k]/\partial\alpha_j = 0$ 进行积分, 可以得到

$$\alpha_j = \frac{\langle u_j | f \rangle}{\langle u_j | u_j \rangle}$$

- 由于 $\partial^2\chi^2[\alpha_k]/\partial\alpha_j^2 = 2\langle u_j | u_j \rangle$, 保证我们找到的参数满足极小化 χ^2 .
- 练习: 写出产生正交多项式及计算最小二次系数的程序.

第五节

函数近似

5.1

插值法

5.2

最小二乘近似

5.3

密立根油滴实验

5.4

样条插值

密立根实验

■ 我们以有名的密立根油滴实验来说明最小二乘法近似.

k	4	5	6	7	8	9	10	11
q_k	6.558	8.206	9.880	11.50	13.14	14.82	16.40	18.
k	12	13	14	15	16	17	18	
q_k	19.68	21.32	22.96	24.60	26.24	27.88	29.52	

密立根实验

- 我们以有名的密立根油滴实验来说明最小二乘法近似.

k	4	5	6	7	8	9	10	11
q_k	6.558	8.206	9.880	11.50	13.14	14.82	16.40	18.
k	12	13	14	15	16	17	18	
q_k	19.68	21.32	22.96	24.60	26.24	27.88	29.52	

- 密立根对每个测量的数据赋予了一个整数 k , q_k 为测量的电量.

密立根实验

- 密立根得到的结论是基本电荷单元是 $e = 1.65 \times 10^{-19} C.$

密立根实验

- 密立根得到的结论是基本电荷单元是 $e = 1.65 \times 10^{-19} C$.
- 我们根据其实验结构，利用最小二乘法来寻找基本电荷.

密立根实验

- 密立根得到的结论是基本电荷单元是 $e = 1.65 \times 10^{-19} C$.
- 我们根据其实验结构，利用最小二乘法来寻找基本电荷.
- 我们取线性方程

$$q_k = ke + \Delta q_k$$

作为测量数据的近似. 为了简化， Δq_k 设为常数.

密立根实验

- 密立根得到的结论是基本电荷单元是 $e = 1.65 \times 10^{-19} C.$
- 我们根据其实验结构，利用最小二乘法来寻找基本电荷.
- 我们取线性方程

$$q_k = ke + \Delta q_k$$

作为测量数据的近似. 为了简化， Δq_k 设为常数.

- 练习：请尝试用以上两种线性基函数来拟合给出 e 以及误差 Δq .

第五节

函数近似

5.1 插值法

5.2 最小二乘近似

5.3 密立根油滴实验

5.4 样条插值

样条近似

- 有些时候，数据可能在小范围内快速震荡，此时很难用整体的多项式来描述

样条近似

- 有些时候，数据可能在小范围内快速震荡，此时很难用整体的多项式来描述
- 这是，我们希望局部的去拟合函数并且希望在连接处函数也是光滑的.

样条近似

- 有些时候，数据可能在小范围内快速震荡，此时很难用整体的多项式来描述
- 这是，我们希望局部的去拟合函数并且希望在连接处函数也是光滑的.
- 样条 (spline) 就是这样的工具，它对数据点进行局部的多项式插值，每段多项式函数要求在数据点处函数值及导数值相等，从而来整体拟合数据点.

三次样条插值

- 三次样条插值：在区间 x_i, x_{i+1} 上构造三次多项式插值函数 $S_i(x)$ ，满足

三次样条插值

- 三次样条插值：在区间 x_i, x_{i+1} 上构造三次多项式插值函数 $S_i(x)$ ，满足
 - 插值函数在节点的函数值等于节点值

三次样条插值

- 三次样条插值：在区间 x_i, x_{i+1} 上构造三次多项式插值函数 $S_i(x)$, 满足
 - 插值函数在节点的函数值等于节点值
 - 插值函数在区间或其边界上一阶导数平滑, 二阶导数连续.

三次样条插值

- 三次样条插值: 在区间 x_i, x_{i+1} 上构造三次多项式插值函数 $S_i(x)$, 满足
 - 插值函数在节点的函数值等于节点值
 - 插值函数在区间或其边界上一阶导数平滑, 二阶导数连续.
- 特点: 插值函数光滑且连续.

三次样条插值

- 二阶导数为 $S_i''(x) = a_i x + b_i$

三次样条插值

- 二阶导数为 $S_i''(x) = a_i x + b_i$

- 区间端点处满足

$$a_i x_i + b_i = y_i'', \quad a_i x_{i+1} + b_i = y_{i+1}''$$

三次样条插值

■ 二阶导数为 $S_i''(x) = a_i x + b_i$

■ 区间端点处满足

$$a_i x_i + b_i = y_i'', \quad a_i x_{i+1} + b_i = y_{i+1}''$$

■ 可得 $a_i = \frac{y_{i+1}'' - y_i''}{x_{i+1} - x_i}$, $b_i = \frac{x_{i+1} y_i'' - x_i y_{i+1}''}{x_{i+1} - x_i}$

三次样条插值

■ 二阶导数为 $S_i''(x) = a_i x + b_i$

■ 区间端点处满足

$$a_i x_i + b_i = y_i'', \quad a_i x_{i+1} + b_i = y_{i+1}''$$

■ 可得 $a_i = \frac{y_{i+1}'' - y_i''}{x_{i+1} - x_i}$, $b_i = \frac{x_{i+1} y_i'' - x_i y_{i+1}''}{x_{i+1} - x_i}$

■ 由函数值在端点值等于节点值可得

$$S_i(x_i) = \frac{1}{6} a_i x_i^3 + \frac{1}{2} b_i x_i^2 + c_i x_i + d_i = y_i$$

$$S_i(x_{i+1}) = \frac{1}{6} a_i x_{i+1}^3 + \frac{1}{2} b_i x_{i+1}^2 + c_i x_{i+1} + d_i = y_{i+1}$$

该等式可给出 c_i, d_i 与 a_i, b_i 的关系式

三次样条插值

- 利用内点上一阶导数连续的条件 $S'_{i-1}(x_i) = S'_i(x_i)$ 可得：

$$\frac{1}{2}a_{i-1}x_i^2 + b_{i-1}x_i + c_{i-1} = \frac{1}{2}a_i x_i^2 + b_i x_i + c_i$$

三次样条插值

- 利用内点上一阶导数连续的条件 $S'_{i-1}(x_i) = S'_i(x_i)$ 可得：

$$\frac{1}{2}a_{i-1}x_i^2 + b_{i-1}x_i + c_{i-1} = \frac{1}{2}a_i x_i^2 + b_i x_i + c_i$$

- 代入 $a_{i-1}, b_{i-1}, c_{i-1}, a_i, b_i, c_i$ 可得

$$\alpha_{i-1}y''_{i-1} + \beta_i y''_i + \gamma_i y''_{i+1} = f_i$$

三次样条插值

- 其中 $\alpha_{i-1} = x_i - x_{i-1}$, $\beta_i = 2(x_{i+1} - x_{i-1})$, $\gamma_i = x_{i-1} - x_i$, $f_i = 6\left(\frac{y_{i+1}-y_i}{\gamma_i} - \frac{y_i-y_{i-1}}{\alpha_{i-1}}\right)$, $i = 1, \dots, n-2$

三次样条插值

■ 其中 $\alpha_{i-1} = x_i - x_{i-1}$, $\beta_i = 2(x_{i+1} - x_{i-1})$, $\gamma_i = x_{i-1} - x_i$, $f_i = 6(\frac{y_{i+1}-y_i}{\gamma_i} - \frac{y_i-y_{i-1}}{\alpha_{i-1}})$, $i = 1, \dots, n-2$



$$\begin{pmatrix} \beta_0 & \gamma_0 & & \\ \alpha_0 & \beta_1 & \gamma_1 & \\ \dots & \dots & \dots & \\ & & & \alpha_{n-3} & \beta_{n-2} & \gamma_{n-2} \\ & & & \alpha_{n-2} & \beta_{n-1} & \end{pmatrix} \begin{pmatrix} y_0'' \\ y_1'' \\ \dots \\ y_{n-2}'' \\ y_{n-1}'' \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ \dots \\ f_{n-2} \\ f_{n-1} \end{pmatrix}$$

三次样条插值

- 该方程为三对角矩阵方程，还需两个条件来确定所有的未知数。常见的两种选择方法：

三次样条插值

■ 该方程为三对角矩阵方程，还需两个条件来确定所有的未知数。常见的两种选择方法：

- 1 给定两个端点出的一阶导数 y'_0, y'_{n-1} ，则可以得到两组关于 $y''_1, y''_2, y''_{n-2}, y''_{n-1}$ 的方程

三次样条插值

■ 该方程为三对角矩阵方程，还需两个条件来确定所有的未知数。常见的两种选择方法：

- 1 给定两个端点出的一阶导数 y'_0, y'_{n-1} , 则可以得到两组关于 $y''_1, y''_2, y''_{n-2}, y''_{n-1}$ 的方程
- 2 设两端点的二阶导数为零, 即取 $\beta_0, \beta_{n-1} = 1, \gamma_0 = \alpha_{n-2} = f_1 = f_n = 0$, 利用三对角矩阵追赶法求解

作业题

■ 3.1.8

作业题

- 3.1.8
- 3.1.9

作业题

- 3.1.8
- 3.1.9
- 3.2.4

- 2.3 The Newton interpolation is another popular interpolation scheme that adopts the polynomial

$$p_n(x) = \sum_{j=0}^n c_j \prod_{i=0}^{j-1} (x - x_i),$$

where $\prod_{i=0}^{-1} (x - x_i) = 1$. Show that this polynomial is equivalent to that of the Lagrange interpolation and the coefficients c_j here are recursively given by

$$c_j = \frac{f_j - \sum_{k=0}^{j-1} c_k \prod_{i=0}^{k-1} (x_j - x_i)}{\prod_{i=0}^{j-1} (x_j - x_i)}.$$

Write a subprogram that creates all c_j with given x_i and f_i .

第四节

Mathematica 基础

第五节

函数近似

第六节

数值微积分

第七节

常微分方程

第八节

矩阵的数值方法

数值微积分

- 微积分是描述物理现象的核心. 一旦我们涉及物体的运动, 一定会涉及到微分和积分.

数值微积分

- 微积分是描述物理现象的核心. 一旦我们涉及物体的运动, 一定会涉及到微分和积分.
- 介绍处理数值微分的积分的基本数值方法

数值微积分

- 微积分是描述物理现象的核心. 一旦我们涉及物体的运动, 一定会涉及到微分和积分.
- 介绍处理数值微分的积分的基本数值方法
- 寻找方程根的数值方法

数值微积分

- 微积分是描述物理现象的核心. 一旦我们涉及物体的运动, 一定会涉及到微分和积分.
- 介绍处理数值微分的积分的基本数值方法
- 寻找方程根的数值方法
- 函数的极值

第六节

数值微积分

6.1 数值微分

6.2 数值积分

6.3 方程的根

6.4 函数极值

6.5 经典散射

数值微分

■ 基本的工具-泰勒展开

$$f(x) = f(x_0) + (x - x_0)f'(x_0)$$

$$+ \frac{(x - x_0)^2}{2!} f''(x_0)$$

$$+ \cdots + \frac{(x - x_0)^n}{n!} f^{(n)}(x_0) + \cdots$$

数值微分

■ 基本的工具-泰勒展开

$$f(x) = f(x_0) + (x - x_0)f'(x_0)$$

$$+ \frac{(x - x_0)^2}{2!} f''(x_0)$$

$$+ \cdots + \frac{(x - x_0)^n}{n!} f^{(n)}(x_0) + \cdots$$

■ 单变量函数 $f(x)$ 的一阶导数定义为

$$f'(x_i) = \lim_{\Delta x \rightarrow 0} \frac{f(x_i + \Delta x) - f(x_i)}{\Delta x}$$

数值微分

- 将空间离散成点 x_i , 每两个点之间的间隔为 $x_{i+1} - x_i = h$, 函数在这些离散的点上的值为 $f_i = f(x_i)$.

数值微分

- 将空间离散成点 x_i , 每两个点之间的间隔为 $x_{i+1} - x_i = h$, 函数在这些离散的点上的值为 $f_i = f(x_i)$.
- 可以得到一阶导数最简单的表达形式

$$f'_i = \frac{f_{i+1} - f_i}{h} + O(h)$$

其中 $O(h)$ 表示与 h 相同阶数的项.

数值微分

- 将空间离散成点 x_i , 每两个点之间的间隔为 $x_{i+1} - x_i = h$, 函数在这些离散的点上的值为 $f_i = f(x_i)$.
- 可以得到一阶导数最简单的表达形式

$$f'_i = \frac{f_{i+1} - f_i}{h} + O(h)$$

其中 $O(h)$ 表示与 h 相同阶数的项.

- 这即为一阶导数的两点公式.

数值微分

- 如果我们将 f_{i+1} 和 f_{i-1} 在 x_i 点展开，则可以得到更高精度的公式（三点公式）

$$f_{i+1} - f_{i-1} = 2hf'_i + O(h^3)$$

$$f'_i = \frac{f_{i+1} - f_{i-1}}{2h} + O(h^2)$$

■ 甚至 5 点公式

$$f_{i+1} - f_{i-1} = 2hf'_i + \frac{h^3}{3}f_i^{(3)} + O(h^5)$$

$$f_{i+2} - f_{i-2} = 4hf'_i + \frac{8h^3}{3}f_i^{(3)} + O(h^5)$$

$$f'_i = \frac{1}{12h} (f_{i-2} - 8f_{i-1} + 8f_{i+1} - f_{i+2}) + O(h^4)$$

- 原则上我们可以加上更多点来得到更高精度的公式

- 原则上我们可以加上更多点来得到更高精度的公式
- 然而在许多情况中，这并不实际.

- 原则上我们可以加上更多点来得到更高精度的公式
- 然而在许多情况中，这并不实际.
- 在实际问题中，边界点的导数值更重要，需要精确计算. 导数表达式中涉及的点越多，在边界处获得精确的导数就越困难.

二阶导数

- 同样的对 $f_{i\pm 1}$ 在 x_i 点的展开，可以得到二阶导数的三点公式

$$f_{i+1} - 2f_i + f_{i-1} = h^2 f''_i + O(h^4)$$

$$f''_i = \frac{f_{i+1} - 2f_i + f_{i-1}}{h^2} + O(h^2)$$

二阶导数

- 同样的对 $f_{i\pm 1}$ 在 x_i 点的展开，可以得到二阶导数的三点公式

$$f_{i+1} - 2f_i + f_{i-1} = h^2 f''_i + O(h^4)$$

$$f''_i = \frac{f_{i+1} - 2f_i + f_{i-1}}{h^2} + O(h^2)$$

- 类似的，也可以用更多的点给出更高精度的公式（五点公式）

$$f''_i = \frac{1}{12h^2} (-f_{i-2} + 16f_{i-1} - 30f_i + 16f_{i+1} - f_{i+2}) + O(h^4)$$

二阶导数

- 同样的对 $f_{i\pm 1}$ 在 x_i 点的展开，可以得到二阶导数的三点公式

$$f_{i+1} - 2f_i + f_{i-1} = h^2 f''_i + O(h^4)$$

$$f''_i = \frac{f_{i+1} - 2f_i + f_{i-1}}{h^2} + O(h^2)$$

- 类似的，也可以用更多的点给出更高精度的公式（五点公式）

$$f''_i = \frac{1}{12h^2} (-f_{i-2} + 16f_{i-1} - 30f_i + 16f_{i+1} - f_{i+2}) + O(h^4)$$

- 然而获取在边界处的导数仍然困难。

二阶导数

- 一个办法是通过已有的导数值进行差值得到边界值.

二阶导数

- 一个办法是通过已有的导数值进行差值得到边界的值.
- 练习：对 $\sin x$ 在 $[0, 2\pi]$ 中间取 101 个点，计算在这些点的一阶导数和二阶导数值，并通过拉格朗日插值给出在边界上的导数值.

不等间隔

- 实际中，我们还会碰到两个问题，一个是数据的间隔是不等的.

不等间隔

- 实际中，我们还会碰到两个问题，一个是数据的间隔是不等的.
- 一种解决的办法是利用函数插值，得到等间隔的数据来计算. 这种方法会因函数插值引入另外的误差来源.

不等间隔

- 实际中，我们还会碰到两个问题，一个是数据的间隔是不等的.
- 一种解决的办法是利用函数插值，得到等间隔的数据来计算. 这种方法会因函数插值引入另外的误差来源.
- 另一种办法则是直接利用泰勒展开，给出不等间隔的公式

$$f'_i = \frac{h_{i-1}^2 f_{i+1} + (h_i^2 - h_{i-1}^2) f_i - h_i^2 f_{i-1}}{h_i h_{i-1} (h_i + h_{i-1})} + O(h^2)$$

不等间隔

- 同样的可以给出二阶导数的三点公式

$$f_i'' = \frac{2[h_{i-1}f_{i+1} - (h_i + h_{i-1})f_i + h_if_{i-1}]}{h_i h_{i+1} (h_i + h_{i-1})} + O(h)$$

不等间隔

- 同样的可以给出二阶导数的三点公式

$$f_i'' = \frac{2[h_{i-1}f_{i+1} - (h_i + h_{i-1})f_i + h_if_{i-1}]}{h_i h_{i+1} (h_i + h_{i-1})} + O(h)$$

- 这里精度要比等间隔的低，因为没法正好抵消三阶项。如果需要更高精度，可以采用五点公式。

Richardson extrapolation

- 另一个问题是在三点公式和五点公式中，我们其实没法控制误差。

Richardson extrapolation

- 另一个问题是在三点公式和五点公式中，我们其实没法控制误差。
- 如果我们能得到连续函数在任意点的值，则可以通过自适应方案来获取任意精度的导数值。

Richardson extrapolation

- 另一个问题是在三点公式和五点公式中，我们其实没法控制误差。
- 如果我们能得到连续函数在任意点的值，则可以通过自适应方案来获取任意精度的导数值。
- 该方法的主要思想是通过计算 $\Delta_1(h)$ 和 $\Delta_1(h/2)$ 来抵消泰勒展开的高阶项，而该过程可以不断进行，从而不断改进精度。

$$\Delta_1(h) = \frac{f(x+h) - f(x-h)}{2h}$$

Richardson extrapolation

■ 例如

$$\Delta_1(h) - 4\Delta_1(h/2) = -3f'(x) + O(h^4)$$

Richardson extrapolation

■ 例如

$$\Delta_1(h) - 4\Delta_1(h/2) = -3f'(x) + O(h^4)$$

■ 则 $f'(x)$ 的值可以由 $\Delta_1(h)$ 和 $\Delta_1(h/2)$ 的差值给出，而这个可以不断迭代，直到获得给定的精度。

Richardson extrapolation

- 实际上，我们可以使用更多的 $\Delta_1(h/2^k)$ 来进行迭代。例如，可以验证有

$$\Delta_1(h) - 20\Delta_1(h/2) + 64\Delta_1(h/4) = 45f'(x) + O(h^6)$$

Richardson extrapolation

- 实际上，我们可以使用更多的 $\Delta_1(h/2^k)$ 来进行迭代. 例如，可以验证有

$$\Delta_1(h) - 20\Delta_1(h/2) + 64\Delta_1(h/4) = 45f'(x) + O(h^6)$$

- 从而每次迭代可以获得更快的精度收敛. 该方案 Richardson 外推法.

