计算物理学

2017-2018 秋季学期 · 北京大学物理学院

第一次大作业 · **文档, 数据与源代码**

林海芃, 1500011415

2017 年 10 月

目录

"Good design adds value faster than it adds cost."

- Thomas C. Gale

前言

在开始本次 “计算物理学” 的大作业文档之前, 笔者希望在此简要做一下本次作业解答的总览, 一方面梳理自己的思路, 另一方面也希望为阅卷者提供一些便利.

本次作业我选择了 Python 编写. 选择 Python 着实是无奈之举; 其并不是一个很适宜数值计算的语言, 其中诸多的为编程初学者予以便利的设计很容易导致编程习惯的歪曲, 同时其相比 Fortran, C++ 等语言距离硬件更远, 很难进行低层次的优化, 性能难以得到保障. 然而鉴于作业的 “全自创” 性要求, Python 无需额外的库所带的功能性超过了前两种语言, 为了更大程度上的聚焦于算法而不是琐碎的编程工具, 不得不选用 Python 这个捷径.

说到 “全自创性”, 笔者很高兴地宣布**本次大作业中的所有代码不依赖于任何一行 Python 的外界库**, 不需要 import 任何额外的库 (除了我自己编写的矩阵库以外), 就可以实行其中所有的计算. 当然, 为了阅卷者和调试者的便利, 如果提供了优秀的 matplotlib 绘图库, 使用者可以对许多可以绘图的类型进行绘图, 例子的代码也以注释的形式给出. 一些 Python 语言的高级功能, 例如 functools, 也可以被 import, 为以后使用我所编写的数值公式的人提供调用的便利. **但所有的核心计算功能不依赖于任何非本人所编写的库. 从基本的阶乘, 三角函数, 到矩阵数据结构、向量操作, 到题目中要求的算法, 所有的代码是完全独立的**.

本代码的另一特点是进行了一定程度上的封装. **所有的算法代码作为独立的子模块进行抽象编写, 不依赖于题目中给定的具体数据、函数、结构,** 从而**可以直接应用于其他的科学运算** (当然其实用 numpy 更好.) 较为突出的例子包括 (1) 题, 我编写的函数名称叫 *ieee2dec, dec2ieee*, 它是支持任意 IEEE 浮点数系的, 从我们熟知的 (single) float, 到题目要求的 double, 到 quadruple, octuple, 只要提供了相应的参数, 代码可以支持任意精度的扩展; (2) 题中的欧拉法求解常微分方程 (eulerForwardSolve) 接受 lambda 函数作为输入; (4) 题中的勒让德函数 (LegendrePolyNA)、连带勒让德函数 (LegendrePolyA) 是可以独立于球谐函数 (SphericalHarmonics) 而调用的. (6) 题中的矩阵库可以独立调用, 共轭梯度法 (conjGradient) 可以求解任意给定的 (满足收敛条件) 的矩阵, 题目中的矩阵以生成函数的形式给定, 等等… **编写不依赖于题目具体的问题的抽象算法代码是我完成本次作业的核心**.

代码的编写离不开注释. 本次作业的编写过程, 心得和许多算法的描述都在注释中得以了集中的体现. 笔者将题目的源代码附在了本文档的每一道题的后面, **因为代码本身就是最好的文档**. Self-documenting code 和优秀的注释是组成文档的一部分, **希望阅卷者在参阅文档的同时, 配合文档的注释进行阅读**, 希望可以令阅卷者满意.

最后还有必要声明, 本作业中我的代码有可能包含很多 “并不 Python” 的用法. *Lambda* 函数, 尾递归 (Tail Recursion), 类似 *Folding* 的许多数列折叠用法, 许多的 *For* 循环求和, 甚至于我编写的矩阵库 *pylitematrix* 中的 “无副作用” 要求, 提供的 class prototype (对象原型) 等, 都是在 Python 界被认为非常 “Unpythonic” 的. **如果读者是一个 Python 拥护者, 请原谅我这一点**; 笔者的主要工作语言是 *Haskell*, 因而函数式编程的思想过于深入人心 (fold, zip, parrying, lambda 基本上就是 Haskell 的生命.). 笔者深知 Python 对这些功能的优化做得并不到位, 甚至说可以是很糟糕, 而笔者对于这些不优化的手动优化甚至可能更糟糕 (参见(4)代码中过分使用的 caching). 这是笔者的一大遗憾, 希望能在之后的大作业中得到解决.

林海芃

2017 年 10 月 31 日夜于燕南园

计算物理学 “大作业” 文档 (一)

IEEE 双精度浮点数的表示与十进制的相互转化

【题目】

（一）【10分】试写两个程序，实现任意十进制数与其规范化的二进制双精度浮点数编码间的相互转换。

【编写思路】

十进制转二进制

这个题目显然可以直接利用编程语言的性质, 直接将数字转化为 IEEE Double Precision 编码:

```
#include <iostream>
#include <string.h>
#include <stdint.h>
using namespace std;

int main() {
    double number;
    uint64_t u;
    cin >> number;
    memcpy(&u, &number, sizeof(number));
    cout << hex << u << endl;
    return 0;
}
```

不过, 这样做有两个问题:

(1) 输出的是十六进制的编码;

(2) 这样做肯定是没有分数的.

因此, 我们还是要按照正常的途径将十进制数字转化为二进制编码. 为了深入地理解概念 (这才是作业的目的), 并且满足作业的要求, 我们需要对数字进行操作而逐步像课件一样 “构造” 出 IEEE Double 表示. 首先重温一下定义:

浮点数的表示:

$$x = \pm\left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \cdots + \frac{d_{p-1}}{\beta^{p-1}}\right) \times \beta^E$$

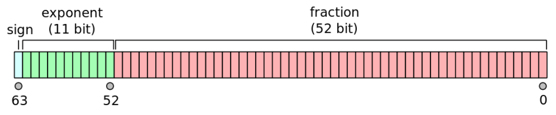
(其中, $d_i \in \mathbb{Z}, 0 \le d_i \le \beta - 1,\ i = 0, \dots, p-1, L \le E \le U$)

对于 IEEE Double Precision, 我们有 $\beta = 2, p = 53, L = -1022, U = 1023$, 所以有

$$x = \pm\left(d_0 + \frac{d_1}{2} + \frac{d_2}{2^2} + \cdots + \frac{d_{52}}{2^{52}}\right) \times 2^E$$

(其中, $d_i \in \mathbb{Z}, 0 \le d_i \le 1,\ i = 0, \dots, 52, -1022 \le E \le +1023$)

我们用 11 bit 储存 E 的长度 (注意到, $2^{11} = 2048$), 所以偏移值为 1023, 我们有 53 位有效数字.



IEEE Double Precision Bit Representation Layout (from Wikipedia)

IEEE 浮点数的规范化表示程序编写是可以独立于上述的参量选取的, 从而我们对于**任意 IEEE 规范浮点数系**进行程序的编写, 把 double 的情况作为一个特例. 设置的参数如下 (也可以参见源代码)

```
def dec2ieee(strN):
    # ~ * Configuration Parameters Here * ~
    # dec2ieee supports any (theoretically) IEEE-specified precision spec,
    # simply configure the parameters below to your liking.
    # Beta = 2 (fixed for binary, this you can't change, sorry)
    # Precision: Number of precision bits (can be ARBITRARY), d0 ~ d(p-1)
    P = 53
    # L (lower exponential bound), E >= -L
    expL = 1022
    # U (upper exponential bound), E <= +U
    expU = 1023
    # Exponential Bits
    E = 11
    # The one sign bit is contained in P.
    # All numbers generated are SIGNED (for UNSIGNED it is trivial to modify the below code.)
    # ~ * NO USER CONFIGURABLE VARIABLES BELOW * ~
```

如果我们希望达到任意大的精度, 并且编写不直接依赖于上述参量性质的代码, 我们需要利用 Python 自带的任意大精度整数计算的功能. 这个功能很强大, 我们可以依赖于它进行任意精度的整数计算, 从而**不受 Python 本身的双精度浮点型的限制**, 就可以用我们即将编写的代码进行, 比如说, IEEE *quadruple* 表示的转化.