Richardson extrapolation

- 实际上，我们可以使用更多的 $\Delta_1(h/2^k)$ 来进行迭代。例如，可以验证有

$$\Delta_1(h) - 20\Delta_1(h/2) + 64\Delta_1(h/4) = 45f'(x) + O(h^6)$$

- 从而每次迭代可以获得更快的精度收敛。该方案 Richardson 外推法。
- 二阶导数的 Richardson 外推公式为

$$\Delta_2(h) - 4\Delta_2(h/2) = -3f''(x) + O(h^4)$$

第六节

数值微积分

6.1 数值微分

6.2 数值积分

6.3 方程的根

6.4 函数极值

6.5 经典散射

数值积分

■ 假设我们希望计算积分

$$I = \int_a^b f(x)dx$$

数值积分

- 假设我们希望计算积分

$$I = \int_a^b f(x) dx$$

- 回到我们最初积分的定义，将区间 $[a, b]$ 分为 N 等份，每份长度为 $h = (b - a)/N$.

数值积分

- 假设我们希望计算积分

$$I = \int_a^b f(x) dx$$

- 回到我们最初积分的定义，将区间 $[a, b]$ 分为 N 等份，每份长度为 $h = (b - a)/N$.
- 如果在每个区间 $[x_i, x_{i+1}]$ 用线性函数来近似，则有

$$I \approx h \left[\frac{1}{2}f_0 + f_1 + \dots + f_{N-1} + \frac{1}{2}f_N \right]$$

数值积分

- 假设我们希望计算积分

$$I = \int_a^b f(x) dx$$

- 回到我们最初积分的定义，将区间 $[a, b]$ 分为 N 等份，每份长度为 $h = (b - a)/N$.
- 如果在每个区间 $[x_i, x_{i+1}]$ 用线性函数来近似，则有

$$I \approx h \left[\frac{1}{2}f_0 + f_1 + \dots + f_{N-1} + \frac{1}{2}f_N \right]$$

- 练习，利用该公式计算积分 $\int_0^\pi \cos x dx$

等间距的数值积分公式:Newton-Cotes

- 进一步的做法，考虑上一章讨论的 Lagrange 多项式来近似函数

$$P_N(x) = \sum_{i=0}^N f_i L_i(x)$$

等间距的数值积分公式:Newton-Cotes

- 进一步的做法，考虑上一章讨论的 Lagrange 多项式来近似函数

$$P_N(x) = \sum_{i=0}^N f_i L_i(x)$$

- 则原积分可化为

$$I \simeq \int_a^b P_N(x) dx = \sum_{i=0}^N f_i \int_a^b L_i(x) dx$$

等间距的数值积分公式:Newton-Cotes

■ 令 $x = a + ht$, 同时令

$$L_i(x) \equiv \phi_i(t) = \prod_{k=0, k \neq i}^N \frac{t - k}{i - k}$$

等间距的数值积分公式:Newton-Cotes

■ 令 $x = a + ht$, 同时令

$$L_i(x) \equiv \phi_i(t) = \prod_{k=0, k \neq i}^N \frac{t - k}{i - k}$$

■ 则积分可化为

$$\int_a^b P_N(x) dx = h \sum_{i=0}^N f_i \alpha_i$$

其中权重 α_i 为 $\alpha_i = \int_0^N \phi_i(t) dt$

等间距的数值积分公式:Newton-Cotes

■ 当取 $N = 2$ 时, 我们有

$$a_0 = \int_0^2 \frac{t-1}{0-1} \frac{t-2}{0-2} dt = \frac{1}{3}$$

以及 $\alpha_1 = 4/3, \alpha_2 = 1/3$. 这即是著名的 Simpson 公式

$$\int_a^b P_2(x) dx = \frac{h}{3}(f_0 + 4f_1 + f_2)$$

等间距的数值积分公式:Newton-Cortes

- 对于任意的 N 点公式则称为 Newton-Cortes 公式.

$$\int_a^b P_N(x) dx = \frac{b-a}{N} \sum_{i=0}^N f_i \sigma_i$$

等间距的数值积分公式:Newton-Cortes

- 对于任意的 N 点公式则称为 Newton-Cortes 公式.

$$\int_a^b P_N(x) dx = \frac{b-a}{N} \sum_{i=0}^N f_i \sigma_i$$

- 由多项式内插所造成的误差可以按照下式估计

$$\int_a^b [P_N(x) - f(x)] dx = h^{p+1} \cdot K \cdot f^{(p)}(\xi)$$

等间距的数值积分公式:Newton-Cortes

- 对于任意的 N 点公式则称为 Newton-Cortes 公式.

$$\int_a^b P_N(x) dx = \frac{b-a}{N} \sum_{i=0}^N f_i \sigma_i$$

- 由多项式内插所造成的误差可以按照下式估计

$$\int_a^b [P_N(x) - f(x)] dx = h^{p+1} \cdot K \cdot f^{(p)}(\xi)$$

- 在真实计算数值积分时，我们一般将积分区间分成若干小段，在每段上运用这些积分公式.

等间距的数值积分公式:Newton-Cotes

N	σ_i	N_s	误差估计	名称
1	1,1	2	$\frac{1}{12}h^3f^2(\xi)$	梯形规则
2	1,4,1	6	$\frac{1}{90}h^5f^4(\xi)$	Simpson 规则
3	1,3,3,1	8	$\frac{3}{80}h^5f^4(\xi)$	3/8 规则
4	7,32,12,32,7	90	$\frac{8}{945}h^7f^6(\xi)$	Milne 规则
5	19,75,50,50,75,19	288	$\frac{275}{12096}h^7f^6(\xi)$	
6	41,216,27,272,27,216,41	840	$\frac{9}{1400}h^9f^8(\xi)$	Weddle 规则

表格: 对于 $N = 1, 2, 3, 4, 5, 6$ 的情况

不等间距积分

- Simpson 积分规则可以很容易的推广到不等间距的数据点.

不等间距积分

- Simpson 积分规则可以很容易的推广到不等间距的数据点.
- 区间上差值函数写为 $f(x) = a^2 + bx + c$

$$a = \frac{h_{i-1}f_{i+1} - (h_{i-1} + h_i)f_i + h_if_{i-1}}{h_{i-1}h_i(h_{i-1} + h_i)}$$

$$b = \frac{h_{i-1}^2f_{i+1} + (h_i^2 - h_{i-1}^2)f_i - h_i^2f_{i-1}}{h_{i-1}h_i(h_{i-1} + h_i)}$$

$$c = f_i$$

不等间距积分

- 对于积分 $S_i = \int_{x_{i-1}}^{x_{i+1}} f(x)dx$ 可以写为

$$S_i = \int_{-h_{i-1}}^{h_i} f(x)dx = \alpha f_{i+1} + \beta f_i + \gamma f_{i-1}$$

$$\alpha = \frac{2h_i^2 + h_i h_{i-1} - h_{i-1}^2}{6h_i}$$

$$\beta = \frac{(h_i + h_{i-1})^3}{6h_i h_{i-1}}$$

$$\gamma = \frac{-h_i^2 + h_i h_{i-1} + 2h_{i-1}^2}{6h_i}$$

自适应积分

- 虽然我们可以给出 Simpson 或梯形积分的误差量级，但由于无法控制前面的系数，故无法完全控制误差。为了控制误差，我们可以采用自适应的积分方法。

自适应积分

- 虽然我们可以给出 Simpson 或梯形积分的误差量级，但由于无法控制前面的系数，故无法完全控制误差。为了控制误差，我们可以采用自适应的积分方法。
- 如果我们将积分在 $x = a$ 处做泰勒展开则有

$$\begin{aligned} S &= \int_a^b f(x) dx \\ &= \int_a^b \left[\sum_{k=0}^{\infty} \frac{(x-a)^k}{k!} f^{(k)}(a) \right] dx \\ &= \sum_{k=0}^{\infty} \frac{h^{k+1}}{(k+1)!} f^{(k)}(a) \end{aligned}$$

自适应积分

- 接下来,我们在取 $x_{i-1} = a, x_i = c = (a+b)/2, x_{i+1} = b$, 将 $f(c), f(b)$ 在 $x = a$ 处展开, 我们可以得到零阶近似

$$S_0 = \frac{h}{6}[f(a) + 4f(c) + f(b)]$$

$$= \frac{h}{6} \left[f(a) + \sum_{k=0}^{\infty} \frac{h^k}{k!} \left(\frac{1}{2^{k-2}} - 1 \right) f^{(k)}(a) \right]$$

自适应积分

- 接下来,我们在取 $x_{i-1} = a, x_i = c = (a+b)/2, x_{i+1} = b$, 将 $f(c), f(b)$ 在 $x = a$ 处展开, 我们可以得到零阶近似

$$S_0 = \frac{h}{6}[f(a) + 4f(c) + f(b)]$$

$$= \frac{h}{6} \left[f(a) + \sum_{k=0}^{\infty} \frac{h^k}{k!} \left(\frac{1}{2^{k-2}} - 1 \right) f^{(k)}(a) \right]$$

- 此时 S 和 S_0 的领头阶之差为

$$\Delta S_0 = S - S_0 \simeq -\frac{h^5}{2880} f^{(4)}(a)$$

自适应积分

- 进一步在 $[a, c]$ 和 $[c, b]$ 之间应用 Simpson 法则得到一阶近似为

$$S_1 = \frac{h}{12} [f(a) + 4f(d) + 2f(c) + 4f(e) + f(b)]$$

自适应积分

- 进一步在 $[a, c]$ 和 $[c, b]$ 之间应用 Simpson 法则得到一阶近似为

$$S_1 = \frac{h}{12} [f(a) + 4f(d) + 2f(c) + 4f(e) + f(b)]$$

- 此时的差为

$$\Delta S_1 = S - S_1$$

$$\approx -\frac{(h/2)^5}{2880} f^{(4)}(a) - \frac{(h/2)^5}{2880} f^{(4)}(c)$$

$$\approx -\frac{1}{2^4} \frac{h^5}{2880} f^{(4)}(a)$$

自适应积分

- 可以发现一阶近似和零阶近似的差别为

$$S_1 - S_0 \approx -\frac{15}{16} \frac{h^5}{2880} f^{(4)}(a) \approx \frac{15}{16} \Delta S_0 \approx 15 \Delta S_1$$

自适应积分

- 可以发现一阶近似和零阶近似的差别为

$$S_1 - S_0 \approx -\frac{15}{16} \frac{h^5}{2880} f^{(4)}(a) \approx \frac{15}{16} \Delta S_0 \approx 15 \Delta S_1$$

- 因此我们可以通过这种方法不断迭代来改进误差. 即不断将区域进行二分划分来积分, 直到获得给定的精度 $|S_{n-1} - S_n| \leq 15\delta$.

自适应积分

- 可以发现一阶近似和零阶近似的差别为

$$S_1 - S_0 \approx -\frac{15}{16} \frac{h^5}{2880} f^{(4)}(a) \approx \frac{15}{16} \Delta S_0 \approx 15 \Delta S_1$$

- 因此我们可以通过这种方法不断迭代来改进误差. 即不断将区域进行二分划分来积分, 直到获得给定的精度 $|S_{n-1} - S_n| \leq 15\delta$.
- 自适应的方法需要被积函数在积分区域连续.

自适应积分

例 1 编写程序，采用自适应的 Simpson 积分来计算积分

$$S = \int_0^{\pi} \frac{[1 + a_0(1 - \cos x)^2] dx}{(1 + a_0 \sin^2 x) \sqrt{1 + 2a_0(1 - \cos x)}}$$

其中取 $a_0 \geq 0$.

第六节

数值微积分

6.1 数值微分

6.2 数值积分

6.3 方程的根

6.4 函数极值

6.5 经典散射

二分法

- 如果知道 $f(x) = 0$ 的一个根在区间 $[a, b]$, 可以采用二分法得到给定的精度.

二分法

- 如果知道 $f(x) = 0$ 的一个根在区间 $[a, b]$, 可以采用二分法得到给定的精度.
- 二分法是一个非常直观又简便的方法.

二分法

- 如果知道 $f(x) = 0$ 的一个根在区间 $[a, b]$, 可以采用二分法得到给定的精度.
- 二分法是一个非常直观又简便的方法.
- 由于一个根在此区间, 故 $f(a)f(b) < 0$. 将区间减半 $x_0 = (a + b)/2$. 如果 $f(a)f(x_0) < 0$ 则, 解在 $[a, x_0]$, 否则在区间 $[x_0, b]$ 继续寻找. 直到区间的长度小于要求的精度, 则停止迭代.

二分法

- 如果知道 $f(x) = 0$ 的一个根在区间 $[a, b]$, 可以采用二分法得到给定的精度.
- 二分法是一个非常直观又简便的方法.
- 由于一个根在此区间, 故 $f(a)f(b) < 0$. 将区间减半 $x_0 = (a + b)/2$. 如果 $f(a)f(x_0) < 0$ 则, 解在 $[a, x_0]$, 否则在区间 $[x_0, b]$ 继续寻找. 直到区间的长度小于要求的精度, 则停止迭代.
- 试计算 $f(x) = e^x \ln x - x^2$ 在区间 $[1, 2]$ 之间的根.

牛顿法

- 牛顿法通过对一个光滑函数在根附近可以用线性函数近似来进行求解.

牛顿法

- 牛顿法通过对一个光滑函数在根附近可以用线性函数近似来进行求解.
- 假设 $f(x_r) = 0$, 则函数在 x_r 附近有

$$f(x_r) \simeq f(x) + (x_r - x)f'(x) + \dots = 0$$

牛顿法

- 如果 x_k 是 x_r 第 k 步的近似值，则下一步的近似值根据

$$f(x_{k+1}) = f(x_k) + (x_{k+1} - x_k)f'(x_k) \simeq 0$$

牛顿法

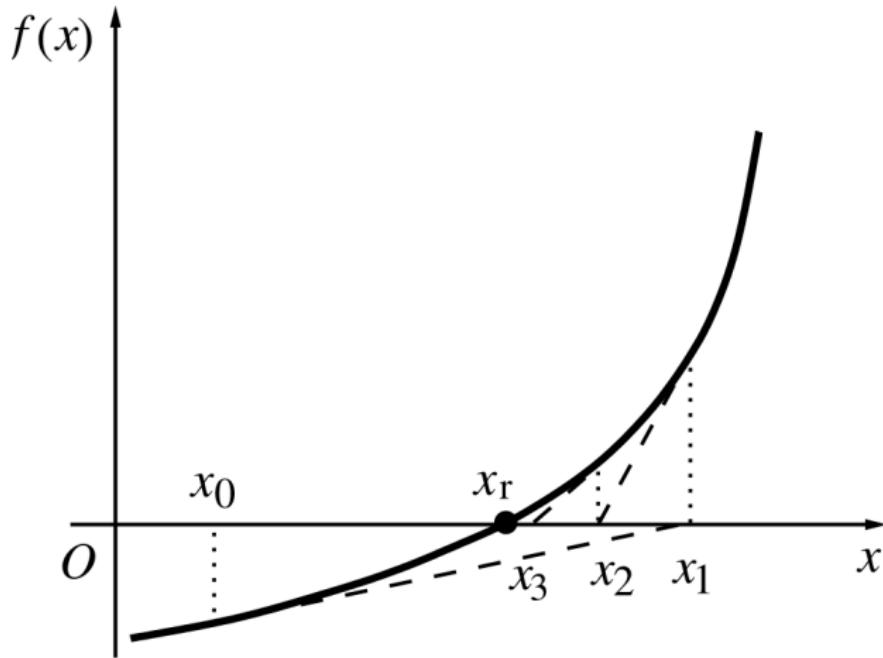
- 如果 x_k 是 x_r 第 k 步的近似值，则下一步的近似值根据

$$f(x_{k+1}) = f(x_k) + (x_{k+1} - x_k)f'(x_k) \approx 0$$

- 故新的 x_{k+1} 应该取

$$x_{k+1} = x_k - f_k/f'_k$$

牛顿法



割线法

- 许多情况下，并不能得到函数的一阶导数的表达式，使得难以使用牛顿法.

割线法

- 许多情况下，并不能得到函数的一阶导数的表达式，使得难以使用牛顿法.
- 可以选择另一个类似的方法，即用两点公式来计算函数的一阶导数

$$x_{k+1} = x_k - (x_k - x_{k-1})f_k / (f_k - f_{k-1})$$

割线法

- 许多情况下，并不能得到函数的一阶导数的表达式，使得难以使用牛顿法.
- 可以选择另一个类似的方法，即用两点公式来计算函数的一阶导数

$$x_{k+1} = x_k - (x_k - x_{k-1})f_k / (f_k - f_{k-1})$$

- 该方法通常被称为割线法或离散的牛顿法.

割线法

- 许多情况下，并不能得到函数的一阶导数的表达式，使得难以使用牛顿法.
- 可以选择另一个类似的方法，即用两点公式来计算函数的一阶导数

$$x_{k+1} = x_k - (x_k - x_{k-1})f_k / (f_k - f_{k-1})$$

- 该方法通常被称为割线法或离散的牛顿法.
- 优点是不再需要 $f(x)$ 的导数信息，缺点是必须使用两点来作为起始. 两点法比二分法更有效而低于牛顿法.

第六节

数值微积分

6.1 数值微分

6.2 数值积分

6.3 方程的根

6.4 函数极值

6.5 经典散射

函数极值

- 与函数求根类似的问题则是求函数的极值.