本题编写的核心思想是通过字符串和利用 Python *Int* 类型具有任意精度的性质, 进行一个任意精度浮点数的计算 (针对于这个题目而言). 问题事实上分为两个部分, 一个是整数部分的转化, 一个是分数部分的转化. 当我们把源数据按照字符串的形式读入的时候, 我们自然可以进行整数-分数类型的转换:

```
components = strN.split('.')
if(len(components) > 2):
    raise ValueError("The decimal inputted has more than a whole and fractional component.")

# "Whole" component
# accept e.g. ".572" input.
whole = (int(components[0]) if len(components[0]) > 0 else 0)

# "Fractional" component
# accept e.g "17." "17" input.
frac  = (components[1] if len(components) == 2 and len(components[1]) > 0 else "0")
```

整数部分的处理很自然, 只需要从最大可能的 2-次数 (在 IEEE 浮点数表示中, 这就是 2^U) 往下进行除法就可以完成. 代码参见源代码.

分数部分的处理是比较有趣的. 如果我们直接依赖于 Python float 类型的精度, 则到最后接近 2^{-L} 的时候会发生严重精度不足的情况. 比如说, 我们知道

$$2^{-1073} \approx 9.22 \times 10^{-625} \qquad 2^{-1074} \approx 4.6 \times 10^{-625}$$

但是, 事实上在 Python 中计算的时候会有

```
>>> 2**(-1073)
1e-323
>>> 2**(-1074)
5e-324
```

所以显然这个方案是不可取的, 我们需要高精度的浮点数计算. 一种很自然的想法就是我们可以将 2 的负次幂的所有有效位数存储在一个整数类型中, 然后把更大的指数与其对齐. 这个原理可以在下面的一个简单的例子中展示出来:

```
Exp     Real Value        Integer Representation
--------------------------------------------------
2^-1  = 0.5               500000
2^-2  = 0.25              250000
2^-3  = 0.125             125000
2^-4  = 0.0625             62500
2^-5  = 0.03125            31250
2^-6  = 0.015625           15625
Largest Precision: Up to all significant digits of 2^-6
```

所以我们就是这样完成的. 通过 "借用" L + P – 1 个位数, 我们可以**计算出 2^-L 的所有有效位数**, 从而没有任何精度损失. 创造这个数据表的过程如下代码:

```
# 2**(-1074) is approximately 4.6e-625, 2**(-1073) is 9.22e-625
# We need to ensure precision of 2**(-1074). A naive try would be 1e626,
# but we need much more than that to prevent precision loss beyond the point
# of 2**-623. A experiment reveals we need to use 1e1074 as the starting seed,
# so we can get all the 751 significant digits of 2**(-1074)

# Starting from 2^(-1)...
# ## PRECOMPILATION OF INTEGER-CODED NEGATIVE TWO POWERS FOR IEEE ##
# For double, configure as follows: decPowerOfTenSeed = 1074 (1022+52),
# minNegTwoPtr = 1075. minNegTwoPtr can be arbitrarily large, but don't
# be wasteful.
decPowerOfTenSeed = expL + P - 1
minNegTwoPtr = expL + P + 1

# Build this table (does not take long)
negTwoIntResultsTable = [None]
negTwoPtr  = 2
negTwoCurr = int("5" + "0" * decPowerOfTenSeed)
```

```
negTwoIntResultsTable.append(negTwoCurr)
while(negTwoPtr != minNegTwoPtr):
    negTwoCurr = negTwoCurr // 2
    negTwoIntResultsTable.append(negTwoCurr)
    # print(negTwoPtr, ":", negTwoCurr) # diag only
    negTwoPtr += 1
```

这样是很巧妙的, 然后我们就只需要做整数减法, 和之前的 2 正次幂的原理类似, 就可以完成小数部分的转化. 注意最后会多出很多 0, 把字符串进行一下处理就可.

下一步就是把诸如 1101.101 的表达式化为 1.101101×2^3 的表达式. 这有两种情况, 一种是存在整数部分, 一种是不存在整数部分 (负指数). 这个部分只需要分开进行一下处理, 判断 wholeResult 是否为 "0" 字符串就可.

其他的计算基本上都和 IEEE 标准类似, 只需要处理最后一个问题: **因为本程序精度过高, 在进行精度截断的时候需要进行 "零舍一入".** 比如如果我们计算出的精确结果高于 52 位, 需要根据第 53 位是否为 1 的情况往上 "进" 一位:

```
# Generate the final IEEE double-precision representation,
# chopping off any non-significant bits (sorry precision!)
result = sign.__str__()
result = result + binExpOffset
if(len(irBinaryRepTail) < P - 1):
    result = result + irBinaryRepTail.ljust(P - 1, '0')
else:
    # Handle rounding errors... round up if necessary
    if(irBinaryRepTail[P-1] == "1"):
        lookAt = P-2
        while(irBinaryRepTail[P-2] == "1"):
            lookAt -= 1
        irBinaryRepTail = irBinaryRepTail[0:lookAt] + "1" + irBinaryRepTail[(lookAt+1):]

    result = result + irBinaryRepTail[0:(P-1)]
```

这里依然用到了很多字符串操作.

二进制转十进制

这个相对十进制转二进制稍微简单一些. 我们依然采用**对任意 IEEE 标准浮点数精度的支持**, 默认设置支持 Double, 满足本题目要求.

首先需要对于输入值进行分离. 这个就根据我们的设置可以进行框定 – 我们一共有 1 个符号位, E 个指数位, 剩下的都是有效位数. 分离并判断符号如下:

```
  signBit = strN[0]
if(signBit == "1"):
    # is negative
    result = result + "-"

expBits = strN[1:(E+1)]
valBits = strN[(E+1):]
```

指数偏移需要从 expBits 中的转化结果扣除. 知道了指数以后, 需要分情况讨论. 如果指数为非负, 则我们分离有效位数编码的前 (指数) 位, 并前面加上 1. 如果指数是负的则整数部分编码为 0, 小数部分编码加上 (|指数|-1) 位的 0, 补 1, 再附上所有有效位数编码. 换言之:

```
# Isolate ValBits through exponent.
# If exponent is non-negative, then simply
# 1.xxxxxxxx -> move . to the right expVal bits
wholeBits = ""
fractionBits = ""

if(expVal >= 0):
    wholeBits = "1" + valBits[:(expVal)]
    fractionBits = valBits[(expVal):]
else:
    wholeBits = "0"
    # If its, e.g. raw result
    # 0.0001100101... -> 1.100101... x 2^(-4) (base 2)
    # then the fractional result is 0001100101..
    # Using 4-1=3 padded left-zeroes, then a 1, then the given component
    # (100101...) encoded in the IEEE standard
    fractionBits = "0" * ((-1) * expVal - 1) + valBits
```

然后就可以进行逐步分离了. 通过上述的方法对小数进行高精度处理, 整数部分已经是全精度的了, 从而完成代码的编写.

【运行方式】

该程序分为两个文件. 如果需要进行十进制 → 二进制转化, 请运行 *ex1-double10.py*, 如果是二进制 → 十进制转化, 则运行 *ex1-2double.py*. 运行后会直接显示输入界面供用户使用.

【运行结果】

十进制转二进制和 C++ 标准比对多次数据结果无差异.

二进制转十进制因程序编写性质, 精度高于所有基于 C Double 输出的语言, 满足题目要求.

几个实例:

```
Decimal                     IEEE-Double
-------------------------------------------------------------------------------------------
+1.0                        0011111111110000000000000000000000000000000000000000000000000000
-1.0                        1011111111110000000000000000000000000000000000000000000000000000
+0.4                        0011111111011001100110011001100110011001100110011001100110011010
+12345678901234567          0100001101000101111011100010101000101110101101011010010111000100
+970215.970814              0100000100101101100110111100111111110001000011101000100001011001
+1.42857e-8                 0011111001001110101011011010010110001001100100101110010101110010
+1.42857e-21                00111011100110101111110000100100000000000011111011111010001100101
+5e177                      0110010011010011101111011111011111100000001101100000110000110110
```

注意因为 `ex1-2double.py` 的内部精度高于 `Double`，转化回的时候会有额外的位数出现．这和 `Python` 在直接输出数字的时候会携带有额外的数字有关，二进制浮点数系并不能精确表示所有的实数，并且也有精度的限制．

运行结果因为是直接进行一个既有标准的实现，不存在误差问题．注意程序内部精度高于题目所要求的精度．

【算法性能】

即时计算. 其中需要注意的是 2 的负次幂的所有有效位数计算生成的 table 需要一共 L 次除法和 O(n) (n=L) 的空间, 但在现代计算机中这个尺度是可以忽略的, 不因输入数据类型而改变.

【源代码】

十进制转二进制

本代码不依赖于任何外界库. 目前设置的目标数据类型是 Double, 如果需要更高/低精度需要根据 IEEE 标准调整.

```
########################################################################
# Computational Physics, 2017-18 Sem1
# HW-1 Ex-1
#
# (c) 2017 Haipeng Lin <linhaipeng@pku.edu.cn>
# All Rights Reserved.
#
# This program is written as Homework for the Computational Physics
# Course in Peking University, School of Physics.
# NO WARRANTIES, EXPRESS OR IMPLIED, ARE OFFERED FOR THIS PRODUCT.
# Copying is strictly prohibited and its usage is solely restricted
# to uses permitted by the author and homework grading.
#
# This program is Python 3 (3.6.2 on MSYS_NT)
# compatible, and does not support Python 2.
#
# Now Playing: Into Your Arms - The Maine
########################################################################

########################################################################
# Decimal to IEEE Any-Precision Floating Point Format
########################################################################
# string dec2ieee(string strN)
# Accepts n in "string" format, returning the IEEE any-precision result
# in a "string" format. (By default as an example we give double)
# The function is freely configurable.
def dec2ieee(strN):
    # ~ * Configuration Parameters Here * ~
    # dec2ieee supports any (theoretically) IEEE-specified precision spec,
    # simply configure the parameters below to your liking.
    #
    # Beta = 2 (fixed for binary, this you can't change, sorry)
    # Precision: Number of precision bits (can be ARBITRARY), d0 ~ d(p-1)
    P = 53
    # L (lower exponential bound), E >= -L
    expL = 1022
    # U (upper exponential bound), E <= +U
    expU = 1023
    # Exponential Bits
    E = 11
    # The one sign bit is contained in P.
    # All numbers generated are SIGNED (for UNSIGNED it is trivial to modify the below code.)

    # ~ * NO USER CONFIGURABLE VARIABLES BELOW * ~
    # TODO: Sanity check if strN is actually a number
    if(strN[0] == "-"):
        sign = 1
        strN = strN[1:]
    elif(strN[0] == "+"):
        sign = 0
        strN = strN[1:]
    else:
        sign = 0

    sh
    # For the "Whole" component, decompose it as a set of exponents
    # ranging from 2^1024 all the way down to 2^0 = 1
    # For 0, it is a special case and we'll handle it differently.
    # Store the "whole part" (pre-dot) binary result in the wholeResult string.
    wholeResult = ""
    if(whole == 0):
        wholeResult = "0"
    else:
        # Gradually write the result to wholeResult, from 2^1023 to 0
        i = expU
        while(i != -1):
            # print("* dec2double diag: [whole] whole = ", whole, " checking i =", i, flush=True)
            if(whole >= 2**i):
                whole -= 2**i
                wholeResult = wholeResult + "1"
            else:
                wholeResult = wholeResult + "0"
            i -= 1

    # Strip the trailing left-zeroes
    wholeResult = wholeResult.lstrip("0")
    if(wholeResult == ""):
        wholeResult = "0"

    # For the "Fractional" component, it is trickier as you have to
    # check for the *negative* exponents of 2, which may of course
    # underflow at any time, also at the cost of precision loss.
    #
    # In order not to be constrained by the Python's default "double"
    # precision (which results in increasingly lower precision as one
    # goes below and below), we had to devise a runtime compiled set of
    # "Integer" coded negative powers of 2 with a bunch of zeroes,
    # and compare the result with that instead,
    # after padding both the "frac" part and all these results.
    #
    # A reference, Python-double dependent implementation is commented out below
    # for your viewing pleasure.

    # fracResult = ""
    # doubleFrac = float("." + frac.__str__()) # I feel so ashamed for using python double
    # i = -1
    # while(i != -1075):
    #     if(doubleFrac >= 2**i):
    #         doubleFrac -= 2**i
    #         fracResult = fracResult + "1"
    #     else:
    #         fracResult = fracResult + "0"
    #     i -= 1
    # fracResult = fracResult.rstrip("0")

    # 2**(-1074) is approximately 4.6e-625, 2**(-1073) is 9.22e-625
    # We need to ensure precision of 2**(-1074). A naive try would be 1e626,
    # but we need much more than that to prevent precision loss beyond the point
    # of 2**-623. A experiment reveals we need to use 1e1074 as the starting seed,
    # so we can get all the 751 significant digits of 2**(-1074)

    # Starting from 2^(-1)...
    # ## PRECOMPILATION OF INTEGER-CODED NEGATIVE TWO POWERS FOR IEEE ##
    # For double, configure as follows: decPowerOfTenSeed = 1074 (1022+52),
    # minNegTwoPtr = 1075. minNegTwoPtr can be arbitrarily large, but don't
    # be wasteful.
    decPowerOfTenSeed = expL + P - 1
    minNegTwoPtr = expL + P + 1

    # Build this table (does not take long)
    negTwoIntResultsTable = [None]
    negTwoPtr  = 2
    negTwoCurr = int("5" + "0" * decPowerOfTenSeed)
    negTwoIntResultsTable.append(negTwoCurr)
    while(negTwoPtr != minNegTwoPtr):
        negTwoCurr = negTwoCurr // 2
        negTwoIntResultsTable.append(negTwoCurr)
        # print(negTwoPtr, ":", negTwoCurr) # diag only
        negTwoPtr += 1

    # Now we can do the calculations necessary
    fracResult = ""

    # First, padd the number to our desired Integer-coded precision
    # Given 0.5 = 5 + 0 * decPowerOfTenSeed (strlen = decPowerOfTenSeed + 1)
    frac = frac.ljust(1 + decPowerOfTenSeed, '0')
    intFrac = int(frac) # mutable
    i = -1
    while(i != (-1) * (minNegTwoPtr)):
        # print("* dec2double diag: i =", i, " intFracRemain =", intFrac, flush=True)
        if(intFrac >= negTwoIntResultsTable[(-1) * i]):
            fracResult = fracResult + "1"
            intFrac -= negTwoIntResultsTable[(-1) * i]
        else:
            fracResult = fracResult + "0"
```

```python
        i -= 1

    # Strip the trailing right-zeroes
    fracResult = fracResult.rstrip("0")

    # Align the dots (not stars) properly
    if(wholeResult == "0"):
        offsetTwoPower = (-1) * (len(fracResult) - len(fracResult.lstrip("0")) + 1)
        irBinaryRepTail = fracResult.lstrip("0")[1:]
    else:
        offsetTwoPower = len(wholeResult) - 1
        # irBinaryRep = wholeResult[:1] + "." + wholeResult[1:] + fracResult
        irBinaryRepTail = wholeResult[1:] + fracResult
    # print("* dec2double diag: intermediate result " + irBinaryRepTail + " x 2^", offsetTwoPower)

    # Calculate the exponential offset (+1023 for IEEE Double Float)
    # 2^(e-1) - 1, e is the length of the exponential bit storage size
    # e = 11 for double-precision
    decExpOffset = 2**(E - 1) - 1 + offsetTwoPower

    # Convert to binary (copied from above)
    binExpOffset = ""
    i = E - 1
    while(i != -1):
        # print("* dec2double diag: [whole] whole = ", whole, " checking i =", i, flush=True)
        if(decExpOffset >= 2**i):
            decExpOffset -= 2**i
            binExpOffset = binExpOffset + "1"
        else:
            binExpOffset = binExpOffset + "0"
        i -= 1

    # print("* dec2double diag: expOffset (bin) =", binExpOffset)

    # Generate the final IEEE double-precision representation,
    # chopping off any non-significant bits (sorry precision!)
    result = sign.__str__()
    result = result + binExpOffset
    if(len(irBinaryRepTail) < P - 1):
        result = result + irBinaryRepTail.ljust(P - 1, '0')
    else:
        # Handle rounding errors... round up if necessary
        if(irBinaryRepTail[P-1] == "1"):
            lookAt = P-2
            while(irBinaryRepTail[lookAt] == "1"):
                # Also note that we need to decrement the 1s on this way...
                irBinaryRepTail = irBinaryRepTail[0:lookAt] + "0" + irBinaryRepTail[(lookAt+1):]
                lookAt -= 1
            irBinaryRepTail = irBinaryRepTail[0:lookAt] + "1" + irBinaryRepTail[(lookAt+1):]

        result = result + irBinaryRepTail[0:(P-1)]

    return result

print(dec2ieee(input("> dec: ")))
```