函数极值

- 与函数求根类似的问题则是求函数的极值.
- 我们知道当一个函数取极值，则有

$$f(x) = \frac{dg(x)}{dx} = 0$$

如果 $f'(x) > 0$ 则为极小值点，如果 $f'(x) < 0$ 则为极大值点. 之前给出的求根方法可以推广到这里.

函数极值

- 不过对于函数求极值时，需要注意，我们需要判断当求函数的极大值时， $g(x_{k+1})$ 是否是在增加。

函数极值

- 不过对于函数求极值时，需要注意，我们需要判断当求函数的极大值时， $g(x_{k+1})$ 是否是在增加。
- 如果是在增加，我们就接受这个改变。如果不是，我们需要拒绝这次改变。

函数极值

- 不过对于函数求极值时，需要注意，我们需要判断当求函数的极大值时， $g(x_{k+1})$ 是否是在增加.
- 如果是在增加，我们就接受这个改变. 如果不是，我们需要拒绝这次改变.
- 并且将迭代关系更改为 $x_{k+1} = x_k - \Delta x_k$.

函数极值

- 不过对于函数求极值时，需要注意，我们需要判断当求函数的极大值时， $g(x_{k+1})$ 是否是在增加.
- 如果是在增加，我们就接受这个改变. 如果不是，我们需要拒绝这次改变.
- 并且将迭代关系更改为 $x_{k+1} = x_k - \Delta x_k$.
- 即如果采用牛顿法，取 $\Delta x_k = -f_k/f'_k$ ，如果是割线法，则 $\Delta x_k = -(x_k - x_{k-1})f_k/(f_k - f_{k-1})$.

离子键长

- 我们通过计算 NaCl 的键长为例. 设 NaCl 中两个离子之间的势为

$$V(r) = -\frac{e^2}{4\pi\epsilon_0 r} + V_0 e^{-r/r_0}$$

离子键长

- 我们通过计算 NaCl 的键长为例. 设 NaCl 中两个离子之间的势为

$$V(r) = -\frac{e^2}{4\pi\epsilon_0 r} + V_0 e^{-r/r_0}$$

- 则它们之间的键长 r_{eq} 既为当 $V(r)$ 取极小值时的距离 r . 取 $V_0 = 1.09 \times 10^3 \text{ eV}$ 及 $r_0 = 0.330 \text{ \AA}$.

离子键长

- 在平衡位置时，两个离子之间的力为

$$f(r) = -\frac{dV(r)}{dr} = -\frac{e^2}{4\pi\epsilon_0 r^2} + \frac{V_0}{r_0} e^{-r/r_0} = 0$$

离子键长

- 在平衡位置时，两个离子之间的力为

$$f(r) = -\frac{dV(r)}{dr} = -\frac{e^2}{4\pi\epsilon_0 r^2} + \frac{V_0}{r_0} e^{-r/r_0} = 0$$

- 即需要寻找 $f(x) = dg(x)/dx = 0$, 其中 $g(x) = -V(x)$.

离子键长

- 在平衡位置时，两个离子之间的力为

$$f(r) = -\frac{dV(r)}{dr} = -\frac{e^2}{4\pi\epsilon_0 r^2} + \frac{V_0}{r_0} e^{-r/r_0} = 0$$

- 即需要寻找 $f(x) = dg(x)/dx = 0$, 其中 $g(x) = -V(x)$.
- 练习：写出极小值的程序，并计算其键长.

第六节

数值微积分

6.1 数值微分

6.2 数值积分

6.3 方程的根

6.4 函数极值

6.5 经典散射

经典散射

- 物理学中，散射是一个非常重要和普遍的物理过程.

经典散射

- 物理学中，散射是一个非常重要和普遍的物理过程.
- 从微观的核子中的质子中子到宏观的星系星球，散射过程是我们了解他们的结构及相互作用的重要途径.

经典散射

- 物理学中，散射是一个非常重要和普遍的物理过程.
- 从微观的核子中的质子中子到宏观的星系星球，散射过程是我们了解他们的结构及相互作用的重要途径.
- 而且对于多体过程，如果没有相干散射，可以将他们处理为一系列的两体散射过程.

两体系统

- 通常两体系统的拉格朗日量可以写为

$$\mathcal{L} = \frac{m_1}{2}v_1^2 + \frac{m_2}{2}v_2^2 - V(\vec{r}_1, \vec{r}_2)$$

两体系统

- 通常两体系统的拉格朗日量可以写为

$$\mathcal{L} = \frac{m_1}{2}v_1^2 + \frac{m_2}{2}v_2^2 - V(\vec{r}_1, \vec{r}_2)$$

- 假设相互作用具有球对称性, 即 $V(\vec{r}_1, \vec{r}_2) = V(r_{21})$

两体系统

- 通常在质心系中处理两体问题更方便，即

$$\vec{r} = \vec{r}_2 - \vec{r}_1$$

$$\vec{r}_c = \frac{m_1 \vec{r}_1 + m_2 \vec{r}_2}{m_1 + m_2}$$

两体系统

- 通常在质心系中处理两体问题更方便，即

$$\vec{r} = \vec{r}_2 - \vec{r}_1$$

$$\vec{r}_c = \frac{m_1 \vec{r}_1 + m_2 \vec{r}_2}{m_1 + m_2}$$

- 该坐标系下拉格朗日量可写为

$$\mathcal{L} = \frac{M}{2} v_c^2 + \frac{m}{2} v^2 - V(r)$$

两体系统

■ 其中

$$M = m_1 + m_2, m = \frac{m_1 m_2}{m_1 + m_2}$$

两体系统

■ 其中

$$M = m_1 + m_2, m = \frac{m_1 m_2}{m_1 + m_2}$$

■ 在质心系下 $v_c = d\vec{r}_c/dt = 0$, 则两体问题可以转化为单粒子在中心势场的问题

两体系统

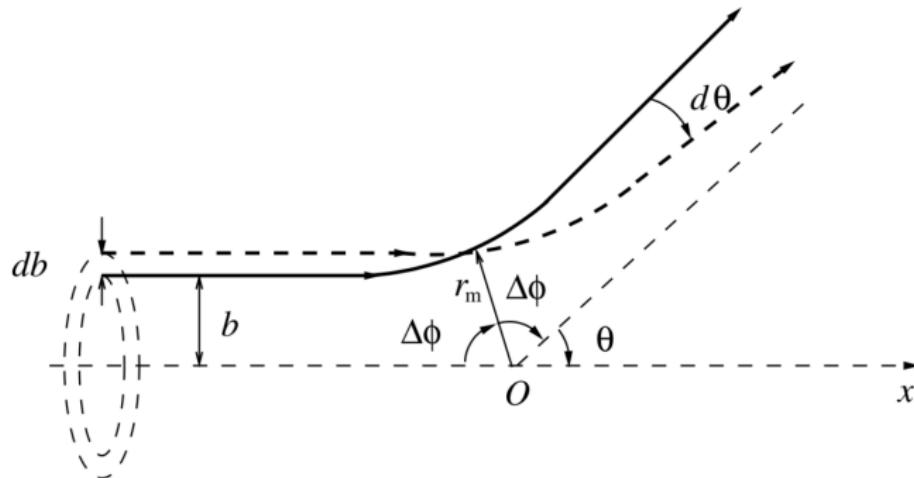
■ 其中

$$M = m_1 + m_2, m = \frac{m_1 m_2}{m_1 + m_2}$$

- 在质心系下 $v_c = d\vec{r}_c/dt = 0$, 则两体问题可以转化为单粒子在中心势场的问题
- 利用牛顿定律也可以推导出同样的结论.

散射截面

■ 研究质量为 m 的粒子在中心势下面的散射过程



散射截面

- 该过程的总散射截面为

$$\sigma = \int \sigma(\theta) d\Omega$$

其中 $\sigma(\theta)$ 为微分散射截面，即在立体角 $d\Omega = 2\pi \sin \theta d\theta$ 中发现粒子的概率。

散射截面

- 该过程的总散射截面为

$$\sigma = \int \sigma(\theta) d\Omega$$

其中 $\sigma(\theta)$ 为微分散射截面，即在立体角 $d\Omega = 2\pi \sin \theta d\theta$ 中发现粒子的概率。

- 设粒子的流密度为 I (每单位散射截面面积每单位时间上的粒子数目)，则每单位时间在碰撞参数 b 的 db 区域中的粒子数为 $2\pi I b db$.

散射截面

- 在这些区域的入射粒子将会以概率 $\sigma(\theta)$ 从立体角 $d\Omega$ 出射，故有

$$2\pi I b db = I \sigma(\theta) d\Omega$$

散射截面

- 在这些区域的入射粒子将会以概率 $\sigma(\theta)$ 从立体角 $d\Omega$ 出射，故有

$$2\pi I b db = I \sigma(\theta) d\Omega$$

- 从而可得微分散射截面为

$$\sigma(\theta) = \frac{b}{\sin \theta} \left| \frac{db}{d\theta} \right| \quad (1)$$

散射截面的数值计算

- 在球对称势下，散射过程中能量和角动量守恒，形式上，我们有

$$l = mbv_0 = mr^2\dot{\phi}$$

散射截面的数值计算

- 在球对称势下，散射过程中能量和角动量守恒，形式上，我们有

$$l = mbv_0 = mr^2\dot{\phi}$$

- 以及

$$E = \frac{m}{2}v_0^2 = \frac{m}{2}(\dot{r}^2 + r^2\dot{\phi}^2) + V(r)$$

散射截面的数值计算

- 在球对称势下，散射过程中能量和角动量守恒，形式上，我们有

$$l = mbv_0 = mr^2\dot{\phi}$$

- 以及

$$E = \frac{m}{2}v_0^2 = \frac{m}{2}(\dot{r}^2 + r^2\dot{\phi}^2) + V(r)$$

- 结合上两式，并利用 $\frac{d\phi}{dr} = \frac{d\phi}{dt}\frac{dt}{dr}$ 有

$$\frac{d\phi}{dr} = \pm \frac{b}{r^2 \sqrt{1 - b^2/r^2 - V(r)/E}}$$

散射截面的数值计算

■ 偏转角 θ 有关系式

$$\theta = \pi - 2\Delta\phi$$

散射截面的数值计算

■ 偏转角 θ 有关系式

$$\theta = \pi - 2\Delta\phi$$

■ 从前式有

$$\begin{aligned}\Delta\phi &= b \int_{r_m}^{\infty} \frac{dr}{r^2 \sqrt{1 - b^2/r^2 - V(r)/E}} \\ &= -b \int_{\infty}^{r_m} \frac{dr}{r^2 \sqrt{1 - b^2/r^2 - V(r)/E}}\end{aligned}$$

其中 r_m 为入射粒子离势心的最小距离.

散射截面的数值计算

■ 根据能量守恒和角动量守恒有

$$1 - \frac{b^2}{r_m^2} - \frac{V(r_m)}{E} = 0$$

对应于 r 方向上的速度为零 $\dot{r} = 0$.

散射截面的数值计算

- 根据能量守恒和角动量守恒有

$$1 - \frac{b^2}{r_m^2} - \frac{V(r_m)}{E} = 0$$

对应于 r 方向上的速度为零 $\dot{r} = 0$.

- 因为当 $\Delta\phi = \pi/2$ 时对应于 $V(r) = 0$, 可以将 θ 表示为

$$\theta = 2b \left[\int_b^\infty \frac{dr}{r^2 \sqrt{1 - b^2/r^2}} - \int_{r_m}^\infty \frac{dr}{r^2 \sqrt{1 - b^2/r^2 - V(r)/E}} \right]$$

散射截面的数值计算

■ 假设给定的势为 Yukawa 势

$$V(r) = \frac{K}{r} e^{-r/a}$$

散射截面的数值计算

- 假设给定的势为 Yukawa 势

$$V(r) = \frac{K}{r} e^{-r/a}$$

- 我们可以用割线法在给定 b 和 E 的情况下计算 r_m .

散射截面的数值计算

- 假设给定的势为 Yukawa 势

$$V(r) = \frac{K}{r} e^{-r/a}$$

- 我们可以用割线法在给定 b 和 E 的情况下计算 r_m .
- 然后利用 Simpson 算法去计算积分.

散射截面的数值计算

- 假设给定的势为 Yukawa 势

$$V(r) = \frac{K}{r} e^{-r/a}$$

- 我们可以用割线法在给定 b 和 E 的情况下计算 r_m .
- 然后利用 Simpson 算法去计算积分.
- 之后利用一阶导数的三点公式得到 $\frac{d\theta}{db}$

散射截面的数值计算

- 假设给定的势为 Yukawa 势

$$V(r) = \frac{K}{r} e^{-r/a}$$

- 我们可以用割线法在给定 b 和 E 的情况下计算 r_m .
- 然后利用 Simpson 算法去计算积分.
- 之后利用一阶导数的三点公式得到 $\frac{d\theta}{db}$
- 最后根据公式1，给出微分散射截面.

散射截面的数值计算

例2 取 $E = m = \kappa = 1$, 计算 Yukawa 势下的微分散射截面, 当 $a \rightarrow \infty$ 和库伦散射截面对比 (卢瑟福公式)

$$\sigma(\theta) = \left(\frac{\kappa}{4E} \right)^2 \frac{1}{\sin^4(\theta/2)}$$

第四节

Mathematica 基础

第五节

函数近似

第六节

数值微积分

第七节

常微分方程

第八节

矩阵的数值方法

常微分方程

- 大部分出现在物理和工程中的问题会以微分方程的方式出现.

常微分方程

- 大部分出现在物理和工程中的问题会以微分方程的方式出现.
- 通常我们呢可以将常微分方程分为三个类别

常微分方程

- 大部分出现在物理和工程中的问题会以微分方程的方式出现.
- 通常我们呢可以将常微分方程分为三个类别
 - 初值问题，主要是含时演化的问题

常微分方程

- 大部分出现在物理和工程中的问题会以微分方程的方式出现.
- 通常我们呢可以将常微分方程分为三个类别
 - 初值问题，主要是含时演化的问题
 - 边值问题，微分方程包含特别的边界

常微分方程

- 大部分出现在物理和工程中的问题会以微分方程的方式出现.
- 通常我们呢可以将常微分方程分为三个类别
 - 初值问题，主要是含时演化的问题
 - 边值问题，微分方程包含特别的边界
 - 本征值问题，求解方程中的给定参数

7.1 初值问题

7.2 欧拉和皮卡德方法

7.3 预修正方法

7.4 Runge-Kutta 方法

7.5 驱动摆的混沌运动

7.6 打靶法

7.7 线性方程与 Sturm-Liouville 问题

7.8 一维薛定谔方程

初值问题

- 初值问题通常涉及一些运动学系统. 如星体的运动, 火箭的运动等.

初值问题

- 初值问题通常涉及一些运动学系统. 如星体的运动, 火箭的运动等.
- 一个运动学系统的行为可以由一组一阶微分方程来描述

$$\frac{dy}{dt} = g(y, t)$$

初值问题

- 初值问题通常涉及一些运动学系统. 如星体的运动, 火箭的运动等.
- 一个运动学系统的行为可以由一组一阶微分方程来描述

$$\frac{dy}{dt} = g(y, t)$$

- 其中 $y = (y_1, \dots, y_l)$, $g(y, t) = [g_1(y, t), \dots, g_l(y, t)]$
| 为运动学变量的个数. 原则上, 给定了初始条件,
我们可以得到方程的解.

初值问题

- 例如一个粒子在弹力的作用下的运动由牛顿方程给出 $f = ma$

初值问题

- 例如一个粒子在弹力的作用下的运动由牛顿方程给出 $f = ma$
- 该方程即可视为上面为 $l = 2$ 的例子, 即 $y_1 = x, y_2 = v = dx/dt$, 而 $g_1 = v = y_2, g_2 = f/m = -kx/m = -ky_1/m$. 则牛顿方程改为上面的形式为

$$\frac{dy_1}{dt} = y_2; \quad \frac{dy_2}{dt} = -\frac{k}{m}y_1$$

初值问题

- 例如一个粒子在弹力的作用下的运动由牛顿方程给出 $f = ma$
- 该方程即可视为上面为 $l = 2$ 的例子, 即 $y_1 = x, y_2 = v = dx/dt$, 而 $g_1 = v = y_2, g_2 = f/m = -kx/m = -ky_1/m$. 则牛顿方程改为上面的形式为

$$\frac{dy_1}{dt} = y_2; \quad \frac{dy_2}{dt} = -\frac{k}{m}y_1$$

- 事实上, 许多高阶微分方程都可以转化为一组耦合的一阶微分方程.

7.1 初值问题

7.2 欧拉和皮卡德方法

7.3 预修正方法

7.4 Runge-Kutta 方法

7.5 驱动摆的混沌运动

7.6 打靶法

7.7 线性方程与 Sturm-Liouville 问题

7.8 一维薛定谔方程

初值问题

- 我们首先以一维的情况为例来说明求解这些一阶方程组的数值方法.

初值问题

- 我们首先以一维的情况为例来说明求解这些一阶方程组的数值方法.
- 利用上一章给出的微分的两点公式，我们可以将原方程化为

$$y_{i+1} = y_i + \tau g_i + O(\tau^2)$$

其中 y_{i+1}, y_i 为时间片 t_{i+1}, t_i 的位置, $\tau = t_{i+1} - t_i$.
该算法既为著名的欧拉算法.

初值问题

- 我们首先以一维的情况为例来说明求解这些一阶方程组的数值方法.
- 利用上一章给出的微分的两点公式, 我们可以将原方程化为

$$y_{i+1} = y_i + \tau g_i + O(\tau^2)$$

其中 y_{i+1}, y_i 为时间片 t_{i+1}, t_i 的位置, $\tau = t_{i+1} - t_i$.
该算法既为著名的欧拉算法.

- 练习: 试用欧拉法求解谐振子系统.

初值问题

- 我们首先以一维的情况为例来说明求解这些一阶方程组的数值方法.
- 利用上一章给出的微分的两点公式，我们可以将原方程化为

$$y_{i+1} = y_i + \tau g_i + O(\tau^2)$$

其中 y_{i+1}, y_i 为时间片 t_{i+1}, t_i 的位置, $\tau = t_{i+1} - t_i$.
该算法既为著名的欧拉算法.

- 练习：试用欧拉法求解谐振子系统.
- 该算法的精度较差，并不适合在实际的数值计算中使用.