二进制转十进制

```
######################################################################
# Computational Physics, 2017-18 Sem1
# HW-1 Ex-1
#
# (c) 2017 Haipeng Lin <linhaipeng@pku.edu.cn>
# All Rights Reserved.
#
# This program is written as Homework for the Computational Physics
# Course in Peking University, School of Physics.
# NO WARRANTIES, EXPRESS OR IMPLIED, ARE OFFERED FOR THIS PRODUCT.
# Copying is strictly prohibited and its usage is solely restricted
# to uses permitted by the author and homework grading.
#
# This program is Python 3 (3.6.2 on MSYS_NT)
# compatible, and does not support Python 2.
#
# Now Playing: 山丘 - 李宗盛
######################################################################

######################################################################
# IEEE Any-Precision Floating Point Format to Decimal
######################################################################
# string ieee2dec(string strI)
# Accepts i in "string" format, returning the decimal result
# in a "string" format. (By default as an example we give double)
# The function is freely configurable.
def ieee2dec(strN):
    # ~ * Configuration Parameters Here * ~
    # ieee2dec supports any (theoretically) IEEE-specified precision spec,
    # simply configure the parameters below to your liking.
    #
    # Beta = 2 (fixed for binary, this you can't change, sorry)
    # Precision: Number of precision bits (can be ARBITRARY), d0 ~ d(p-1)
    P = 53
    # L (lower exponential bound), E >= -L
    expL = 1022
    # U (upper exponential bound), E <= +U
    expU = 1023
    # Exponential Bits
    E = 11
    # The one sign bit is contained in P.
    # All numbers generated are SIGNED (for UNSIGNED it is trivial to modify the below code.)

    # ~ * NO USER CONFIGURABLE VARIABLES BELOW * ~
    result = ""

    # Split the strings according to configuration.
    signBit = strN[0]
    if(signBit == "1"):
        # is negative
        result = result + "-"

    expBits = strN[1:(E+1)]
    valBits = strN[(E+1):]

    # Calculate the exponent by deducting the exponent bias
    # (IEEE defined as 2**(E-1) - 1)
    expVal = 1 - 2**(E-1)
    i = E - 1
    while(i != -1):
        expVal += 2**(E-1-i) * (1 if expBits[i] == "1" else 0)
        i -= 1

    # Isolate ValBits through exponent.
    # If exponent is non-negative, then simply
    # 1.xxxxxxxx -> move . to the right expVal bits
    wholeBits = ""
    fractionBits = ""

    if(expVal >= 0):
        wholeBits = "1" + valBits[:(expVal)]
        if(len(wholeBits) < (1 + expVal)):
            wholeBits = wholeBits.ljust(1 + expVal, "0")
        fractionBits = valBits[(expVal):]
        if(len(fractionBits) == 0):
            fractionBits = "0"
    else:
        wholeBits = "0"
        # If its, e.g. raw result
        # 0.0001100101... -> 1.100101... x 2^(-4) (base 2)
        # then the fractional result is 0001100101..
        # Using 4-1=3 padded left-zeroes, then a 1, then the given component
        # (100101...) encoded in the IEEE standard
        fractionBits = "0" * ((-1) * expVal - 1) + "1" + valBits

        # print("* ieee2dec diag: expVal=", expVal, " is neg, wholeBits=0, fractionBits=", fractionBits)

    # print("* ieee2dec diag: expVal conversion", expBits, "->", expVal)

    # For the whole number component it is easy to recreate
    # by simply multiplying by 2 where it counts
    # Up from expU down to 0
    # We refrain from left-padding zeroes; instead we simply loop on the string
    # Reverse the bits
    wholeResult = 0
    # print("* ieee2dec diag wholeBits:", wholeBits)
    wholeBitsR = wholeBits[::-1]
    for i in range(len(wholeBitsR)):
        wholeResult += 2**i * (1 if wholeBitsR[i] == "1" else 0)

    fracResult = 0
    # For higher-precision, non-Python double restrained results
    # we need to do as dec2ieee and pre-build a decimal table,
    # then later adjust precision as needed.
    # ## PRECOMPILATION OF INTEGER-CODED NEGATIVE TWO POWERS FOR IEEE ##
    # For double, configure as follows: decPowerOfTenSeed = 1074 (1022+52),
    # minNegTwoPtr = 1075. minNegTwoPtr can be arbitrarily large, but don't
    # be wasteful.
    decPowerOfTenSeed = expL + P - 1
    minNegTwoPtr = expL + P + 1

    # Build this table (does not take long)
    negTwoIntResultsTable = []
    negTwoPtr  = 2
    negTwoCurr = int("5" + "0" * decPowerOfTenSeed)
    negTwoIntResultsTable.append(negTwoCurr)
    while(negTwoPtr != minNegTwoPtr):
        negTwoCurr = negTwoCurr // 2
        negTwoIntResultsTable.append(negTwoCurr)
        negTwoPtr += 1

    # Now simply generate the number. No padding is required as the
    # fracResult is already post-decimal divider component.
    for i in range(len(fractionBits)):
        # note this begins with 0, 1, 2, ...
        # but the actual index meaning should be -1, -2, -3, ...
        # luckily, negTwoIntResultsTable is also 0, 1, 2, ... indexed
        # print("* ieee2dec diag: adding", negTwoIntResultsTable[i])
        fracResult += negTwoIntResultsTable[i] * (1 if fractionBits[i] == "1" else 0)

    # Padd fracResult to desired precision
    fracResult = fracResult.__str__()
    if(len(fracResult) < (1 + decPowerOfTenSeed)):
        fracResult = fracResult.rjust(1 + decPowerOfTenSeed, "0")

    # Chop chop
    fracResult = fracResult.rstrip("0")

    return result

print(ieee2dec(input("> ieee: ")))
```

计算物理学 “大作业” 文档 (二)

常微分方程初值问题的有限差分解法

【题目】

（二）【10分】^{235}U核是放射性核，能衰变成两个质量较小的原子核。现假定它以如下规律衰变：

$$\frac{dN(t)}{dt} = -\frac{N(t)}{\tau}, \tag{1}$$

初始时刻$N(0) = 100$，衰变常数$\tau = 1$秒。请利用有限差分编写程序计算$t = 0 - 5$秒里U核的数目$N(t)$，利用四种时间步长$\Delta t = 0.4, 0.2, 0.1, 0.05$，并与精确结果比较。最后，将所有结果画在一张图上。

【编写思路与算法描述】

利用有限差分方法求解该常微分方程的初值问题. 查阅相关资料, 问题的抽象表述和解决方法如下:

$$\begin{cases} \dfrac{dy}{dx} = f(x,y), \qquad a \le x \le b, \\ \qquad\quad y(a) = y_0 \end{cases}$$

其中 f 是 x, y 的已知函数, y_0是给定的初值.

在区间 [a, b] 上引入有限个离散点 $a = x_0 < x_1 < \cdots < x_N = b$, 计算这些离散点的函数 y(x) 的近似值 y_n $(n = 1, 2, \dots, N)$. 就可以解决本问题. 这里的 x_i 取等间距, 意即 $x_n = a + nh, n = 0, 1, \dots, N$. 并称 $h = (b - a)/N$ 为步长.

根据数值微商公式

$$f'(x_i) \approx \frac{f(x_{i+1}) - f(x_i)}{h}$$

我们可以推导出向前 Euler 格式的常微分方程初值问题求解方法. 假设 y(x) 是解, 其在 [a, b] 上有连续的二阶导数.

$$\mathrm{y}(\mathrm{x}_{\mathrm{n+1}}) = \mathrm{y}(\mathrm{x}_{\mathrm{n}}) + hy'(x_n) + \frac{h^2}{2}y''(\xi_n) = y(x_n) + hf\big(x_n, y(x_n)\big) + \frac{h^2}{2}y''(\xi_n)$$

略去高阶项逼近后就有

$$y_{n+1} = y_n + hf(x_n, y_n), n = 0, 1, \dots, N-1.$$

这是一个单步一阶显式格式.

我们就利用这个方法对本题中的方程进行求解, 其中题目也给定了步长.

【运行方式】

源码直接调用 eulerForwardSolve 函数.

```
eulerForwardSolve(f x, y: f(x, y), a, b, y₀, dt = None)
```

- f 接受 lambda x, y 供给 f(x, y) 值.
- a, b 为求解自变量范围 $x \in [a, b]$
- y_0 为 y(a) 的初值.
- dt 为指定步长, 如不指定步长, 将会进行 1000-等分 (这个数字是随意制定的, 为了代码的编写严谨性而设置, 在本次题目中并不需要用到.)

返回值: 含有 Tuple (x, y) 的数组.

注意, 如果步长无法恰好整除覆盖 [a, b], 则最后一个值会超出 b.

源码包含了四种步长的呼叫和输出函数.

【运行结果】原始运行结果:

dt = 0.4s	dt = 0.2s
(0, 100)	(0, 100) (0.2, 80.0) (0.4, 64.0)
(0.4, 60.0)	(0.6, 51.2) (0.8, 40.96) (1.0, 32.768)
(0.8, 36.0)	(1.2, 26.2144) (1.4, 20.97152)
(1.2000000000000002, 21.6)	(1.6, 16.777216) (1.8, 13.4217728)
(1.6, 12.96)	(2.0, 10.73741824)
(2.0, 7.776)	(2.2, 8.589934592)
(2.4, 4.6655999999999995)	(2.4, 6.871947673600002)
(2.8, 2.7993599999999996)	(2.6, 5.497558138880001)

```
(3.2, 1.6796159999999998)
(3.6, 1.0077695999999998)
(4.0, 0.6046617599999999)
(4.4, 0.36279705599999995)
(4.8, 0.21767823359999997)
(5.2, 0.13060694015999996)
```

```
(2.8, 4.398046511104001)
(3.0, 3.5184372088832006)
(3.2, 2.8147497671065604)
(3.4, 2.251799813685248)
(3.6, 1.8014398509481986)
(3.8, 1.4411518807585588)
(4.0, 1.152921504606847)
(4.2, 0.9223372036854777)
(4.4, 0.7378697629483821)
(4.6, 0.5902958103587057)
(4.8, 0.47223664828696454)
(5.0, 0.37778931862957166)
```

dt = 0.1s

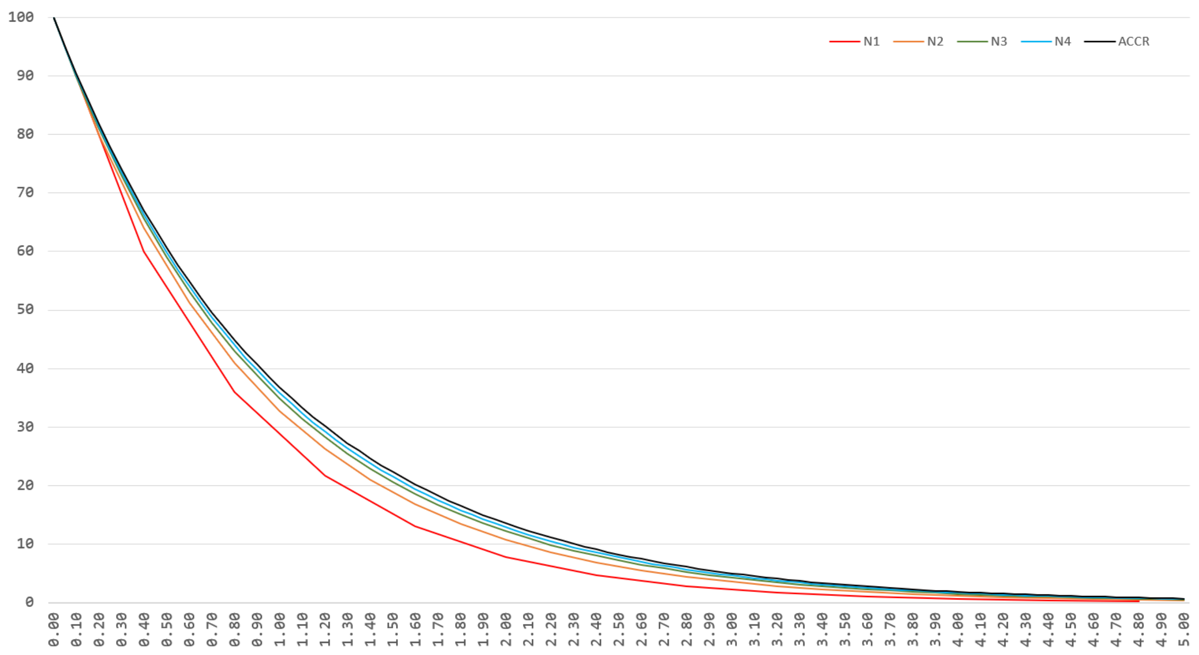
```
(0, 100)
(0.1, 90.0)
(0.2, 81.0)
(0.30000000000000004, 72.9)
(0.4, 65.61)
(0.5, 59.049)
(0.6, 53.1441)
(0.7, 47.82969)
(0.7999999999999999, 43.046721)
(0.8999999999999999, 38.7420489)
(0.9999999999999999, 34.86784401)
(1.0999999999999999, 31.381059608999998)
(1.2, 28.2429536481)
(1.3, 25.41865828329)
(1.4000000000000001, 22.876792454961)
(1.5000000000000002, 20.589113209464898)
(1.6000000000000003, 18.53020188851841)
(1.7000000000000004, 16.67718169966657)
(1.8000000000000005, 15.009463529699913)
(1.9000000000000006, 13.508517176729920)
(2.0000000000000004, 12.157665459056929)
(2.1000000000000005, 10.941898913151237)
(2.2000000000000006, 9.847709021836113)
(2.3000000000000007, 8.862938119652503)
(2.400000000000001, 7.976644307687252)
(2.500000000000001, 7.178979876918527)
(2.600000000000001, 6.461081889226675)
(2.700000000000001, 5.814973700304007)
(2.800000000000001, 5.233476330273606)
(2.9000000000000012, 4.710128697246246)
(3.0000000000000013, 4.239115827521621)
(3.1000000000000014, 3.815204244769459)
(3.2000000000000015, 3.4336838202925133)
(3.3000000000000016, 3.090315438263262)
(3.4000000000000017, 2.781283894436936)
(3.5000000000000018, 2.5031555049932424)
(3.600000000000002, 2.252839954493918)
(3.700000000000002, 2.027555959044526)
(3.800000000000002, 1.8248003631400733)
(3.900000000000002, 1.642320326826066)
(4.000000000000002, 1.4780882941434594)
(4.100000000000001, 1.3302794647291134)
(4.200000000000001, 1.197251518256202)
(4.300000000000001, 1.0775263664305819)
(4.4, 0.9697737297875236)
(4.5, 0.8727963568087713)
(4.6, 0.7855167211278942)
(4.699999999999999, 0.7069650490151047)
(4.799999999999999, 0.6362685441135942)
(4.899999999999999, 0.5726416897022348)
(4.999999999999998, 0.5153775207320114)
(5.099999999999998, 0.4638397686588102)
```