初值问题

■ 我们将微分方程转化为积分方程

$$y_{i+j} = y_i + \int_{t_i}^{t_{i+j}} g(y, t) dt$$

初值问题

- 我们将微分方程转化为积分方程

$$y_{i+j} = y_i + \int_{t_i}^{t_{i+j}} g(y, t) dt$$

- 如果能够给出这个积分方程，则得到完全解。然而通常没法将积分解析积出，而我们可以进行函数近似。

初值问题

- 我们将微分方程转化为积分方程

$$y_{i+j} = y_i + \int_{t_i}^{t_{i+j}} g(y, t) dt$$

- 如果能够给出这个积分方程，则得到完全解。然而通常没法将积分解析积出，而我们可以进行函数近似。
- 该积分的精度则决定了解的精度。如果取 $j = 1$ ，及近似 $g(y, t) \approx g_i$ 则回到了欧拉算法。

初值问题

- 皮卡德方法则是一种自适应方法. 每次迭代通过使用前一次迭代的解作为方程右边的输入.

初值问题

- 皮卡德方法则是一种自适应方法. 每次迭代通过使用前一次迭代的解作为方程右边的输入.
- 实际中, 我们需要用数值求积的方法来处理右侧的积分. 例如, 如果我们取 $j = 1$ 并采用梯形法求积, 我们得到算法

$$y_{i+1} = y_i + \frac{\tau}{2} (g_i + g_{i+1}) + O(\tau^3)$$

初值问题

- 注意到 $g_{i+1} = g(y_{i+1}, t_{i+1})$ 包含了 y_{i+1} , 这可以利用皮卡德法自适应给出.

初值问题

- 注意到 $g_{i+1} = g(y_{i+1}, t_{i+1})$ 包含了 y_{i+1} , 这可以利用皮卡德法自适应给出.
- 例如, 我们先猜想初始时 $y_{i+1}^{(0)} = y_i$. 从而可以计算出 $y_{i+1}^{(k+1)} = y_i + \frac{\tau}{2}(g_i + g_{i+1}^{(k)})$.

初值问题

- 注意到 $g_{i+1} = g(y_{i+1}, t_{i+1})$ 包含了 y_{i+1} , 这可以利用皮卡德法自适应给出.
- 例如, 我们先猜想初始时 $y_{i+1}^{(0)} = y_i$. 从而可以计算出 $y_{i+1}^{(k+1)} = y_i + \frac{\tau}{2}(g_i + g_{i+1}^{(k)})$.
- 如果初始的猜测解离真实解较远, 可能会不收敛.

7.1 初值问题

7.2 欧拉和皮卡德方法

7.3 预修正方法

7.4 Runge-Kutta 方法

7.5 驱动摆的混沌运动

7.6 打靶法

7.7 线性方程与 Sturm-Liouville 问题

7.8 一维薛定谔方程

预修正方法

- 预修正方法可以有效的减少迭代次数.

预修正方法

- 预修正方法可以有效的减少迭代次数.
- 首先通过一个较低精度的算法来预测一个 y_{i+1} 的值, 例如欧拉法

预修正方法

- 预修正方法可以有效的减少迭代次数.
- 首先通过一个较低精度的算法来预测一个 y_{i+1} 的值, 例如欧拉法
- 然后利用更好的算法改进给出新的值, 如采用上面例子的皮卡德方法.

预修正方法

例 1 试利用预修正算法计算谐振子的解.

预修正方法

- 另一种改进的算法则是通过增加积分包含的数据点. 则可以采用高精度的积分算法.

预修正方法

- 另一种改进的算法则是通过增加积分包含的数据点. 则可以采用高精度的积分算法.
- 例如, 取 $j = 2$, 并采用线性插值来近似 $g(y, t)$, 则有

$$g(y, t) = \frac{(t - t_i)}{\tau} g_{i+1} - \frac{(t - t_{i+1})}{\tau} g_i + O(\tau^2)$$

预修正方法

- 另一种改进的算法则是通过增加积分包含的数据点. 则可以采用高精度的积分算法.
- 例如, 取 $j = 2$, 并采用线性插值来近似 $g(y, t)$, 则有

$$g(y, t) = \frac{(t - t_i)}{\tau} g_{i+1} - \frac{(t - t_{i+1})}{\tau} g_i + O(\tau^2)$$

- 对插值的 $g(y, t)$ 进行积分后, 可得新的算法

$$y_{i+2} = y_i + 2\tau g_{i+1} + O(\tau^3) \quad (2)$$

预修正方法

- 该算法比欧拉法的精度更高，但是需要有两个起始点来实现该算法。

预修正方法

- 该算法比欧拉法的精度更高，但是需要有两个起始点来实现该算法。
- 另一个点可以通过对初始点做泰勒展开得到

$$y_1 = y_0 + \tau g_0 + \frac{\tau^2}{2} \left(\frac{\partial g_0}{\partial t} + g_0 \frac{\partial g_0}{\partial y} \right) + O(\tau^3)$$

$$g_1 = g(y_1, \tau)$$

预修正方法

- 该算法比欧拉法的精度更高，但是需要有两个起始点来实现该算法。
- 另一个点可以通过对初始点做泰勒展开得到

$$y_1 = y_0 + \tau g_0 + \frac{\tau^2}{2} \left(\frac{\partial g_0}{\partial t} + g_0 \frac{\partial g_0}{\partial y} \right) + O(\tau^3)$$

$$g_1 = g(y_1, \tau)$$

- 看起来增加积分点得到更高精度的算法，但是需要更多的起始点。而不精确起始点带来的误差最终抵消了算法的精度。

预修正方法

- 有了更多的点，我们可以采用更高精度的积分公式。例如采用 Simpson 公式，则方程变为

$$y_{i+2} = y_i + \frac{\tau}{3} (g_{i+2} + 4g_{i+1} + g_i) + O(\tau^5)$$

预修正方法

- 有了更多的点，我们可以采用更高精度的积分公式。例如采用 Simpson 公式，则方程变为

$$y_{i+2} = y_i + \frac{\tau}{3} (g_{i+2} + 4g_{i+1} + g_i) + O(\tau^5)$$

- 该方程可以作为方程 (2) 的修正。

预修正方法

例2 设一个摩托冲过一个峡谷. 空气阻力为 $f_r = -\kappa|v|\vec{v} = -cA\rho|v|\vec{v}$, 设摩托车和车手总重量为 250kg 初速度为 $|v| = 67\text{m/s}$, 横截面积 $A = 0.93\text{m}^2$, 空气密度 $\rho = 1.2\text{kg/m}^3$, 取 $c = 1$, 试给出初速度水平角分别为 $40^\circ, 42.5^\circ, 45^\circ$ 时的运动轨迹.

7.1 初值问题

7.2 欧拉和皮卡德方法

7.3 预修正方法

7.4 Runge-Kutta 方法

7.5 驱动摆的混沌运动

7.6 打靶法

7.7 线性方程与 Sturm-Liouville 问题

7.8 一维薛定谔方程

Runge-Kutta 方法

- 之前改进的方案中都需要更多的起始点，这在许多实际问题中并不一定可以实现.

Runge-Kutta 方法

- 之前改进的方案中都需要更多的起始点，这在许多实际问题中并不一定可以实现.
- 例如在初值问题中，只有一个起始是已知的. 只需要一个起始点并且改进的方法是 Runge-kutta 方法.

Runge-Kutta 方法

- 形式上，我们可以将解 $y(t + \tau)$ 在 t 附近做泰勒展开

$$y(t + \tau) = y + \tau y' + \frac{\tau^2}{2} y'' + \frac{\tau^3}{3!} y^{(3)} + \dots$$

$$= y + \tau g + \frac{\tau^2}{2} (g_t + gg_y)$$

$$+ \frac{\tau^3}{6} (g_{tt} + 2gg_{ty} + g^2 g_{yy} + gg_y^2 + g_t g_y) + \dots$$

Runge-Kutta 方法

■ 同样的，可以形式上将解写成

$$y(t + \tau) = y(t) + \alpha_1 c_1 + \alpha_2 c_2 + \dots + \alpha_m c_m$$

Runge-Kutta 方法

- 同样的，可以形式上将解写成

$$y(t + \tau) = y(t) + \alpha_1 c_1 + \alpha_2 c_2 + \dots + \alpha_m c_m$$

- 其中

$$c_1 = \tau g(y, t)$$

$$c_2 = \tau g(y + \nu_{21} c_1, t + \nu_{21} \tau)$$

$$c_3 = \tau g(y + \nu_{31} c_1 + \nu_{32} c_2, t + \nu_{31} \tau + \nu_{32} \tau)$$

$$c_m = \tau g\left(y + \sum_{i=1}^{m-1} \nu_{mi} c_i, t + \tau \sum_{i=1}^{m-1} \nu_{mi}\right)$$

Runge-Kutta 方法

- 将上式同样展开成 τ 的级数，并对比两式，给出关于 α_i 和 v_{ij} 的方程。

Runge-Kutta 方法

- 将上式同样展开成 τ 的级数，并对比两式，给出关于 α_i 和 v_{ij} 的方程。
- 在截断到 $O(\tau^m)$ ，我们可以得到 m 个方程，但是有 $m + m(m - 1)/2$ 个系数需要确定。因此参数有一定的选择性。

Runge-Kutta 方法

■ 例如，取 $m = 2$ ，则 $y(t + \tau)$ 展开为

$$y(t + \tau) = y + \tau g + \frac{\tau^2}{2} (g_t + gg_y)$$

Runge-Kutta 方法

- 例如，取 $m = 2$ ，则 $y(t + \tau)$ 展开为

$$y(t + \tau) = y + \tau g + \frac{\tau^2}{2} (g_t + gg_y)$$

- 同样的对另一种形式的展开有

$$y(t + \tau) = y(t) + \alpha_1 c_1 + \alpha_2 c_2$$

$$c_1 = \tau g(y, t)$$

$$c_2 = \tau g(y + \nu_{21} c_1, t + \nu_{21} \tau)$$

Runge-Kutta 方法

- 将 c_2 展开到 $O(\tau^2)$ 则有

$$c_2 = \tau g + \nu_{21} \tau^2 (g_t + gg_y)$$

Runge-Kutta 方法

- 将 c_2 展开到 $O(\tau^2)$ 则有

$$c_2 = \tau g + v_{21} \tau^2 (g_t + gg_y)$$

- 则原式为

$$y(t + \tau) = y(t) + (\alpha_1 + \alpha_2) \tau g + \alpha_2 \tau^2 v_{21} (g_t + gg_y)$$

Runge-Kutta 方法

- 将 c_2 展开到 $O(\tau^2)$ 则有

$$c_2 = \tau g + v_{21} \tau^2 (g_t + gg_y)$$

- 则原式为

$$y(t + \tau) = y(t) + (\alpha_1 + \alpha_2) \tau g + \alpha_2 \tau^2 v_{21} (g_t + gg_y)$$

- 对比可得

$$\alpha_1 + \alpha_2 = 1$$

$$\alpha_2 v_{21} = \frac{1}{2}$$

Runge-Kutta 方法

- 两个方程，三个未知数，因此可以有一定的任意性，例如可以取

$$\alpha_1 = \alpha_2 = 1/2, \nu_{21} = 1$$

$$or \alpha_1 = 1/3, \alpha_2 = 2/3, \nu_{21} = 3/4$$

Runge-Kutta 方法

- 两个方程，三个未知数，因此可以有一定的任意性，例如可以取

$$\alpha_1 = \alpha_2 = 1/2, \nu_{21} = 1$$

$$or \alpha_1 = 1/3, \alpha_2 = 2/3, \nu_{21} = 3/4$$

- 这种自由度使得我们可以挑选参数使得数值精度更高，这是与问题相关的。

Runge-Kutta 方法

- 常用的 Runge-Kutta 方法则是截断到 $O(\tau^4)$. 公式为

$$y(t + \tau) = y(t) + \frac{1}{6}(c_1 + 2c_2 + 2c_3 + c_4)$$

Runge-Kutta 方法

- 常用的 Runge-Kutta 方法则是截断到 $O(\tau^4)$. 公式为

$$y(t + \tau) = y(t) + \frac{1}{6}(c_1 + 2c_2 + 2c_3 + c_4)$$

- 其中

$$c_1 = \tau g(y, t)$$

$$c_2 = \tau g\left(y + \frac{c_1}{2}, t + \frac{\tau}{2}\right)$$

$$c_3 = \tau g\left(y + \frac{c_2}{2}, t + \frac{\tau}{2}\right)$$

$$c_4 = \tau g(y + c_3, t + \tau)$$

7.1 初值问题

7.2 欧拉和皮卡德方法

7.3 预修正方法

7.4 Runge-Kutta 方法

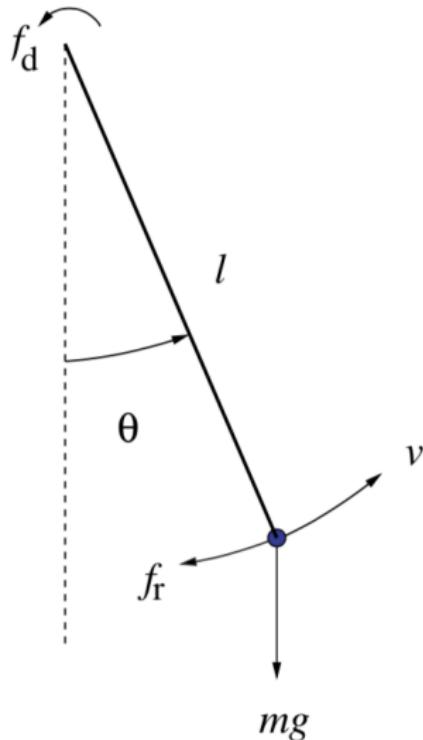
7.5 驱动摆的混沌运动

7.6 打靶法

7.7 线性方程与 Sturm-Liouville 问题

7.8 一维薛定谔方程

混沌运动



混沌运动

- 单摆受三个力的作用，驱动力 f_d ，阻力 f_r ，以及重力。可得牛顿方程

$$ma_t = f_g + f_d + f_r$$

混沌运动

- 单摆受三个力的作用，驱动力 f_d , 阻力 f_r , 以及重力. 可得牛顿方程

$$ma_t = f_g + f_d + f_r$$

- $f_g = -mg \sin \theta$ 为重力在运动方向的分量. 假设驱动力为周期性的力

$$f_d(t) = f_0 \cos \omega_0 t$$

阻力为 $f_r = -\kappa v$.

混沌运动

■ 从而方程可化为

$$\frac{d^2\theta}{dt^2} + q \frac{d\theta}{dt} + \sin \theta = b \cos \omega_0 t$$

其中 $q = \kappa/m$ 以及 $b = f_0/ml$

混沌运动

■ 从而方程可化为

$$\frac{d^2\theta}{dt^2} + q \frac{d\theta}{dt} + \sin \theta = b \cos \omega_0 t$$

其中 $q = \kappa/m$ 以及 $b = f_0/ml$

■ 设 $y_1 = \theta, y_2 = \omega = d\theta/dt$, 将高阶方程化为低阶方程组有

$$\frac{dy_1}{dt} = y_2$$

$$\frac{dy_2}{dt} = -qy_2 - \sin y_1 + b \cos \omega_0 t$$

混沌运动

- 在某些参数空间 (q, b, ω_0) 中，单摆的运动完全是混沌的。

混沌运动

- 在某些参数空间 (q, b, ω_0) 中，单摆的运动完全是混沌的。
- 试取 $\omega_0 = 2/3, q = 0.5, b = 0.9$ 给出单摆的角速度随 θ 角的变换。

混沌运动

- 在某些参数空间 (q, b, ω_0) 中，单摆的运动完全是混沌的.
- 试取 $\omega_0 = 2/3, q = 0.5, b = 0.9$ 给出单摆的角速度随 θ 角的变换.
- 试取 $\omega_0 = 2/3, q = 0.5, b = 1.15$ 给出单摆的角速度随 θ 角的变换.

7.1 初值问题

7.2 欧拉和皮卡德方法

7.3 预修正方法

7.4 Runge-Kutta 方法

7.5 驱动摆的混沌运动

7.6 打靶法

7.7 线性方程与 Sturm-Liouville 问题

7.8 一维薛定谔方程

打靶法

- 打靶法常用来求解边值问题和本征值问题.

打靶法

- 打靶法常用来求解边值问题和本征值问题.
- 首先, 假设我们将一个二阶方程转化为两个一阶方程, $y_1 = u$ 以及 $y_2 = u'$ 有

$$\frac{dy_1}{dx} = y_2$$

$$\frac{dy_2}{dx} = f(y_1, y_2; x)$$

打靶法

- 打靶法常用来求解边值问题和本征值问题.
- 首先, 假设我们将一个二阶方程转化为两个一阶方程, $y_1 = u$ 以及 $y_2 = u'$ 有

$$\frac{dy_1}{dx} = y_2$$

$$\frac{dy_2}{dx} = f(y_1, y_2; x)$$

- 假设边值条件为 $u(0) = u_0$, $u(1) = u_1$. 其它类型的边值条件也可以类似求解.

打靶法

- 求解的核心思想是映入一个可调参数，使得问题转换成初值问题.

打靶法

- 求解的核心思想是映入一个可调参数，使得问题转换成初值问题.
- 由于 $u(0)$ 已知, 猜测一个一阶导数的初值 $u'(0) = \alpha$. α 是一个可调参数.

打靶法

- 求解的核心思想是映入一个可调参数，使得问题转换成初值问题.
- 由于 $u(0)$ 已知, 猜测一个一阶导数的初值 $u'(0) = \alpha$. α 是一个可调参数.
- 对于一个确定的 α , 我们可以采用之前的方法求解出一个解, 从而得到一个在 1 处的 $u_\alpha(1)$.

打靶法

- 求解的核心思想是映入一个可调参数，使得问题转换成初值问题.
- 由于 $u(0)$ 已知, 猜测一个一阶导数的初值 $u'(0) = \alpha$. α 是一个可调参数.
- 对于一个确定的 α , 我们可以采用之前的方法求解出一个解, 从而得到一个在 1 处的 $u_\alpha(1)$.
- 此时 $u_\alpha(1)$ 不会等于 u_1 . 打靶法的思想是采用一个求根方法使得 $f(\alpha) = u_\alpha(1) - u_1 = 0$.

打靶法

■ 具体实例，假设我们要求解方程

$$u'' = -\frac{\pi^2}{4}(u + 1)$$

给定边值 $u(0) = 0, u(1) = 1.$

打靶法

- 具体实例，假设我们要求解方程

$$u'' = -\frac{\pi^2}{4}(u + 1)$$

给定边值 $u(0) = 0, u(1) = 1.$

- 令 $y_1 = u, y_2 = u'$, 可得

$$\frac{dy_1}{dx} = y_2$$

$$\frac{dy_2}{dx} = -\frac{\pi^2}{4}(y_1 + 1)$$

打靶法

- 首先猜一个初值为 $y_2(0) = \alpha$.