dt = 0.05s

```
(0, 100)
(0.05, 95.0)
(0.1, 90.25)
(0.15, 85.7375)
(0.2, 81.450625)
(0.25, 77.37809375)
(0.3, 73.5091890625)
(0.35, 69.833729609375)
(0.40, 66.34204312890625)
(0.44999999999999996, 63.02494097246094)
(0.49999999999999994, 59.87369392383789)
(0.5499999999999999, 56.880009227646)
(0.6, 54.0360087662637)
(0.65, 51.33420832795051)
(0.70, 48.76749791155299)
(0.75, 46.329123015975334)
(0.80, 44.01266686517657)
(0.85, 41.81203352191774)
(0.90, 39.721431845821854)
(0.95, 37.735360253530764)
(1.00, 35.848592240854224)
(1.05, 34.056162628811514)
(1.10, 32.35335449737094)
(1.15, 30.73568677250239)
(1.20, 29.198902433877272)
(1.25, 27.73895731218341)
(1.30, 26.35200944657424)
(1.3500000000000005, 25.034408974245526)
(1.4000000000000006, 23.78268852553325)
(1.4500000000000006, 22.593554099256586)
(1.5000000000000007, 21.463876394293756)
(1.5500000000000007, 20.390682574579067)
(1.6000000000000008, 19.371148445850114)
(1.6500000000000008, 18.40259102355761)
(1.7000000000000008, 17.48246147237973)
(1.7500000000000009, 16.608338398760743)
(1.800000000000001, 15.777921478822707)
(1.850000000000001, 14.989025404881572)
(1.900000000000001, 14.239574134637493)
(1.950000000000001, 13.527595427905618)
(2.000000000000001, 12.851215656510337)
(2.0500000000000007, 12.208654873684821)
(2.1000000000000005, 11.59822213000058)
(2.1500000000000004, 11.018311023500551)
(2.2, 10.467395472325524)
(2.25, 9.944025698709249)
(2.3, 9.446824413773786)
(2.3499999999999996, 8.974483193085097)
(2.3999999999999995, 8.525759033430843)
(2.4499999999999993, 8.099471081759301)
(2.499999999999999, 7.694497527671336)
(2.549999999999999, 7.309772651287769)
(2.5999999999999988, 6.944284018723381)
(2.6499999999999986, 6.597069817787212)
(2.6999999999999984, 6.267216326897851)
(2.7499999999999982, 5.953855510552959)
(2.799999999999998, 5.656162735025311)
(2.849999999999998, 5.373354598274045)
(2.8999999999999977, 5.104686868360343)
(2.9499999999999975, 4.8494525249423255)
```

```
(2.9999999999999973, 4.606979898695209)
(3.049999999999997, 4.376630903760448)
(3.099999999999997, 4.157799358572426)
(3.149999999999997, 3.9499093906438043)
(3.1999999999999966, 3.7524139211116143)
(3.2499999999999964, 3.5647932250560337)
(3.2999999999999963, 3.386553563803232)
(3.349999999999996, 3.21722588561307)
(3.399999999999996, 3.0563645913324167)
(3.4499999999999957, 2.903546361765796)
(3.4999999999999956, 2.758369043677506)
(3.5499999999999954, 2.620450591493631)
(3.599999999999995, 2.4894280619189493)
(3.649999999999995, 2.3649566588230018)
(3.699999999999995, 2.2467088258818517)
(3.7499999999999947, 2.134373384587759)
(3.7999999999999945, 2.0276547153583713)
(3.8499999999999943, 1.9262719795904526)
(3.899999999999994, 1.8299583806109299)
(3.949999999999994, 1.7384604615803834)
(3.999999999999994, 1.6515374385013641)
(4.049999999999994, 1.568960566576296)
(4.099999999999993, 1.490512538247481)
(4.149999999999993, 1.4159869113351071)
(4.199999999999993, 1.3451875657683519)
(4.249999999999993, 1.2779281874799342)
(4.299999999999993, 1.2140317781059375)
(4.3499999999999925, 1.1533301892006407)
(4.399999999999992, 1.0956636797406087)
(4.449999999999992, 1.0408804957535782)
(4.499999999999992, 0.9888364709658993)
(4.549999999999992, 0.9393946474176044)
(4.599999999999992, 0.8924249150467242)
(4.6499999999999915, 0.847803669294388)
(4.699999999999991, 0.8054134858296687)
(4.749999999999991, 0.7651428115381852)
(4.799999999999991, 0.7268856709612759)
(4.849999999999991, 0.690541387413212)
(4.899999999999991, 0.6560143180425514)
(4.94999999999999, 0.6232136021404239)
(4.99999999999999, 0.5920529220334027)
(5.04999999999999, 0.5624502759317325)
```

比较四种步长与精确结果绘图: (N1 dt = 0.4s, N2 dt = 0.2s, N3 dt = 0.1s, N4 dt = 0.05s, ACCR 为精确值)



【误差分析】

向前 Euler 方法的误差分析是很完备的. 通常的分析方法是考虑典型的微分方程 (实验方程)

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

考虑步长为 h 时, 如果只对某步产生误差, 以后各步长为 h 的步都能逐步削弱误差, 则称为解法对于 $\xi = \lambda h$ 是**绝对稳定**的. 对于向前 Euler 方法, 绝对稳定的区域可以如下方法求出. 格式

$$y_{n+1} = y_n + hf(x_n, y_n) = y_n + \lambda h y_n$$

如果存在误差, y_n 变为 y_n^*

$$y_{n+1}^* = y_n^* + \lambda h y_n^*$$

令 $e_n = y_n^* - y_n$, 两式相减得到

$$e_{n+1} = e_n + \lambda h e_n = (1+\xi)e_n$$

如果让误差逐步削弱则有

$$|1+\xi| < 1$$

事实上, 如果我们让 h 变得更小自然就可以得到更精细的结果. 根据我们需要多少有效位数的结果, 我们会采取什么样的计算步长. 但是这并不是无限度的. 如果我们希望能够更快得到更精确的结果, 我们不能用小步长而是应该使用一个更精确的、高阶算法, 比如 Runge-Kutta 方法, 或线性多步法.

【源代码】

本源代码不依赖于任何外界库.

```
######################################################################
# Computational Physics, 2017-18 Sem1
# HW-1 Ex-2
#
# (c) 2017 Haipeng Lin <linhaipeng@pku.edu.cn>
# All Rights Reserved.
#
# This program is written as Homework for the Computational Physics
# Course in Peking University, School of Physics.
# NO WARRANTIES, EXPRESS OR IMPLIED, ARE OFFERED FOR THIS PRODUCT.
# Copying is strictly prohibited and its usage is solely restricted
# to uses permitted by the author and homework grading.
#
# This program is Python 3 (3.6.2 on darwin)
# compatible, and does not support Python 2.
#
# Now Playing: I know - Tom Odell
######################################################################

# Solves a linear differential equation with an initial condition
# dy/dx = f(x, y)  a <= x <= b
# y(a) = y_0

# Euler Forward Method
# eulerForwardSolve(\x, y -> f(x,y), a, b, y0, dt = None)
# If timestep dt is none, then a default value will be used given N = 1000 partitions
# Returns the values for all the partitioned f(x_i) = y_i
#
# FIXME: 1000 partitions is completely arbitrary
# although not relevant in this exercise as dt is given in the problem.
def eulerForwardSolve(f, a, b, y0, dt = None):
    k = 0
    if(dt != None):
        h = dt    # timestep given
    else:
        h = (b-a)/1000
    xn = a
    ys = [(a,y0)]

    while((xn) <= b): # do a little nudging because python's double is a little too sensitive
        xn += h
        ys.append((xn, ys[-1][1] + h * f(xn, ys[-1][1])))

    return ys

# Actual Code for exercise 2
# Solve
#   dN(t)        N(t)
#  ------- = - -------
#    dt          tau       (tau = 1s)
#
print("dt = 0.4s")
r1 = (eulerForwardSolve(lambda x, y: (-1)*y/1, 0, 5, 100, 0.4))
for i in range(1, len(r1) + 1):
    print(r1[i-1])

print("dt = 0.2s")
r2 = (eulerForwardSolve(lambda x, y: (-1)*y/1, 0, 5, 100, 0.2))
for i in range(1, len(r2) + 1):
    print(r2[i-1])

print("dt = 0.1s")
r3 = (eulerForwardSolve(lambda x, y: (-1)*y/1, 0, 5, 100, 0.1))
for i in range(1, len(r3) + 1):
    print(r3[i-1])

print("dt = 0.05s")
r4 = (eulerForwardSolve(lambda x, y: (-1)*y/1, 0, 5, 100, 0.05))
for i in range(1, len(r4) + 1):
    print(r4[i-1])
```

计算物理学 “大作业” 文档 (三)

Gauss 求积公式的系数求解

【题目】

（三）【10分】请分别构造如下三种非标准权函数的Guass求积系数的求积系数和节点：

$$\int_0^1 \sqrt{x} f(x)dx \approx \quad A_0 f(x_0) + A_1 f(x_1); \tag{2}$$

$$\int_{-1}^1 (1+x^2) f(x)dx \approx \quad A_0 f(x_0) + A_1 f(x_1); \tag{3}$$

$$\int_0^1 \sqrt{x} f(x)dx \approx \quad A_0 f(x_0) + A_1 f(x_1) + A_2 f(x_2). \tag{4}$$

【求解】

本题无需计算机求解即可完成, 只需要把积分算出来并求解高次方程组即可.