打靶法

- 首先猜一个初值为 $y_2(0) = \alpha$.
- 利用割线法去寻找方程的根 $f(\alpha) = u_\alpha(1) - 1 = 0$

打靶法

- 首先猜一个初值为 $y_2(0) = \alpha$.
- 利用割线法去寻找方程的根 $f(\alpha) = u_\alpha(1) - 1 = 0$
- 利用 Runge-Kutta 去求解每一个 α 下的方程的解 $u_\alpha(1)$.

打靶法

- 首先猜一个初值为 $y_2(0) = \alpha$.
- 利用割线法去寻找方程的根 $f(\alpha) = u_\alpha(1) - 1 = 0$
- 利用 Runge-Kutta 去求解每一个 α 下的方程的解 $u_\alpha(1)$.
- 练习：利用打靶法求解该方程，并与解析解对比

$$u(x) = \cos \frac{\pi x}{2} + 2 \sin \frac{\pi x}{2} - 1$$

打靶法

- 首先猜一个初值为 $y_2(0) = \alpha$.
- 利用割线法去寻找方程的根 $f(\alpha) = u_\alpha(1) - 1 = 0$
- 利用 Runge-Kutta 去求解每一个 α 下的方程的解 $u_\alpha(1)$.
- 练习：利用打靶法求解该方程，并与解析解对比
$$u(x) = \cos \frac{\pi x}{2} + 2 \sin \frac{\pi x}{2} - 1$$
- 如果边界条件为 $u'(0) = u_0, u(1) = u_1$, 我们可以猜测一个 $u(0) = \alpha$ 来使用打靶法.

打靶法

- 如果我们求解的是一个本征值问题，则可调参数变为问题中的本征值.

打靶法

- 如果我们求解的是一个本征值问题，则可调参数变为问题中的本征值.
- 例如，如果给定了 $u(0) = u_0, u(1) = u_1$ ，我们选择 $u'(0) = \alpha$ 来求解方程.

打靶法

- 如果我们求解的是一个本征值问题，则可调参数变为问题中的本征值.
- 例如，如果给定了 $u(0) = u_0, u(1) = u_1$ ，我们选择 $u'(0) = \alpha$ 来求解方程.
- 之后我们通过改变 λ 来求解方程 $f(\lambda) = u_\lambda(1) - u_1 = 0$.

打靶法

- 如果我们求解的是一个本征值问题，则可调参数变为问题中的本征值.
- 例如，如果给定了 $u(0) = u_0, u(1) = u_1$ ，我们选择 $u'(0) = \alpha$ 来求解方程.
- 之后我们通过改变 λ 来求解方程 $f(\lambda) = u_\lambda(1) - u_1 = 0$.
- 当 $f(\lambda) = 0$ ，则我们即得到了本征方程的 λ 以及相应的本征函数 $u_\lambda(x)$. 这里的 α 其实并不相关，它在求解过程中自动进行了调节.

7.1 初值问题

7.2 欧拉和皮卡德方法

7.3 预修正方法

7.4 Runge-Kutta 方法

7.5 驱动摆的混沌运动

7.6 打靶法

7.7 线性方程与 Sturm-Liouville 问题

7.8 一维薛定谔方程

线性方程与 Sturm-Liouville 问题

- 一些本征值问题或边值问题以线性方程的形式出现, 如

$$u'' + d(x)u' + q(x)u = s(x)$$

其中 $d(x), q(x), s(x)$ 都是 x 的函数.

线性方程与 Sturm-Liouville 问题

- 对于这类问题，我们同样可以采用打靶法来求解。
不过并不需要大量的计算不同的 α 来求解 $f(\alpha) = u_\alpha(1) - u_1 = 0$.

线性方程与 Sturm-Liouville 问题

- 对于这类问题，我们同样可以采用打靶法来求解。不过并不需要大量的计算不同的 α 来求解 $f(\alpha) = u_\alpha(1) - u_1 = 0$.
- 由于线性方程的叠加原理；解的线性组合同样是方程的解. 我们只需要两个试探解 $u_{\alpha_0}(x)$ 和 $u_{\alpha_1}(x)$. 方程的正确解为

$$u(x) = au_{\alpha_0}(x) + bu_{\alpha_1}(x)$$

其中 a, b 决定于 $u(0) = u_0, u(1) = u_1$.

线性方程与 Sturm-Liouville 问题

■ 因为 $u_{\alpha_0}(0) = u_{\alpha_1}(0) = u(0) = u_0$, 所以我们有

$$a + b = 1$$

$$u_{\alpha_0}(1)a + u_{\alpha_1}(1)b = u_1$$

线性方程与 Sturm-Liouville 问题

■ 因为 $u_{\alpha_0}(0) = u_{\alpha_1}(0) = u(0) = u_0$, 所以我们有

$$a + b = 1$$

$$u_{\alpha_0}(1)a + u_{\alpha_1}(1)b = u_1$$

■ 从而有

$$a = \frac{u_{\alpha_1}(1) - u_1}{u_{\alpha_1}(1) - u_{\alpha_0}(1)}$$

$$b = \frac{u_1 - u_{\alpha_0}(1)}{u_{\alpha_1}(1) - u_{\alpha_0}(1)}$$

线性方程与 Sturm-Liouville 问题

- 因为 $u_{\alpha_0}(0) = u_{\alpha_1}(0) = u(0) = u_0$, 所以我们有

$$a + b = 1$$

$$u_{\alpha_0}(1)a + u_{\alpha_1}(1)b = u_1$$

- 从而有

$$a = \frac{u_{\alpha_1}(1) - u_1}{u_{\alpha_1}(1) - u_{\alpha_0}(1)}$$

$$b = \frac{u_1 - u_{\alpha_0}(1)}{u_{\alpha_1}(1) - u_{\alpha_0}(1)}$$

- 得到 a, b 后, 我们即得到了微分方程的解.

线性方程与 Sturm-Liouville 问题

- 物理中一组重要的线性方程为 Sturm-Liouville 问题，

$$[p(x)u'(x)]' + q(x)u(x) = s(x)$$

线性方程与 Sturm-Liouville 问题

- 物理中一组重要的线性方程为 Sturm-Liouville 问题,

$$[p(x)u'(x)]' + q(x)u(x) = s(x)$$

- 在许多实际问题中 $s(x) = 0$, 以及 $q(x) = -r(x) + \lambda w(x)$. 其中 λ 为方程的本征值.

$$(p(x)u'(x))' - r(x)u(x) = -\lambda w(x)u(x)$$

线性方程与 Sturm-Liouville 问题

- 物理中一组重要的线性方程为 Sturm-Liouville 问题,

$$[p(x)u'(x)]' + q(x)u(x) = s(x)$$

- 在许多实际问题中 $s(x) = 0$, 以及 $q(x) = -r(x) + \lambda w(x)$. 其中 λ 为方程的本征值.

$$(p(x)u'(x))' - r(x)u(x) = -\lambda w(x)u(x)$$

- 勒让德方程、贝塞尔方程、以及它们相关的方程都是 Sturm-Liouville 方程的例子.

线性方程与 Sturm-Liouville 问题

■ 对一阶导数和二阶导数，我们有三点公式

$$\Delta_1 = \frac{u_{i+1} - u_{i-1}}{2h} = u'_i + \frac{h^2 u_i^{(3)}}{6} + O(h^4) \quad (3)$$

$$\Delta_2 = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} = u''_i + \frac{h^2 u_i^{(4)}}{12} + O(h^4) \quad (4)$$

线性方程与 Sturm-Liouville 问题

■ 对一阶导数和二阶导数，我们有三点公式

$$\Delta_1 = \frac{u_{i+1} - u_{i-1}}{2h} = u'_i + \frac{h^2 u_i^{(3)}}{6} + O(h^4) \quad (3)$$

$$\Delta_2 = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} = u''_i + \frac{h^2 u_i^{(4)}}{12} + O(h^4) \quad (4)$$

■ 对上面的公式 (3) 乘以 p'_i ，对公式 (4) 乘以 p_i ，
并求和可得

$$p'_i \Delta_1 + p_i \Delta_2 = (p_i u'_i)' + \frac{h^2}{12} (p_i u_i^{(4)} + 2p'_i u_i^{(3)}) + O(h^4)$$

线性方程与 Sturm-Liouville 问题

- 利用原始的微分方程，可将等式的右边第一项替换成 $s_i - q_i u_i$ 并扔掉第二项，可以得到最简单的数值解法

$$(2p_i + hp'_i)u_{i+1} + (2p_i - hp'_i)u_{i-1} = 4p_i u_i + 2h^2(s_i - q_i u_i)$$

线性方程与 Sturm-Liouville 问题

- 利用原始的微分方程，可将等式的右边第一项替换成 $s_i - q_i u_i$ 并扔掉第二项，可以得到最简单的数值解法

$$(2p_i + hp'_i)u_{i+1} + (2p_i - hp'_i)u_{i-1} = 4p_i u_i + 2h^2(s_i - q_i u_i)$$

- 该算法的精度为 $O(h^4)$

线性方程与 Sturm-Liouville 问题

例 3 勒让德方程

$$\frac{d}{dx} \left[(1-x^2) \frac{du}{dx} \right] + l(l+1)u = 0$$

其中 $l = 0, 1, \dots, \infty$, $x \in [-1, 1]$. 假设我们不知道 l 的值, 但是知道 $P_l(x) = x$ 的前两点的值. 试利用刚刚的算法及割线法求解该勒让德方程.

Numerov 算法

- Numerov 算法则是对公式 (4) 的四阶导数进行替换，根据方程有

$$u^{(4)}(x) = \frac{d^2}{dx^2}[-q(x)u(x) + s(x)]$$

Numerov 算法

- Numerov 算法则是对公式 (4) 的四阶导数进行替换，根据方程有

$$u^{(4)}(x) = \frac{d^2}{dx^2}[-q(x)u(x) + s(x)]$$

- 将三点公式应用于等式的右边有

$$\begin{aligned} u^{(4)}(x) &= \frac{(s_{i+1} - q_{i+1}u_{i+1}) - 2(s_i - q_iu_i)}{h^2} \\ &\quad + \frac{(s_{i-1} - q_{i-1}u_{i-1})}{h^2} \end{aligned}$$

Numerov 算法

- 结合等式 $\Delta_2 = (u_{i+1} - 2u_i + u_{i-1})/h^2 = u''_i + h^2 u^{(4)}/12$ 以及 $u''_i = s_i - q_i u_i$. 我们可以得到
$$c_{i+1}u_{i+1} + c_{i-1}u_{i-1} = c_i u_i + d_i + O(h^6)$$

Numerov 算法

■ 其中

$$c_{i+1} = 1 + \frac{h^2}{12} q_{i+1}$$

$$c_{i-1} = 1 + \frac{h^2}{12} q_{i-1}$$

$$c_i = 2 - \frac{5h^2}{6} q_i$$

$$d_i = \frac{h^2}{12} (s_{i+1} + 10s_i + s_{i-1})$$

Numerov 算法

■ 其中

$$c_{i+1} = 1 + \frac{h^2}{12} q_{i+1}$$

$$c_{i-1} = 1 + \frac{h^2}{12} q_{i-1}$$

$$c_i = 2 - \frac{5h^2}{6} q_i$$

$$d_i = \frac{h^2}{12} (s_{i+1} + 10s_i + s_{i-1})$$

■ 注意由于重复使用了三点公式，Numerov 算法精度为 $O(h^4)$

7.1 初值问题

7.2 欧拉和皮卡德方法

7.3 预修正方法

7.4 Runge-Kutta 方法

7.5 驱动摆的混沌运动

7.6 打靶法

7.7 线性方程与 Sturm-Liouville 问题

7.8 一维薛定谔方程

一维薛定谔方程

- 求解一维薛定谔方程对我们理解量子力学和量子过程非常重要.

一维薛定谔方程

- 求解一维薛定谔方程对我们理解量子力学和量子过程非常重要.
- 一维薛定谔方程

$$-\frac{\hbar^2}{2m} \frac{d^2\phi(x)}{dx^2} + V(x)\phi(x) = \varepsilon\phi(x)$$

一维薛定谔方程

- 求解一维薛定谔方程对我们理解量子力学和量子过程非常重要.
- 一维薛定谔方程

$$-\frac{\hbar^2}{2m} \frac{d^2\phi(x)}{dx^2} + V(x)\phi(x) = \varepsilon\phi(x)$$

- 可以将方程重写为

$$\phi''(x) + \frac{2m}{\hbar^2} [\varepsilon - V(x)]\phi(x) = 0$$

一维薛定谔方程

- 求解一维薛定谔方程对我们理解量子力学和量子过程非常重要.
- 一维薛定谔方程

$$-\frac{\hbar^2}{2m} \frac{d^2\phi(x)}{dx^2} + V(x)\phi(x) = \epsilon\phi(x)$$

- 可以将方程重写为

$$\phi''(x) + \frac{2m}{\hbar^2} [\epsilon - V(x)]\phi(x) = 0$$

- 即相当于 Sturm-Liouville 方程中, 取 $p(x) = 1$, $q(x) = 2m[\epsilon - V(x)]/\hbar^2$, 以及 $s(x) = 0$.

本征值问题

- 在本征值问题中，粒子被禁闭在势阱 $V(x)$ 中，使得当 $|x| \rightarrow \infty$ 时， $\phi(x) \rightarrow 0$.

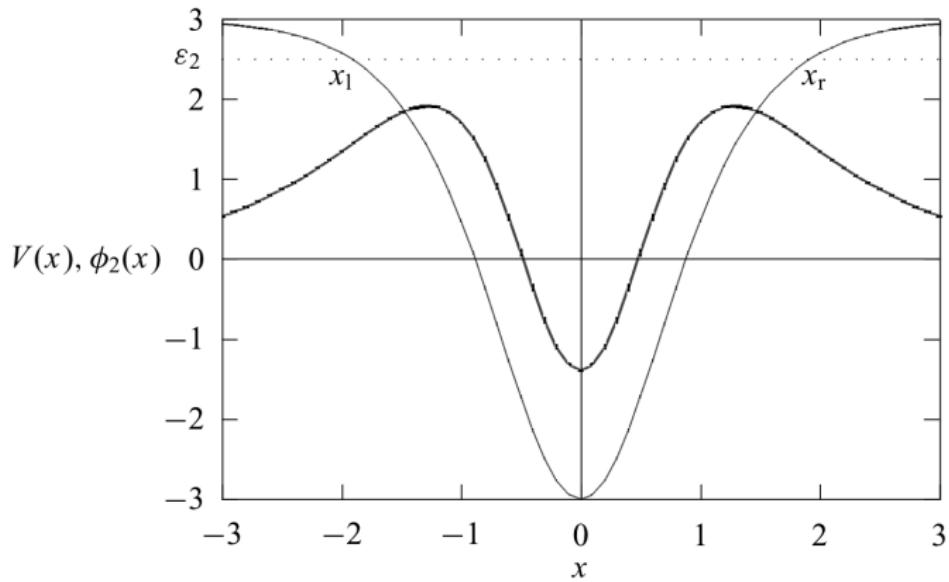
本征值问题

- 在本征值问题中，粒子被禁闭在势阱 $V(x)$ 中，使得当 $|x| \rightarrow \infty$ 时， $\phi(x) \rightarrow 0$.
- 为了求解该方程，我们可以采用 Numerov 算法在该势场的区域从左到右或从右到左进行积分.

本征值问题

- 在本征值问题中，粒子被禁闭在势阱 $V(x)$ 中，使得当 $|x| \rightarrow \infty$ 时， $\phi(x) \rightarrow 0$.
- 为了求解该方程，我们可以采用 Numerov 算法在该势场的区域从左到右或从右到左进行积分.
- 整个积分过程会通过一个指数增长的过程，再到中心的振荡过程，最后到远处的指数衰减过程.

本征值问题



本征值问题

- 由于指数增长的解同样是方程的解，导致我们在从振荡区域到指数增长区域积分时，会带来明显的误差累积.

本征值问题

- 由于指数增长的解同样是方程的解，导致我们在从振荡区域到指数增长区域积分时，会带来明显的误差累积.
- 经验法则可以用来避免指数区域的积分. 即从两面来求解方程，并在势阱区域来匹配它们.

本征值问题

- 由于指数增长的解同样是方程的解，导致我们在从振荡区域到指数增长区域积分时，会带来明显的误差累积.
- 经验法则可以用来避免指数区域的积分. 即从两面来求解方程，并在势阱区域来匹配它们.
- 通常匹配点取在一个转折点，即能量等于势能的位置，如图中的 x_l 和 x_r .

本征值问题

- 这里的匹配过程为，通过调节试验的本征值，直到两个解： ϕ_r （从右边积分过来的解） ϕ_l （从左边积分过来的解）在转折点满足连续性条件（这里转折点取 x_r ）

$$\phi_1(x_r) = \phi_r(x_r)$$

$$\phi'_1(x_r) = \phi'_r(x_r)$$

本征值问题

- 这里的匹配过程为，通过调节试验的本征值，直到两个解： ϕ_r （从右边积分过来的解） ϕ_l （从左边积分过来的解）在转折点满足连续性条件（这里转折点取 x_r ）

$$\phi_1(x_r) = \phi_r(x_r)$$

$$\phi'_1(x_r) = \phi'_r(x_r)$$

- 联立这两个条件，有

$$\frac{\phi'_1(x_r)}{\phi_1(x_r)} = \frac{\phi'_r(x_r)}{\phi_r(x_r)}$$

本征值问题

■ 对上面方程的一阶导数使用三点函数，可以得到

$$\begin{aligned} f(\varepsilon) &= \frac{[\phi_1(x_r + h) - \phi_1(x_r - h)]}{2h\phi(x_r)} \\ &\quad - \frac{[\phi_r(x_r + h) - \phi_r(x_r - h)]}{2h\phi(x_r)} \\ &= 0 \end{aligned}$$

本征值问题

- 对上面方程的一阶导数使用三点函数，可以得到

$$\begin{aligned}f(\varepsilon) &= \frac{[\phi_1(x_r + h) - \phi_1(x_r - h)]}{2h\phi(x_r)} \\&\quad - \frac{[\phi_r(x_r + h) - \phi_r(x_r - h)]}{2h\phi(x_r)} \\&= 0\end{aligned}$$

- 该方程用来保证求根的过程.