对 (2) 式, 其具有 A_0, x_0, A_1, x_1 四个未知数. 我们要求其对于 f(x) = 1, x, x^2, x^3 准确成立, 得

$$\begin{cases} \int_0^1 \sqrt{x}dx = A_0 + A_1 = \frac{2}{3} \\ \int_0^1 x\sqrt{x}dx = A_0x_0 + A_1x_1 = \frac{2}{5} \\ \int_0^1 x^2\sqrt{x}dx = A_0x_0^2 + A_1x_1^2 = \frac{2}{7} \\ \int_0^1 x^3\sqrt{x}dx = A_0x_0^3 + A_1x_1^3 = \frac{2}{9} \end{cases} \Rightarrow \begin{cases} A_0 = \frac{1}{150}(50 \mp \sqrt{70}) \\ x_0 = \frac{1}{63}(35 \mp 2\sqrt{70}) \\ A_1 = \frac{1}{150}(50 \pm \sqrt{70}) \\ x_1 = \frac{1}{63}(35 \pm 2\sqrt{70}) \end{cases}$$

(事实上两个解是对称的, 因而对于 Gauss 求积公式具有意义的解是唯一的)

对 (3) 式同理.

$$\begin{cases} \int_{-1}^1 (1+x^2)dx = A_0 + A_1 = \frac{8}{3} \\ \int_{-1}^1 x(1+x^2)dx = A_0x_0 + A_1x_1 = 0 \\ \int_{-1}^1 x^2(1+x^2)dx = A_0x_0^2 + A_1x_1^2 = \frac{16}{15} \\ \int_{-1}^1 x^3(1+x^2)dx = A_0x_0^3 + A_1x_1^3 = 0 \end{cases} \Rightarrow \begin{cases} A_0 = \frac{4}{3} \\ x_0 = \mp\sqrt{\frac{2}{5}} \\ A_1 = \frac{4}{3} \\ x_1 = \pm\sqrt{\frac{2}{5}} \end{cases}$$

(事实上两个解是对称的, 因而对于 Gauss 求积公式具有意义的解是唯一的)

对 (4) 式, 有六个未知数. 从而我们要求求积公式对于 $f(x) = 1, x, x^2, x^3, x^4, x^5$ 准确成立. 得

$$\begin{cases} \int_0^1 \sqrt{x}dx = A_0 + A_1 + A_2 = \frac{2}{3} \\ \int_0^1 x\sqrt{x}dx = A_0x_0 + A_1x_1 + A_2x_2 = \frac{2}{5} \\ \int_0^1 x^2\sqrt{x}dx = A_0x_0^2 + A_1x_1^2 + A_2x_2^2 = \frac{2}{7} \\ \int_0^1 x^3\sqrt{x}dx = A_0x_0^3 + A_1x_1^3 + A_2x_2^2 = \frac{2}{9} \\ \int_0^1 x^4\sqrt{x}dx = A_0x_0^4 + A_1x_1^4 + A_2x_2^4 = \frac{2}{11} \\ \int_0^1 x^3\sqrt{x}dx = A_0x_0^5 + A_1x_1^5 + A_2x_2^5 = \frac{2}{13} \end{cases} \Rightarrow \begin{cases} A_0 = 0.307602 \\ x_0 = 0.549868 \\ A_1 = 0.125783 \\ x_1 = 0.164710 \\ A_2 = 0.233282 \\ x_2 = 0.900806 \end{cases}$$

由于本次大作业的范围是 1~5 讲, 不包含 “非线性方程求根” 的问题求解要求, 经询问老师后确认给出方程解即可. 更高精度的结果可以通过数值计算得出.

计算物理学 “大作业” 文档 (四)

球谐函数的数值求解: Legendre 多项式与阶乘等递归函数高效率的求值

【题目】

（四）【15分】在实际问题中，球谐函数是一种非常重要的特殊函数：

$$Y_{lm}(\theta,\phi) = \sqrt{\frac{(2l+1)}{4\pi}\frac{(l-m)!}{(l+m)!}}P_l^m(\cos\theta)e^{im\phi}, \tag{5}$$

其中关联勒让德函数$P_l^m(x)$满足偏微分方程：

$$\frac{d}{dx}\left[(1-x^2)\frac{dP}{dx}\right]+\left[l(l+1)-\frac{m^2}{1-x^2}\right]P=0, \quad x=\cos\theta \tag{6}$$

其中$l = 0, 1, 2, \cdots$，$m = -l, -l+1, \cdots, 0, \cdots, l-1, l$。请自行查阅相关数学资料，设计算法，编写出高效而准确子函数程序SphericalHarmonics，对任意给定L, M, θ, ϕ，返回精确的SphericalHarmonics$[L, M, \theta, \phi]$值。作为测试，固定$\phi = \pi/6$，而$\theta = \pi/1000, 3\pi/10, 501\pi/1000$，列表给出$L = 100, 500, 1000$，而$M$分别为$1, L/100, L/10, L-1$时，你的程序的运行结果，每个结果要求至少有10位有效数字。调试和检查你的程序时，你可以利用Mathematica等软件的内置函数直接进行高精度的运算来检查你自己程序的结果的可靠性和精确性。

【补充数学材料】

查阅数学资料可知如下一些有用的结论, 在下的论述中会直接使用它们.

关联勒让德函数 (Associated Legendre Function) $P_l^m(x)$ $(x = cos\theta)$ 是 Legendre 方程的解:

$$\frac{\mathrm{d}}{\mathrm{dx}}\left[(1-x^2)\frac{dP}{dx}\right]+\left[l(l+1)-\frac{m^2}{1-x^2}\right]P=0$$

其中, $l = 0, 1, 2, \ldots$, $m = -l, -l+1, \ldots, 0, \ldots, l-1, l$.

关联勒让德函数由勒让德多项式定义如下

$$P_l^m(x) = (-1)^m(1-x^2)^{\frac{m}{2}}\frac{d^m}{dx^m}P_l(x)$$

其中, **勒让德多项式**又可以定义如下

$$P_n(x) = \frac{1}{2^n n!}[\frac{d^n}{dx^n}(x^2-1)^n]$$

根据勒让德多项式的定义和数学物理方法的知识我们可以得知递推关系

$$nP_n(x) = (2n-1)xP_{n-1}(x) - (n-1)P_{n-2}, \quad P_0(x) = 1, P_1(x) = x$$

这是计算勒让德多项式的很好的办法, 也是程序采用的办法.

连带勒让德多项式也满足一定的递推关系, 其中最主要的是

$$(l-m)P_l^m(x) = (2l-1)xP_{l-1}^m(x) - (l+m-1)P_{l-2}^m(x)$$

这是因为我们可以通过这种方式把任何待求的结果化简为下面几个初值

$$P_l^l(x) = (-1)^l(2l-1)!!\,(1-x^2)^{l/2}$$
$$P_l^m(x) = 0, \qquad |m| > l$$

【编写思路与算法描述】

**** 最终的源代码请参见本部分最后的附录, 编写思路中的代码是中间过程的实例, 不是最终答案 ****

该题目的运算主要就依赖于如何实现 LegendrePoly 这个函数. 首先作为练习和试探精度, 我的初步想法是首先计算出 Legendre 多项式 (LegendrePolyNA, **N**on-**A**ssociated), 然后再用数值微分进行计算. 数值微分的代码用中心差商和递归做高阶的, 实现如下:

```
########################################################################
# Numerical Differentiation
# Given a function f(x), get a numerical f'(x) result
# The timestep is dynamically assigned.
# f(x) must accept continuous values.
########################################################################
```

```
# Arbitrarily defined timestep
h_default = 0.0001

# derivF_c(f = lambda x: f(x), x, h)
def derivF_c(f, x, h = None):
        if(h == None):
                h = h_default
        return (f(x + h) - f(x - h))/(2 * h)

# deriv2F_c(f = lambda x: f(x), x, h)
def deriv2F_c(f, x, h = None):
        if(h == None):
                h = h_default
        return (f(x + h) - 2 * f(x) + f(x - h))/(h * h)

# get f(level) derivative
# this is VERY inaccurate and should only be used for 1-3 levels
def derivF_c_cycle(f, x, level = 1, h = None):
        if(level == 0):
                return f(x)
        elif(level == 2):
                return deriv2F_c(f, x, h)
        elif(level == 1):
                return derivF_c(f, x, h)

        return derivF_c_cycle(lambda x: derivF_c(f, x), x, level - 1, h)
```

我们可以根据定义书写 LegendrePolyNA:

```
# Utility: Factorial Function
# factorial(n) returns n! (for n in natural number)
def factorial(n):
        if(n == 0):
                return 1
        return n * factorial(n - 1)

# LegendrePolyNA (Non-Associated)
def legendrePolyNA(n, x):
        # By definition (don't use this, it's completely not good)
        return 1/(2**n * factorial(n)) * derivF_c_cycle(lambda x: (x**2 - 1)**n, x, n)
```