本征值问题

- 总结数值求解一维薛定谔方程的步骤：

本征值问题

- 总结数值求解一维薛定谔方程的步骤：
 - 选择一个求解的区域，保证区域外势场对解的影响可以忽略。

本征值问题

- 总结数值求解一维薛定谔方程的步骤：
 - 选择一个求解的区域，保证区域外势场对解的影响可以忽略.
 - 提供一个对最低本征值的合理猜测值.

本征值问题

- 总结数值求解一维薛定谔方程的步骤：
 - 选择一个求解的区域，保证区域外势场对解的影响可以忽略。
 - 提供一个对最低本征值的合理猜测值。
 - 从左到点 $x_r + h$ 进行积分得到 $\phi_l(x)$ ，从右到点 $x_r - h$ 积分出 $\phi_r(x)$. 在匹配前，将两个解重标度使得 $\phi_l(x_r) = \phi_r(x_r)$

本征值问题

■ 总结数值求解一维薛定谔方程的步骤：

- 选择一个求解的区域，保证区域外势场对解的影响可以忽略。
- 提供一个对最低本征值的合理猜测值。
- 从左到点 $x_r + h$ 进行积分得到 $\phi_l(x)$ ，从右到点 $x_r - h$ 积分出 $\phi_r(x)$. 在匹配前，将两个解重标度使得 $\phi_l(x_r) = \phi_r(x_r)$
- 计算 $f(\epsilon_0) = [\phi_r(x_r - h) - \phi_r(x_r + h) - \phi_l(x_r - h)]$ 并应用求根公式求解得到 ϵ_0 .

本征值问题

例 4 求解如下势阱下的薛定谔方程 (取 $\alpha = 1, \lambda = 4$ 以及 $\hbar = 1, m = 1$)

$$V(x) = \frac{\hbar^2}{2m} \alpha^2 \lambda (\lambda - 1) \left[\frac{1}{2} - \frac{1}{\cosh^2(\alpha x)} \right]$$

并与解析解进行对比

$$\varepsilon_n = \frac{\hbar^2}{2m} \alpha^2 \left[\frac{\lambda(\lambda - 1)}{2} - (\lambda - 1 - n)^2 \right]$$

第四节 Mathematica 基础

第五节 函数近似

第六节 数值微积分

第七节 常微分方程

第八节 矩阵的数值方法

第八节

矩阵的数值方法

8.1

物理学中的矩阵

8.2

基本的矩阵操作

8.3

线性方程系统

8.4

多变量函数的根和极值

8.5

本征值问题

物理学中的矩阵

- 物理学中，许多问题都可以转化为矩阵问题

物理学中的矩阵

- 物理学中，许多问题都可以转化为矩阵问题
- 例如，我们要研究具有 n 个振动自由度的分子的振动光谱。首先将势能在平衡结构附近展开到二阶项，研究系统的简谐振动。势能可以表示为

$$U(q_1, q_2, \dots, q_n) \simeq \frac{1}{2} \sum_{i,j=1}^n A_{ij} q_i q_j$$

物理学中的矩阵

- 物理学中，许多问题都可以转化为矩阵问题
- 例如，我们要研究具有 n 个振动自由度的分子的振动光谱。首先将势能在平衡结构附近展开到二阶项，研究系统的简谐振动。势能可以表示为

$$U(q_1, q_2, \dots, q_n) \simeq \frac{1}{2} \sum_{i,j=1}^n A_{ij} q_i q_j$$

- 将平衡态的势能看作零点，而系统的动能可以表示为

$$T(\dot{q}_1, \dot{q}_2, \dots, \dot{q}_n) \simeq \frac{1}{2} \sum_{i,j=1}^n M_{ij} \dot{q}_i \dot{q}_j$$

物理学中的矩阵

- 系统的拉格朗日量为 $T - U$, 应用拉格朗日方程有

$$\frac{\partial \mathcal{L}}{\partial q_i} - \frac{d}{dt} \frac{\partial \mathcal{L}}{\partial \dot{q}_i} = 0$$

物理学中的矩阵

■ 系统的拉格朗日量为 $T - U$, 应用拉格朗日方程有

$$\frac{\partial \mathcal{L}}{\partial q_i} - \frac{d}{dt} \frac{\partial \mathcal{L}}{\partial \dot{q}_i} = 0$$

■ 从而可得

$$\sum_{j=1}^n (A_{ij}q_j + M_{ij}\ddot{q}_j) = 0$$

物理学中的矩阵

- 假设广义坐标对时间的依赖是简谐的，即

$$q_j = x_j e^{-i\omega t}$$

物理学中的矩阵

- 假设广义坐标对时间的依赖是简谐的，即

$$q_j = x_j e^{-i\omega t}$$

- 则有

$$\sum_{j=1}^n (A_{ij} - \omega^2 M_{ij}) x_j = 0$$

物理学中的矩阵

■ 上面的方程写成矩阵形式为

$$\begin{pmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \vdots & \vdots \\ A_{n1} & \cdots & A_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \lambda \begin{pmatrix} M_{11} & \cdots & M_{1n} \\ \vdots & \vdots & \vdots \\ M_{n1} & \cdots & M_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

物理学中的矩阵

■ 上面的方程写成矩阵形式为

$$\begin{pmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \vdots & \vdots \\ A_{n1} & \cdots & A_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \lambda \begin{pmatrix} M_{11} & \cdots & M_{1n} \\ \vdots & \vdots & \vdots \\ M_{n1} & \cdots & M_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

■ 等价为

$$A \cdot x = \lambda M \cdot x$$

物理学中的矩阵

- 这是一个齐次线性方程组，其有非零平庸解的条件为

$$|A - \lambda M| = 0$$

物理学中的矩阵

- 这是一个齐次线性方程组，其有非零平庸解的条件为

$$|A - \lambda M| = 0$$

- 该方程的解， λ_k 给出了分子所有可能的振动角频率 $\omega_k = \sqrt{\lambda_k}$.

第八节

矩阵的数值方法

8.1

物理学中的矩阵

8.2

基本的矩阵操作

8.3

线性方程系统

8.4

多变量函数的根和极值

8.5

本征值问题

基本定义

- 将 $n \times m$ 个元素按行列排列 A_{ij} 而成的矩形集合称为矩阵.

基本定义

- 将 $n \times m$ 个元素按行列排列 A_{ij} 而成的矩形集合称为矩阵.
- 若 $n = m$ 则该矩阵称为方阵.

基本定义

- 将 $n \times m$ 个元素按行列排列 A_{ij} 而成的矩形集合称为矩阵.
- 若 $n = m$ 则该矩阵称为方阵.
- n 个元素构成一列元素的集合可以视为一个 $n \times 1$ 的矩阵.

基本定义

■ 一个典型的线性代数方程为

$$\sum_{j=1}^n A_{ij}x_j = b_i$$

基本定义

- 一个典型的线性代数方程为

$$\sum_{j=1}^n A_{ij}x_j = b_i$$

- 也可将其写为

$$A \cdot x = b$$

基本定义

- 一个典型的线性代数方程为

$$\sum_{j=1}^n A_{ij}x_j = b_i$$

- 也可将其写为

$$A \cdot x = b$$

- 矩阵的乘法

$$C_{ij} = \sum_k A_{ik}B_{kj}; \quad C = A \cdot B$$

基本定义

- 一个方阵的逆矩阵定义为

$$A^{-1}A = AA^{-1} = I$$

基本定义

- 一个方阵的逆矩阵定义为

$$A^{-1}A = AA^{-1} = I$$

- 一个 $n \times n$ 矩阵 A 的行列式定义为

$$|\mathbf{A}| = \sum_{i=1}^n (-1)^{i+j} A_{ij} |\mathbf{R}_{ij}|$$

基本定义

- 一个方阵的逆矩阵定义为

$$A^{-1}A = AA^{-1} = I$$

- 一个 $n \times n$ 矩阵 A 的行列式定义为

$$|\mathbf{A}| = \sum_{i=1}^n (-1)^{i+j} A_{ij} |R_{ij}|$$

- 其中 $|R_{ij}|$ 为余子式的行列式. $C_{ij} = (-1)^{i+j} |R_{ij}|$ 称为矩阵 A 的代数余子式.

基本定义

■ 原则上，一个矩阵的逆可以通过如下求得

$$A_{ij}^{-1} = \frac{C_{ji}}{|A|}$$

基本定义

- 原则上，一个矩阵的逆可以通过如下求得

$$A_{ij}^{-1} = \frac{C_{ji}}{|A|}$$

- 如果一个矩阵有逆或者有非零的行列式，则称为非奇异矩阵，否则称为奇异矩阵.

基本定义

- 原则上，一个矩阵的逆可以通过如下求得

$$A_{ij}^{-1} = \frac{C_{ji}}{|A|}$$

- 如果一个矩阵有逆或者有非零的行列式，则称为非奇异矩阵，否则称为奇异矩阵.
- 矩阵的迹为所有对角元素的和，写作

$$\text{Tr } \mathbf{A} = \sum_{i=1}^n A_{ii}$$

基本定义

- 一个矩阵的转置为其行列指标交换，即

$$(A^T)_{ij} = A_{ji}$$

基本定义

- 一个矩阵的转置为其行列指标交换，即

$$(A^T)_{ij} = A_{ji}$$

- 如果满足 $A^T = A^{-1}$ ，则称 A 为正交矩阵.

基本定义

- 一个矩阵的转置为其行列指标交换，即

$$(A^T)_{ij} = A_{ji}$$

- 如果满足 $A^T = A^{-1}$ ，则称 A 为正交矩阵.
- 一个矩阵转置后取复共轭，称为厄米操作，记为 $(A^\dagger)_{ij} = A_{ji}^*$. 如果满足 $A^\dagger = A$ ，则称 A 为厄米矩阵，如果满足 $A^\dagger = A^{-1}$ ，则称 A 为么正矩阵.

基本操作

- 一个常见的操作是将某一个行乘以一个因子加到另一行，即

$$A'_{ij} = A_{ij} + \lambda A_{kj} \quad j = 1, 2, \dots, n$$

基本操作

- 一个常见的操作是将某一个行乘以一个因子加到另一行，即

$$A'_{ij} = A_{ij} + \lambda A_{kj} \quad j = 1, 2, \dots, n$$

- 该操作保持行列式不变，并可以写成一个矩阵乘法

$$A' = MA$$

基本操作

- 一个常见的操作是将某一个行乘以一个因子加到另一行，即

$$A'_{ij} = A_{ij} + \lambda A_{kj} \quad j = 1, 2, \dots, n$$

- 该操作保持行列式不变，并可以写成一个矩阵乘法

$$A' = MA$$

- M 为单位矩阵加上只有一个不为零非对角元素 $M_{ik} = \lambda$ 的矩阵.

基本操作

■ 矩阵的本征值问题，即求解方程

$$A \cdot x = \lambda x$$

其中 λ 为本征值， x 为本征矢量.

基本操作

- 矩阵的本征值问题，即求解方程

$$A \cdot x = \lambda x$$

其中 λ 为本征值， x 为本征矢量.

- 如前面的分子振动问题，即可转化为本征值问题，令 $B = M^{-1} \cdot A$ ，则原方程变为

$$B \cdot x = \lambda x$$

基本操作

- 矩阵的本征值问题也可以视为迭代求解线性方程组.

基本操作

- 矩阵的本征值问题也可以视为迭代求解线性方程组.
- 如为求解上面的本征值本征矢，可以采用如下迭代解法

$$\mathbf{A}\mathbf{x}^{(k+1)} = \lambda^{(k)}\mathbf{x}^{(k)}$$

基本操作

- 矩阵的本征值问题也可以视为迭代求解线性方程组.
- 如为求解上面的本征值本征矢，可以采用如下迭代解法

$$\mathbf{A}\mathbf{x}^{(k+1)} = \lambda^{(k)}\mathbf{x}^{(k)}$$

- 其中 $\lambda^{(k)}, \mathbf{x}^{(k)}$ 为第 k 次迭代的近似本征值和本征矢.

基本操作

- 如果另一个矩阵 B 是矩阵 A 的相似变换即

$$B = S^{-1}AS$$

基本操作

- 如果另一个矩阵 B 是矩阵 A 的相似变换即

$$B = S^{-1}AS$$

- 则 B 与 A 具有相同的本征值及行列式. 这里 S 为非奇异矩阵.

第八节 矩阵的数值方法

8.1 物理学中的矩阵

8.2 基本的矩阵操作

8.3 线性方程系统

8.4 多变量函数的根和极值

8.5 本征值问题

线性方程系统

- 如果一个矩阵的对角元以下(以上)的元素都为零, 则称该矩阵为上三角(下三角)矩阵.

线性方程系统

- 如果一个矩阵的对角元以下(以上)的元素都为零, 则称该矩阵为上三角(下三角)矩阵.
- 高斯消元法则是求解矩阵问题最简单的方案, 通过首先将系数矩阵化为上三角矩阵, 然后再回代方程进行求解.

线性方程系统

- 如果一个矩阵的对角元以下(以上)的元素都为零, 则称该矩阵为上三角(下三角)矩阵.
- 高斯消元法则是求解矩阵问题最简单的方案, 通过首先将系数矩阵化为上三角矩阵, 然后再回代方程进行求解.
- 该方法能同时求解矩阵的逆及行列式.

线性方程系统

- 如果一个矩阵的对角元以下(以上)的元素都为零, 则称该矩阵为上三角(下三角)矩阵.
- 高斯消元法则是求解矩阵问题最简单的方案, 通过首先将系数矩阵化为上三角矩阵, 然后再回代方程进行求解.
- 该方法能同时求解矩阵的逆及行列式.
- 这里我们首先求解线性方程组

$$A \cdot x = b$$

高斯消元法

- 高斯消元法的目标是将矩阵 A 转化为一个上三角矩阵（下三角矩阵），使得方程变为

$$A^{(n-1)}x = b^{(n-1)}$$

其中当 $i > j, A_{ij}^{(n-1)} = 0.$

高斯消元法

- 高斯消元法的目标是将矩阵 A 转化为一个上三角矩阵 (下三角矩阵), 使得方程变为

$$A^{(n-1)}x = b^{(n-1)}$$

其中当 $i > j, A_{ij}^{(n-1)} = 0.$

- 步骤: 首先我们将第一行的方程乘以 $-A_{i1}^{(0)}/A_{11}^{(0)}$. 然后加结果加到下面其他的方程上去. 这样第一列的元素, 除第一行的元素外, 都被消去.

高斯消元法

- 高斯消元法的目标是将矩阵 A 转化为一个上三角矩阵(下三角矩阵),使得方程变为

$$A^{(n-1)}x = b^{(n-1)}$$

其中当 $i > j, A_{ij}^{(n-1)} = 0.$

- 步骤:首先我们将第一行的方程乘以 $-A_{i1}^{(0)}/A_{11}^{(0)}$. 然后加结果加到下面其他的方程上去. 这样第一列的元素,除第一行的元素外,都被消去.
- 然后我们将第二个方程乘以 $-A_{i2}^{(1)}/A_{22}^{(1)}$, 加到 $i > 2$ 的所有方程上,这样第二列除第一第二个元素外,都变成了零.

高斯消元法

- 重复该过程，这样我们即可以得到一个上三角矩阵 $A^{(n-1)}$. 之后进行回代就可以得到方程组的解.

高斯消元法

- 重复该过程，这样我们即可以得到一个上三角矩阵 $A^{(n-1)}$. 之后进行回代就可以得到方程组的解.
- 在计算过程中，对角元会做为每一次的除数，如果对角元为零或非常小的值，则该算法会失效.

高斯消元法

- 重复该过程，这样我们即可以得到一个上三角矩阵 $A^{(n-1)}$. 之后进行回代就可以得到方程组的解.
- 在计算过程中，对角元会做为每一次的除数，如果对角元为零或非常小的值，则该算法会失效.
- 一个解决的办法则是在每一次消元前，先做一次选主元的过程.

高斯消元法

- 重复该过程，这样我们即可以得到一个上三角矩阵 $A^{(n-1)}$. 之后进行回代就可以得到方程组的解.
- 在计算过程中，对角元会做为每一次的除数，如果对角元为零或非常小的值，则该算法会失效.
- 一个解决的办法则是在每一次消元前，先做一次选主元的过程.
- 例如，在第一次消元前，先比较第一列所有的元素，把具有最大元素的行和第一行进行交换.

高斯消元法

- 在第二次消元前，对比第二列，除第一个元素外其它元素的大小，把最大元素所在行与第二行进行交换。

高斯消元法

- 在第二次消元前，对比第二列，除第一个元素外其它元素的大小，把最大元素所在行与第二行进行交换。
- 在每次消元前，进行选主元的步骤，直到化为上三角矩阵。

高斯消元法

- 在第二次消元前，对比第二列，除第一个元素外其它元素的大小，把最大元素所在行与第二行进行交换。
- 在每次消元前，进行选主元的步骤，直到化为上三角矩阵。
- 高斯消元法的程序见 libsec5.c

矩阵的行列式

- 矩阵经过高斯消元法化成三角矩阵后，很容易即可计算其行列式.

矩阵的行列式

- 矩阵经过高斯消元法化成三角矩阵后，很容易即可计算其行列式.
- 在我们对原始矩阵进行行交换后，其行列式的值只可能差一个符号. 符号与交换的顺序有关.

矩阵的行列式

- 矩阵经过高斯消元法化成三角矩阵后，很容易即可计算其行列式.
- 在我们对原始矩阵进行行交换后，其行列式的值只可能差一个符号. 符号与交换的顺序有关.
- 行列式的值为所有对角元的乘积再乘上高斯消元过程的符号.

矩阵的行列式

- 矩阵经过高斯消元法化成三角矩阵后，很容易即可计算其行列式.
- 在我们对原始矩阵进行行交换后，其行列式的值只可能差一个符号. 符号与交换的顺序有关.
- 行列式的值为所有对角元的乘积再乘上高斯消元过程的符号.
- 求矩阵的行列式的程序见 libsec5.c 及 sec5.c

线性方程的求解

- 在得到化为三角矩阵的系数矩阵后，我们可以给出线性方程的解

$$x_i = \frac{1}{A_{k_i i}^{(n-1)}} \left(b_{k_i}^{(n-1)} - \sum_{j=i+1}^n A_{k_i j}^{(n-1)} x_j \right), i = n-1, \dots, 1$$

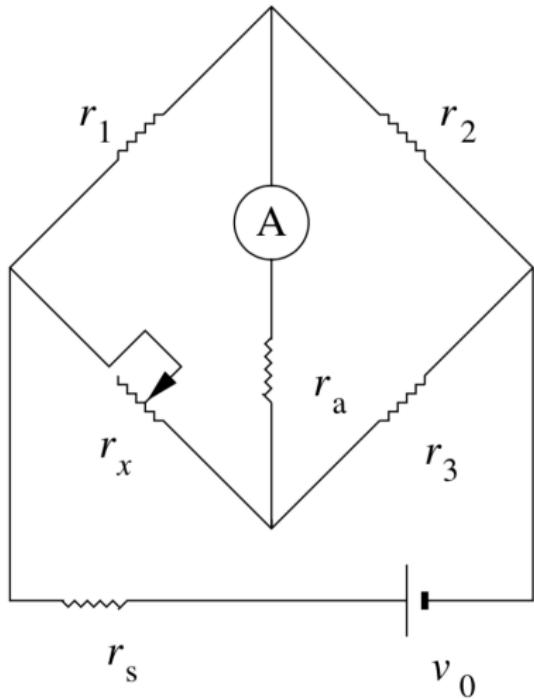
线性方程的求解

- 在得到化为三角矩阵的系数矩阵后，我们可以给出线性方程的解

$$x_i = \frac{1}{A_{k_i i}^{(n-1)}} \left(b_{k_i}^{(n-1)} - \sum_{j=i+1}^n A_{k_i j}^{(n-1)} x_j \right), i = n-1, \dots, 1$$

- 其中 $x_n = b_{k_n}^{(n-1)} / A_{k_n n}^{(n-1)}$. 其中 k_i 代表第 i 列的主元所在的行指标.

惠斯登电桥



惠斯登电桥

例 1 通过三个相互独立的圈，可以建立方程组

$$r_s i_1 + r_1 i_2 + r_2 i_3 = v_0$$

$$-r_x i_1 + (r_1 + r_x + r_a) i_2 - r_a i_3 = 0$$

$$-r_3 i_1 - r_a i_2 + (r_2 + r_3 + r_a) i_3 = 0$$

设 $r_s = r_1 = r_2 = r_x = r_a = 100\Omega$, $v_0 = 200V$, 试用
写程序计算电流.

矩阵求逆

- 为了求解矩阵的逆，同样可以将其转化为方程组的求解，即

$$\mathbf{A}\mathbf{x}_j = \mathbf{b}_j$$

其中 $\mathbf{b}_j = \delta_{ij}$

矩阵求逆

- 为了求解矩阵的逆，同样可以将其转化为方程组的求解，即

$$\mathbf{A}\mathbf{x}_j = \mathbf{b}_j$$

其中 $\mathbf{b}_j = \delta_{ij}$

- 这里我们简单的更改一下程序即可实现.

矩阵求逆

- 为了求解矩阵的逆，同样可以将其转化为方程组的求解，即

$$\mathbf{A}\mathbf{x}_j = \mathbf{b}_j$$

其中 $\mathbf{b}_j = \delta_{ij}$

- 这里我们简单的更改一下程序即可实现.
- 练习：试计算上一个例子中矩阵 A 的逆.

LU 分解

- 另一种求解矩阵问题的算法称为 LU 分解. 其主要将一个非奇异矩阵分解成一个下三角和一个上三角矩阵的乘积.

LU 分解

- 另一种求解矩阵问题的算法称为 LU 分解. 其主要将一个非奇异矩阵分解成一个下三角和一个上三角矩阵的乘积.
- 在之前的高斯消元法中, 也可以将消元的过程看作是

$$\mathbf{A}^{n-1} = \mathbf{M}\mathbf{A}$$

其中

$$\mathbf{M} = \mathbf{M}^{(n-1)} \mathbf{M}^{(n-2)} \dots \mathbf{M}^{(1)}$$

LU 分解

- 每一步的 M 矩阵为所有对角元为-1 以及 $M_{kij} = -A_{kj}^{(j-1)} / A_{jj}^{(j-1)}$, $i > j$. 因此上面的式子可以表示为

$$\mathbf{A} = \mathbf{L}\mathbf{U}$$

其中 $\mathbf{U} = \mathbf{A}^{(n-1)}$ 为上三角矩阵而 $\mathbf{L} = \mathbf{M}^{-1}$ 为下三角矩阵.

LU 分解

- 每一步的 M 矩阵为所有对角元为-1 以及 $M_{kij} = -A_{kj}^{(j-1)} / A_{jj}^{(j-1)}$, $i > j$. 因此上面的式子可以表示为

$$\mathbf{A} = \mathbf{L}\mathbf{U}$$

其中 $\mathbf{U} = \mathbf{A}^{(n-1)}$ 为上三角矩阵而 $\mathbf{L} = \mathbf{M}^{-1}$ 为下三角矩阵.

- 注意下三角矩阵的逆仍然是一个下三角矩阵.

LU 分解

- 因为 $M_{ii} = -1$, 故必有 $L_{ii} = 1$ 以及 $L_{ij} = -M_{ij} = A_{kij}^{(j-1)} / A_{kj}^{(j-1)}$, $i > j$.

LU 分解

- 因为 $M_{ii} = -1$, 故必有 $L_{ii} = 1$ 以及 $L_{ij} = -M_{ij} = A_{kij}^{(j-1)} / A_{kj}^{(j-1)}$, $i > j$.
- 这里取 $L_{ii} = 1$ 的算法称为 Doolittle 因子化, 另外也可以取 $U_{ii} = 1$, 这种称为 Crout 因子化. 两种方法是等价的.

LU 分解

- 因为 $M_{ii} = -1$, 故必有 $L_{ii} = 1$ 以及 $L_{ij} = -M_{ij} = A_{kij}^{(j-1)} / A_{kj}^{(j-1)}$, $i > j$.
- 这里取 $L_{ii} = 1$ 的算法称为 Doolittle 因子化, 另外也可以取 $U_{ii} = 1$, 这种称为 Crout 因子化. 两种方法是等价的.
- 通常 **L** 和 **U** 的元素可以通过比较它们的乘积与 **A** 中元素的比较得出.

$$L_{ij} = \frac{1}{U_{jj}} \left(A_{ij} - \sum_{k=1}^{j-1} L_{ik} U_{kj} \right)$$

$$U_{ij} = \frac{1}{L_{ii}} \left(A_{ij} - \sum_{k=1}^{i-1} L_{ik} U_{kj} \right)$$

LU 分解

- 其中起始值取为 $L_{i1} = A_{i1}/U_{11}$, $U_{1j} = A_{1j}/L_{11}$. 在该公式中因为需要除以对角元, 故仍需要选主元.

LU 分解

- 其中起始值取为 $L_{i1} = A_{i1}/U_{11}$, $U_{1j} = A_{1j}/L_{11}$. 在该公式中因为需要除以对角元, 故仍需要选主元.
- 我们可以将 **L**, **U** 的结果存在一个矩阵当中, 对角元放不为 1 的三角矩阵的值.

LU 分解

- 其中起始值取为 $L_{i1} = A_{i1}/U_{11}$, $U_{1j} = A_{1j}/L_{11}$. 在该公式中因为需要除以对角元, 故仍需要选主元.
- 我们可以将 \mathbf{L}, \mathbf{U} 的结果存在一个矩阵当中, 对角元放不为 1 的三角矩阵的值.
- 而矩阵 \mathbf{A} 的行列式为

$$|\mathbf{A}| = |\mathbf{L}| |\mathbf{U}| = \prod_{i=1}^n L_{ii} U_{ii}$$

LU 分解

- 而对于线性方程组，可以通过向前和向后替代求解：

$$\mathbf{L}\mathbf{y} = \mathbf{b}$$

$$\mathbf{U}\mathbf{x} = \mathbf{y}$$

LU 分解

- 而对于线性方程组，可以通过向前和向后替代求解：

$$\mathbf{Ly} = \mathbf{b}$$

$$\mathbf{Ux} = \mathbf{y}$$

- 从而有

$$y_i = \frac{1}{L_{ii}} \left(b_i - \sum_{k=1}^{i-1} L_{ik} y_k \right)$$

$$x_i = \frac{1}{U_{ii}} \left(y_i - \sum_{k=i+1}^n U_{ik} x_k \right)$$

LU 分解

例 2 试用 LU 分解法完成三次样条插值.

第八节 矩阵的数值方法

8.1 物理学中的矩阵

8.2 基本的矩阵操作

8.3 线性方程系统

8.4 多变量函数的根和极值

8.5 本征值问题

多变量函数的根和极值

- 然而物理上还会经常碰到一些非线性的问题，如求解一组非线性多变量方程组的解或寻找一个多变量函数的极大或极小值.

多变量函数的根和极值

- 然而物理上还会经常碰到一些非线性的问题，如求解一组非线性多变量方程组的解或寻找一个多变量函数的极大或极小值.
- 这些问题可以将其分解成每一步都成为一些线性问题.

多变量函数的根和极值

- 然而物理上还会经常碰到一些非线性的问题，如求解一组非线性多变量方程组的解或寻找一个多变量函数的极大或极小值。
- 这些问题可以将其分解成每一步都成为一些线性问题。
- 首先来看如何运用前面介绍的矩阵方法来求解这些问题。

多维牛顿法

- 在之前我们介绍了用牛顿法求解单变量函数的根.

多维牛顿法

- 在之前我们介绍了用牛顿法求解单变量函数的根.
- 现在我们扩张到求解一组多变量函数的根. 如

$$\mathbf{f}(\mathbf{x}) = 0$$

多维牛顿法

- 在之前我们介绍了用牛顿法求解单变量函数的根.
- 现在我们扩张到求解一组多变量函数的根. 如

$$\mathbf{f}(\mathbf{x}) = 0$$

- 其中 $\mathbf{f} = (f_1, \dots, f_n), \mathbf{x} = (x_1, \dots, x_n)$.

多维牛顿法

■ 同样的，我们在解 \mathbf{x}_r 附近的 \mathbf{x} 点做泰勒展开

$$\mathbf{f}(\mathbf{x}_r) = \mathbf{f}(\mathbf{x}) + \Delta\mathbf{x} \cdot \nabla\mathbf{f}(\mathbf{x}) + O(\Delta\mathbf{x}^2) \simeq \mathbf{0}$$

多维牛顿法

- 同样的，我们在解 \mathbf{x}_r 附近的 \mathbf{x} 点做泰勒展开

$$\mathbf{f}(\mathbf{x}_r) = \mathbf{f}(\mathbf{x}) + \Delta\mathbf{x} \cdot \nabla\mathbf{f}(\mathbf{x}) + O(\Delta\mathbf{x}^2) \simeq \mathbf{0}$$

- 该线性方程可以写成矩阵的形式

$$\mathbf{A}\Delta\mathbf{x} = \mathbf{b}$$

多维牛顿法

- 同样的，我们在解 \mathbf{x}_r 附近的 \mathbf{x} 点做泰勒展开

$$\mathbf{f}(\mathbf{x}_r) = \mathbf{f}(\mathbf{x}) + \Delta\mathbf{x} \cdot \nabla\mathbf{f}(\mathbf{x}) + O(\Delta\mathbf{x}^2) \simeq \mathbf{0}$$

- 该线性方程可以写成矩阵的形式

$$\mathbf{A}\Delta\mathbf{x} = \mathbf{b}$$

- 其中 $A_{ij} = \frac{\partial f_i(\mathbf{x})}{\partial x_j}, b_i = -f_i.$

多维牛顿法

- 因此我们可以通过每次按照上面的方式进行迭代来求解非线性方程的解。迭代为

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta \mathbf{x}_k$$

多维牛顿法

- 因此我们可以通过每次按照上面的方式进行迭代来求解非线性方程的解. 迭代为

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta \mathbf{x}_k$$

- 其中 $\Delta \mathbf{x}_k$ 为上面线性方程的解.

多维牛顿法

- 因此我们可以通过每次按照上面的方式进行迭代来求解非线性方程的解. 迭代为

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta \mathbf{x}_k$$

- 其中 $\Delta \mathbf{x}_k$ 为上面线性方程的解.
- 而之前 1 维的方法则是上面方法中 $n = 1$ 的特例.

多维牛顿法

- 类似的，如果我们不能的得到 \mathbf{f} 的一阶导数的具体表达式，则可以采用割线法

$$A_{ij} = \frac{f_i(\mathbf{x} + h_j \hat{\mathbf{x}}_j) - f_i(\mathbf{x})}{h_j}$$

多维牛顿法

- 类似的，如果我们不能的得到 \mathbf{f} 的一阶导数的具体表达式，则可以采用割线法

$$A_{ij} = \frac{f_i(\mathbf{x} + h_j \hat{\mathbf{x}}_j) - f_i(\mathbf{x})}{h_j}$$

- 其中 h_j 为第 j 个方向上的步长，一个常见选择的方式是取

$$h_j \simeq \delta_0 x_j$$

其中 δ_0 为浮点数的公差的平方根.

多维牛顿法

- 类似的，如果我们不能的得到 \mathbf{f} 的一阶导数的具体表达式，则可以采用割线法

$$A_{ij} = \frac{f_i(\mathbf{x} + h_j \hat{\mathbf{x}}_j) - f_i(\mathbf{x})}{h_j}$$

- 其中 h_j 为第 j 个方向上的步长，一个常见选择的方式是取

$$h_j \simeq \delta_0 x_j$$

其中 δ_0 为浮点数的公差的平方根.

- 如果函数足够光滑，牛顿法和割线法都可以局域的收敛.

多维牛顿法

例3 试写出牛顿法和割线法的代码求解下面的两个方程

$$f_1(x, y) = e^{x^2} \ln y - x^2 = 0$$

$$f_2(x, y) = e^{y^2} \ln x - y^2 = 0$$

初始点在 $x = y = 1.5$ 附近.

多维函数的极值

- 在有了求解多变量函数方程组的解法后，可以很方便地推广到多变量函数求极值。一个多变量函数 $g(\mathbf{x})$ 的极值为

$$\mathbf{f}(\mathbf{x}) = \nabla g(\mathbf{x}) = 0$$

多维函数的极值

- 在有了求解多变量函数方程组的解法后，可以很方便地推广到多变量函数求极值。一个多变量函数 $g(\mathbf{x})$ 的极值为

$$\mathbf{f}(\mathbf{x}) = \nabla g(\mathbf{x}) = 0$$

- 这里 $\mathbf{f}(\mathbf{x})$ 是一个 n 维的矢量函数。

多维函数的极值

■ 因此问题转化为方程求根的问题. 在我们的一维求极值的问题当中, 我们每次 Δx 其方向与 $g'(x)$ 的方向相反. 类似的, 对于多维问题, 我们可以采用如下的方式给出 $\Delta \mathbf{x}$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta \mathbf{x}_k = \mathbf{x}_k - a \nabla g(\mathbf{x}_k) / |\nabla g(\mathbf{x}_k)|$$

多维函数的极值

- 因此问题转化为方程求根的问题. 在我们的一维求极值的问题当中, 我们每次 Δx 其方向与 $g'(x)$ 的方向相反. 类似的, 对于多维问题, 我们可以采用如下方式给出 $\Delta \mathbf{x}$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta \mathbf{x}_k = \mathbf{x}_k - a \nabla g(\mathbf{x}_k) / |\nabla g(\mathbf{x}_k)|$$

- a 为一个正的, 小的可调参数. 这样是的 \mathbf{x} 的方向沿着每一点的负梯度方向改变 (最陡), 该方法称为最陡下降法.

多维函数的极值

■ 为了求解多维函数的极值，将矩阵改写成

$$A_{ij} = \frac{\partial f_i}{\partial x_j} + \mu \delta_{ij}$$

多维函数的极值

- 为了求解多维函数的极值，将矩阵改写成

$$A_{ij} = \frac{\partial f_i}{\partial x_j} + \mu \delta_{ij}$$

- μ 为一个小的正值来保证 A 是正定的。这样可以保证每次迭代的方向向 $g(x)$ 减小的方向。

多维函数的极值

■ BFGS 是一种用来选择 μ 的方法，即取

$$\mathbf{A}_k = \mathbf{A}_{k-1} + \frac{\mathbf{y}\mathbf{y}^T}{\mathbf{y}^T \mathbf{w}} - \frac{\mathbf{A}_{k-1} \mathbf{w} \mathbf{w}^T \mathbf{A}_{k-1}}{\mathbf{w}^T \mathbf{A}_{k-1} \mathbf{w}}$$

多维函数的极值

- BFGS 是一种用来选择 μ 的方法，即取

$$\mathbf{A}_k = \mathbf{A}_{k-1} + \frac{\mathbf{y}\mathbf{y}^T}{\mathbf{y}^T \mathbf{w}} - \frac{\mathbf{A}_{k-1} \mathbf{w} \mathbf{w}^T \mathbf{A}_{k-1}}{\mathbf{w}^T \mathbf{A}_{k-1} \mathbf{w}}$$

- 其中

$$\mathbf{w} = \mathbf{x}_k - \mathbf{x}_{k-1}$$

$$\mathbf{y} = \mathbf{f}_k - \mathbf{f}_{k-1}$$

多维函数的极值

- BFGS 是一种用来选择 μ 的方法，即取

$$\mathbf{A}_k = \mathbf{A}_{k-1} + \frac{\mathbf{y}\mathbf{y}^T}{\mathbf{y}^T \mathbf{w}} - \frac{\mathbf{A}_{k-1} \mathbf{w} \mathbf{w}^T \mathbf{A}_{k-1}}{\mathbf{w}^T \mathbf{A}_{k-1} \mathbf{w}}$$

- 其中

$$\mathbf{w} = \mathbf{x}_k - \mathbf{x}_{k-1}$$

$$\mathbf{y} = \mathbf{f}_k - \mathbf{f}_{k-1}$$

- 该算法在许多实际问题当中取得了成功，但是为啥这么有效还不太清楚... 目前寻找多维函数的极值仍然是一个正在不断深入研究的事情。

多电荷团簇的几何结构

- 在研究原子、离子和分子的小团簇中，多电荷团簇的稳定几何结构非常重要。

多电荷团簇的几何结构

- 在研究原子、离子和分子的小团簇中，多电荷团簇的稳定几何结构非常重要。
- 假设我们研究 $(Na^+)_l(Cl^-)_m$ 的小团簇。其中一共包含 $n = l + m$ 个电荷粒子。我们取晶体的两个离子间的相互作用势为

$$V(r_{ij}) = \eta_{ij} \frac{e^2}{4\pi\epsilon_0 r_{ij}} + \delta_{ij} V_0 e^{-r_{ij}/r_0}$$

多电荷团簇的几何结构

- 在研究原子、离子和分子的小团簇中，多电荷团簇的稳定几何结构非常重要。
- 假设我们研究 $(Na^+)_l(Cl^-)_m$ 的小团簇。其中一共包含 $n = l + m$ 个电荷粒子。我们取晶体的两个离子间的相互作用势为

$$V(r_{ij}) = \eta_{ij} \frac{e^2}{4\pi\epsilon_0 r_{ij}} + \delta_{ij} V_0 e^{-r_{ij}/r_0}$$

- 如果两个离子带相反的电荷则 $\eta_{ij} = -1, \delta_{ij} = 1$ 。否则 $\eta_{ij} = 1$ 以及 $\delta_{ij} = 0$ 。 V_0 和 r_0 为经验参数。

多电荷团簇的几何结构

- 系统的总相互作用能量为

$$U(\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_n) = \sum_{i>j}^n V(r_{ij})$$

多电荷团簇的几何结构

- 系统的总相互作用能量为

$$U(\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_n) = \sum_{i>j}^n V(r_{ij})$$

- 当 U 达到一个局域最小值势即得到了该团簇的一个稳定结构. 虽然 U 看起来是 $3n$ 个坐标变量的函数, 考虑到系统的转动对称性, 可以只取 $3n-6$ 个坐标变量来进行模拟计算.

多电荷团簇的几何结构

- 系统的总相互作用能量为

$$U(\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_n) = \sum_{i>j}^n V(r_{ij})$$

- 当 U 达到一个局域最小值势即得到了该团簇的一个稳定结构. 虽然 U 看起来是 $3n$ 个坐标变量的函数, 考虑到系统的转动对称性, 可以只取 $3n-6$ 个坐标变量来进行模拟计算.
- 这个模拟程序就留给感兴趣的同学吧...

第八节 矩阵的数值方法

8.1 物理学中的矩阵

8.2 基本的矩阵操作

8.3 线性方程系统

8.4 多变量函数的根和极值

8.5 本征值问题

本征值问题

- 在物理学中矩阵的本征值问题非常重要. 定义为

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$$

其中 λ 即为矩阵 A 关于本征矢量 x 的本征值.

本征值问题

- 在物理学中矩阵的本征值问题非常重要. 定义为

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$$

其中 λ 即为矩阵 A 关于本征矢量 x 的本征值.

- 其由久期方程所决定

$$|\mathbf{A} - \lambda\mathbf{I}| = 0$$

本征值问题

- 在物理学中矩阵的本征值问题非常重要. 定义为

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$$

其中 λ 即为矩阵 A 关于本征矢量 x 的本征值.

- 其由久期方程所决定

$$|\mathbf{A} - \lambda\mathbf{I}| = 0$$

- 如果两个及以上的本征矢量对应相同的本征值, 则称它们是简并的.

厄米矩阵的本征值

- 由于各类物理问题在数值计算中经常会转化成矩阵问题. 故矩阵的本征值问题在各个方面都有广泛的应用.

厄米矩阵的本征值

- 由于各类物理问题在数值计算中经常会转化成矩阵问题. 故矩阵的本征值问题在各个方面都有广泛的应用.
- 在一些物理问题中, 面对的矩阵属于厄米矩阵

$$\mathbf{A}^\dagger = \mathbf{A}$$

厄米矩阵的本征值

- 因为厄米矩阵具有一些良好的性质，使得其特征值的计算相对简单：

厄米矩阵的本征值

- 因为厄米矩阵具有一些良好的性质，使得其特征值的计算相对简单：
 - 厄米矩阵所有本征值都是实数

厄米矩阵的本征值

- 因为厄米矩阵具有一些良好的性质，使得其特征值的计算相对简单：
 - 厄米矩阵所有本征值都是实数
 - 厄米矩阵的本征矢可以选择为相互正交

厄米矩阵的本征值

- 因为厄米矩阵具有一些良好的性质，使得其特征值的计算相对简单：
 - 厄米矩阵所有本征值都是实数
 - 厄米矩阵的本征矢可以选择为相互正交
 - 厄米矩阵经过由它所有本征矢组成的么正矩阵进行相似变换，得到一个对角矩阵，其对角元为它的本征值。

厄米矩阵的本征值

- 对于 $n \times n$ 的复厄米矩阵其本征值问题等价于一个 $2n \times 2n$ 的实对称矩阵. 对一个复矩阵可以写为

$$\mathbf{A} = \mathbf{B} + i\mathbf{C}$$

其中 \mathbf{B} 和 \mathbf{C} 为 \mathbf{A} 矩阵的实部和虚部.

厄米矩阵的本征值

- 对于 $n \times n$ 的复厄米矩阵其本征值问题等价于一个 $2n \times 2n$ 的实对称矩阵. 对一个复矩阵可以写为

$$\mathbf{A} = \mathbf{B} + i\mathbf{C}$$

其中 \mathbf{B} 和 \mathbf{C} 为 \mathbf{A} 矩阵的实部和虚部.

- 如果 \mathbf{A} 矩阵是厄米矩阵, 则 \mathbf{B} 是一个实对称矩阵, \mathbf{C} 是一个实反对称矩阵. 即

$$B_{ij} = B_{ji}$$

$$C_{ij} = -C_{ji}$$

厄米矩阵的本征值

- 如果类似的将本征矢量 \mathbf{z} 分解为 $\mathbf{z} = \mathbf{x} + i\mathbf{y}$. 则问题转化为

$$(\mathbf{B} + i\mathbf{C})(\mathbf{x} + i\mathbf{y}) = \lambda(\mathbf{x} + i\mathbf{y})$$

厄米矩阵的本征值

- 如果类似的将本征矢量 \mathbf{z} 分解为 $\mathbf{z} = \mathbf{x} + i\mathbf{y}$. 则问题转化为

$$(\mathbf{B} + i\mathbf{C})(\mathbf{x} + i\mathbf{y}) = \lambda(\mathbf{x} + i\mathbf{y})$$

- 矩阵形式为

$$\begin{pmatrix} \mathbf{B} & -\mathbf{C} \\ \mathbf{C} & \mathbf{B} \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} = \lambda \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix}$$

厄米矩阵的本征值

- 如果类似的将本征矢量 \mathbf{z} 分解为 $\mathbf{z} = \mathbf{x} + i\mathbf{y}$. 则问题转化为

$$(\mathbf{B} + i\mathbf{C})(\mathbf{x} + i\mathbf{y}) = \lambda(\mathbf{x} + i\mathbf{y})$$

- 矩阵形式为

$$\begin{pmatrix} \mathbf{B} & -\mathbf{C} \\ \mathbf{C} & \mathbf{B} \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} = \lambda \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix}$$

- 即化为了实对称矩阵的本征值问题.

厄米矩阵的本征值

- 将实对称矩阵变换成对角矩阵的变换矩阵此时成为了实正交矩阵.

厄米矩阵的本征值

- 将实对称矩阵变换成对角矩阵的变换矩阵此时成为了实正交矩阵.
- 可以将步骤分为两步

厄米矩阵的本征值

- 将实对称矩阵变换成对角矩阵的变换矩阵此时成为了实正交矩阵.
- 可以将步骤分为两步
 - 首先采用一个实正交矩阵将实对称矩阵进行相似变换, 变换成实三对角矩阵.

厄米矩阵的本征值

- 将实对称矩阵变换成对角矩阵的变换矩阵此时成为了实正交矩阵.
- 可以将步骤分为两步
 - 首先采用一个实正交矩阵将实对称矩阵进行相似变换, 变换成实三对角矩阵.
 - 求解实三对角矩阵的本征值问题.

厄米矩阵的本征值

- 将实对称矩阵变换成对角矩阵的变换矩阵此时成为了实正交矩阵.
- 可以将步骤分为两步
 - 首先采用一个实正交矩阵将实对称矩阵进行相似变换, 变换成实三对角矩阵.
 - 求解实三对角矩阵的本征值问题.
- 常用来将一个实对称矩阵化为实三对角矩阵的算法为 Householder 算法.

厄米矩阵的本征值

- 该算法通过总共 $n - 2$ 个连续变换来实现三对角化. 每次变换操作矩阵的一个行和一列. 可表示为递归操作

$$\mathbf{A}^{(k)} = \mathbf{O}_k^T \mathbf{A}^{(k-1)} \mathbf{O}_k$$

厄米矩阵的本征值

- 该算法通过总共 $n - 2$ 个连续变换来实现三对角化. 每次变换操作矩阵的一个行和一列. 可表示为递归操作

$$\mathbf{A}^{(k)} = \mathbf{O}_k^T \mathbf{A}^{(k-1)} \mathbf{O}_k$$

- \mathbf{O}_k 是一个正交矩阵, 它作用在第 k 列的第 $i = k + 2, \dots, n$ 行以及第 k 行的第 $j = k + 2, \dots, n$ 列的元素上.

厄米矩阵的本征值

■ \mathbf{O}_k 可以表示为

$$\mathbf{O}_k = \mathbf{I} - \frac{1}{\eta_k} \mathbf{w}_k \mathbf{w}_k^T$$

厄米矩阵的本征值

- \mathbf{O}_k 可以表示为

$$\mathbf{O}_k = \mathbf{I} - \frac{1}{\eta_k} \mathbf{w}_k \mathbf{w}_k^T$$

- 矢量 \mathbf{w}_k 的第 l 分量为

$$w_{kl} = \begin{cases} 0 & \text{for } l \leq k \\ A_{k+1}^{(k-1)} + \alpha_k & \text{for } l = k+1 \\ A_{kl}^{(k-1)} & \text{for } l \geq k+2 \end{cases}$$

厄米矩阵的本征值

■ \mathbf{O}_k 可以表示为

$$\mathbf{O}_k = \mathbf{I} - \frac{1}{\eta_k} \mathbf{w}_k \mathbf{w}_k^T$$

■ 矢量 \mathbf{w}_k 的第 l 分量为

$$w_{kl} = \begin{cases} 0 & \text{for } l \leq k \\ A_{k+1}^{(k-1)} + \alpha_k & \text{for } l = k+1 \\ A_{kl}^{(k-1)} & \text{for } l \geq k+2 \end{cases}$$

■ 其中 $\alpha_k = \pm \sqrt{\sum_{l=k+1}^n [A_{kl}^{(k-1)}]^2}$, $\eta_k = \alpha_k [\alpha_k + A_{kk+1}^{(k-1)}$

厄米矩阵的本征值

- 实际中 α_k 的符号选取与 $A_{kk+1}^{(k-1)}$ 的符号相同，避免 η_k 可能出现发散.

厄米矩阵的本征值

- 实际中 α_k 的符号选取与 $A_{kk+1}^{(k-1)}$ 的符号相同，避免 η_k 可能出现发散。
- 从递归公式可以发现，这样定义的 \mathbf{O}_k 确实可以将 $\mathbf{A}^{(k-1)}$ 的第 k 列的第 $i = k + 2, \dots, n$ 行以及第 k 行的第 $j = k + 2, \dots, n$ 列的元素变为零。

厄米矩阵的本征值

- 实际中 α_k 的符号选取与 $A_{kk+1}^{(k-1)}$ 的符号相同，避免 η_k 可能出现发散.
- 从递归公式可以发现，这样定义的 \mathbf{O}_k 确实可以将 $\mathbf{A}^{(k-1)}$ 的第 k 列的第 $i = k + 2, \dots, n$ 行以及第 k 行的第 $j = k + 2, \dots, n$ 列的元素变为零.
- 程序见 libsec5.c

厄米矩阵的本征值

- 在得到三对角矩阵后，本征值可以采用求根的算法给出。对于久期方程 $\mathbf{A} - \lambda \mathbf{I} = 0$ ，等价于多项式方程 $p_n(\lambda) = 0$.

厄米矩阵的本征值

- 在得到三对角矩阵后，本征值可以采用求根的算法给出。对于久期方程 $\mathbf{A} - \lambda \mathbf{I} = 0$ ，等价于多项式方程 $p_n(\lambda) = 0$.
- 由于三对角矩阵的对称性，方程可以利用递归求解

$$p_i(\lambda) = (a_i - \lambda)p_{i-1}(\lambda) - b_{i-1}^2 p_{i-2}(\lambda)$$

其中 $a_i = A_{ii}, b_i = A_{i,i+1} = A_{i+1,i}$. 起始值取为 $p_0(\lambda) = 1, p_1(\lambda) = a_1 - \lambda$.

厄米矩阵的本征值

- 在得到三对角矩阵后，本征值可以采用求根的算法给出。对于久期方程 $\mathbf{A} - \lambda \mathbf{I} = 0$ ，等价于多项式方程 $p_n(\lambda) = 0$.
- 由于三对角矩阵的对称性，方程可以利用递归求解

$$p_i(\lambda) = (a_i - \lambda)p_{i-1}(\lambda) - b_{i-1}^2 p_{i-2}(\lambda)$$

其中 $a_i = A_{ii}, b_i = A_{i,i+1} = A_{i+1,i}$. 起始值取为 $p_0(\lambda) = 1, p_1(\lambda) = a_1 - \lambda$.

- 因此可以利用递归法给出 $p_n(\lambda)$ 的表示.

厄米矩阵的本征值

- 在得到 $p_n(\lambda)$ 的递归表达式之后，原则上可以采用任意的求根法来进行求解.

厄米矩阵的本征值

- 在得到 $p_n(\lambda)$ 的递归表达式之后，原则上可以采用任意的求根法来进行求解.
- 由于 $p_n(\lambda)$ 零点的分布具有特别的规律：

厄米矩阵的本征值

- 在得到 $p_n(\lambda)$ 的递归表达式之后，原则上可以采用任意的求根法来进行求解.
- 由于 $p_n(\lambda)$ 零点的分布具有特别的规律：
 - $p_n(\lambda) = 0$ 的根在区域 $[-\|A\|, \|A\|]$ 之间，其中 $\|A\|$ 为 A 的列矢量最大模，即

$$\|A\| = \max \left\{ \sum_{j=1}^n |A_{ij}| \right\}$$

厄米矩阵的本征值

- 在得到 $p_n(\lambda)$ 的递归表达式之后，原则上可以采用任意的求根法来进行求解.
- 由于 $p_n(\lambda)$ 零点的分布具有特别的规律：
 - $p_n(\lambda) = 0$ 的根在区域 $[-\|A\|, \|A\|]$ 之间，其中 $\|A\|$ 为 A 的列矢量最大模，即

$$\|A\| = \max \left\{ \sum_{j=1}^n |A_{ij}| \right\}$$

- 对于 $\lambda > \lambda_0$ 时，方程 $p_n(\lambda) = 0$ 的根的数目和 $p_j(\lambda_0)$ 与 $p_{j-1}(\lambda_0)$ 具有相同符号的数目相同.

厄米矩阵的本征值

- 根据这些性质，我们可以得到一个简单又快速的求根方法.

厄米矩阵的本征值

- 根据这些性质，我们可以得到一个简单又快速的求根方法.
- 首先计算出 $\|A\|$, 即得到了 λ 的边界. 由于这里有 n 个本征值，我们需要确定寻找哪一个. 例如基态本征值.

厄米矩阵的本征值

- 根据这些性质，我们可以得到一个简单又快速的求根方法.
- 首先计算出 $||A||$, 即得到了 λ 的边界. 由于这里有 n 个本征值，我们需要确定寻找哪一个. 例如基态本征值.
- 首先对区间进行二分，并计算所有的 $p_i(0)$ 的符号. 这样我们就可以知道分别在 $[-||A||, 0]$ 和 $[0, ||A||]$ 区间中的根的数目. 注意这里包含了简并.

厄米矩阵的本征值

- 进一步在将区间划分为四个，并计算出在划分点多项式的符号.

厄米矩阵的本征值

- 进一步在将区间划分为四个，并计算出在划分点多项式的符号.
- 该程序可以一直进行，直到每个小的区间只包含一个本征值.

厄米矩阵的本征值

- 进一步在将区间划分为四个，并计算出在划分点多项式的符号.
- 该程序可以一直进行，直到每个小的区间只包含一个本征值.
- 当程序进行了 l 次后，则每个解的区间长度为 $||A||/2^{l-1}$.

厄米矩阵的本征值

- 进一步在将区间划分为四个，并计算出在划分点多项式的符号.
- 该程序可以一直进行，直到每个小的区间只包含一个本征值.
- 当程序进行了 l 次后，则每个解的区间长度为 $\|A\|/2^{l-1}$.
- 因此为了求解一个实对称矩阵，可以分为两个程序，一个是将矩阵化为三对角的形式，第二个则求解三对角矩阵的本征值.

普通矩阵的本征值

- 虽然大部分物理上碰到的矩阵是厄米矩阵，但仍然可能会碰到一般矩阵的本征值求解。

普通矩阵的本征值

- 虽然大部分物理上碰到的矩阵是厄米矩阵，但仍然可能会碰到一般矩阵的本征值求解.
- 这里我们主要讨论非退化矩阵的求解. 非退化意味着该矩阵可以通过相似变换变成对角矩阵，且本征矢构成一个完备的矢量空间.

普通矩阵的本征值

- 虽然大部分物理上碰到的矩阵是厄米矩阵，但仍然可能会碰到一般矩阵的本征值求解.
- 这里我们主要讨论非退化矩阵的求解. 非退化意味着该矩阵可以通过相似变换变成对角矩阵，且本征矢构成一个完备的矢量空间.
- 通常我们可以定义一个类似于函数去定义一个矩阵函数，例如：

$$f(\mathbf{A}) = e^{-\alpha \mathbf{A}} = \sum_{k=0}^{\infty} \frac{(-\alpha \mathbf{A})^k}{k!}$$

普通矩阵的本征值

- 当一个矩阵函数作用到该矩阵的本征矢，矩阵可以替换为其本征值

$$f(\mathbf{A})\mathbf{x}_i = f(\lambda_i)\mathbf{x}_i$$

普通矩阵的本征值

- 当一个矩阵函数作用到该矩阵的本征矢，矩阵可以替换为其本征值

$$f(\mathbf{A})\mathbf{x}_i = f(\lambda_i) \mathbf{x}_i$$

- 如果函数包含矩阵的逆并且正好本征值为 0，可以在原始的矩阵中加入 $\eta \mathbf{I}$, 从而移除奇异性，在最后再令 $\eta \rightarrow 0$.