经过测试发现这种方法误差太大了, 大到高阶几乎没有任何有效数字可言, 低阶也只有 4-5 位数. 从而, **我们肯定不能采用数值微分的方法, 而是通过传统的递推进行操作**. 根据 Legendre 多项式的递推性质我们可以书写出一个基本的 LegendrePolyNA 函数:

```
def legendrePolyNA(n, x):
    # By definition (don't use this, it's completely not good)
    # return 1/(2**n * factorial(n)) * derivF_c_cycle(lambda x: (x**2 - 1)**n, x, n)

    # Using the recursive relation for Pn(x):
    # (n)P_n = (2n-1)xP_n-1 - (n-1)P_n-2, P0=1, P1=x
    # we can have better results..

    if(n == 0):
        return 1
    elif(n == 1):
        return x

    # print("* legendrePolyNA diag: n = ", n, ", x = ", x, flush=True)
    return (2*n-1)/n*x*legendrePolyNA(n-1, x) - (n-1)/n*legendrePolyNA(n-2, x)
```

这样的方法虽然很好, **对于任意阶精度可以达到约 11 位有效位数**, 但是由于 Python 不是一个尾递归 (Tail Recursion) 优化的编程语言, 过深的递归深度一方面会造成运算量的指数爆炸, 也会导致 Python 释放出 RecursionError. 因此我们需要对于大的 n 进行一个优化, 将低阶运算结果积累并缓存起来. 基于这样的 Caching 思路我们就设计出了最终程序的 LegendrePolyNA 函数 (参见源代码), 在 n > 500 作为 Caching 分界点; 在此之下的计算足够快而不需要 Caching 就可以进行 (测试环境: i5-4300U 1.8GHz)

对于 LegendrePolyA 也是类似的思路, 只是这里尾递归不优化的缺点更加明显了, 因此 Caching 分界在 L > 25 的时候就开始进行. 对于 M 较小的情况, 我们可以依赖于这个递推式

$$(l-m)P_l^m(x) = (2l-1)xP_{l-1}^m(x) - (l+m-1)P_{l-2}^m(x)$$

对于 M 较大的情况事实上还有更加优化的递推式, 这个可以通过判断 M/L 大小进行选择, 也许会得到更快的算力. 但是目前使用的是上述递推式, 因为可以保持 M 固定而只计算 L 递减的情况, 其他的递推式基本上都会增加 M 的大小 (这在 M 相对 L 较大的时候自然是很有优势的, 因为 |M|>L 则 LegendrePolyA = 0). **我的源码中有对于 M/L 大小判断进行选择的一个实验性模块, 但在实际使用中没有开启的必要.**

最后还有几个小问题需要解决, 一个是三角函数 cos 的实现, 另一个是指数函数 e 的实现. 这些都比较容易, 加上 Python 原生支持的复数运算 (用 *j* 作为复数单位, 类似电力学的记号), 问题就迎刃而解了. 最终的子程序 *SphericalHarmonics* 算力, 精度都达到了要求.

程序中内置 pi 的值精确到 …9793, 是 Python Float (Double) 支持的最大精度限.

Cos 和 exp 的 Taylor 展开精度是任意的, 对于目前的用途, 分别取 45 和 32 项展开.

为了避免 RecursionError 的复发, Factorial 函数也进行了 Caching 优化, 经测试可以在<0.2s 以内计算 100000!.

此外, 为了优化球谐函数中的:

$$Y_{lm}(\theta,\phi) \propto \sqrt{\frac{(l-m)!}{(l+m)!}}$$

项 (在 l, m 足够大的时候, 比如 L > 100) 会导致双精度浮点数溢出的情况, 我特意设计了 *factorialFromTo* 函数, 可以接受 "从 a 到 b 的阶乘", 包括 a > b 的情况 (就是倒数), 该函数接受 *acc* 参数 (熟悉函数式编程语言 *Haskell* 的用户会对这个参数感到无比亲切.), 有一种类似 *fold* 的工作原理, 简言之就是提供一个非 1 的初始元, 防止除到最后小于最小双精度浮点数表示. 这个优化函数可以很好的解决 l, m 较大, 中间步骤向下溢出的情况:

```python
# Utility: FactorialFromTo
# If from > to (this is an acceptable usage.), then the inverse is given and precisely calculated, float-
first
# If acc (accumulator) is given, then the result is multiplied from the acc.
def factorialFromTo(a, b, acc = 1):
    result = acc
    if(a == b):
        return a

    if(a < b):
        for i in range(a + 1, b + 1):
            result = result * i
    elif(a > b):
        for i in range(b + 1, a + 1):
            result = result / i

    return result
```

最后一个需要解决的问题, 类似 *factorialFromTo* 的问题, 就是在 *legendrePolyA* 函数中, 如果 m 过度接近 L, 而 L 很大导致的浮点数向上溢出的情况:

$$P_l^l(x) = (-1)^l(\mathbf{2}\boldsymbol{l}-\mathbf{1})!!\,(1-x^2)^{l/2}$$

这个双阶乘的结果在 L=1000, M=999 时会取 L=999 (这是上述递推式的性质.) 因而对于 M 很接近而且 (2M-1)!! 向上溢出的情况, 我们需要进行特殊的处理. 结果在浮点数精度下需要满足精度要求, 因而可以如法炮制, 将中间步骤使用 *acc* "借用" 其他结果的位数, 可以避免使用额外计数法或者自行编写 (可能充满 bug 的) 任意精度浮点数库带来的麻烦. 具体的操作, 可以参见 *SphericalHarmonics* 函数中注释掉的 diagnostics 标记的 3 号与 4 号策略.

事实上, 浮点数精度的提高问题不在本题的编写范畴内. 希望深究的读者很容易可以对于 Python 做出相应的修改以支持目标精度的浮点数计算, 具体的细节可以参照 *cpython* 项目 (Python 编程语言的源代码), 对于`/Objects/floatobject.c` 做出对应的修改, 比如:

```c
PyObject *
PyFloat_FromDouble(long double fval)
```

我们就可以提高 Python *float* 的精度使其超过 C double.

希望深究的读者也可以利用 SWIG 库 (http://www.swig.org/Doc1.3/Python.html) 将 Python 接入更深层次的高精度浮点操作, 包括但不限于目前很流行的 AVX-512.

【算法性能】

对于所有测试数据样本在要求精度与浮点数精度范围内可以”近实时”计算, 时间 < 0.2s.

测试环境: MSYS_NT 10.0 (Windows 10, MinGW), Python 3.6.2 AMD64 on Intel® i5-4300U.

【运行方式】

调用 SphericalHarmonics 即可. LegendrePolyNA, LegendrePolyA 是关键函数, factorial, doubleFactorial, cos, exp 为自行实现的辅助函数. 它们共同组成该子程序.

【参考结果】

利用 *Wolfram Mathematica* 计算出的 *SphericalHarmonicY* 参考结果 (对于测试值) 参照下表, **取 12 位有效位数.**

固定 $\phi = \pi/6$.

L = 100:

M \ θ	$\pi/1000$	$3\pi/10$	$501\pi/1000$
1	-0.539985243909 -0.311760625929i	0.0939196480195 +0.0542245340663i	-0.0855954419091 -0.0494185514276i
L/100			
L/10	5.19887459224e-15 -9.00471493593e-15i	-0.177420280974 +0.307300940940i	-0.151741358911 +0.262823743243i
L-1	0 -0i	0 -6.09501379277e-9i	0 +0.0421585998443i

L = 500:

M \ θ	$\pi/1000$	$3\pi/10$	$501\pi/1000$
1	-4.38252794051 -2.53025368618i	0.0945489081647 +0.0545878375805i	-0.2756649951557 -0.1591552591593i
L/100	0.0174370877846 -0.0100673073263i	-0.0894595944097 +0.0516495209140i	0.275671630157 -0.159159089879i
L/10	7.96590544107e-70 +1.37973529522e-69i	-0.0115812530881 -0.0200593187640i	0.00100426934932 +0.00173944553750i
L-1	4.63966191086e-1248 +2.67871005318e-1248i	2.6862214708246e-45 +1.5508906892835e-45	-0.121657308310 -0.0702388797017i

L = 1000:

M \ θ	$\pi/1000$	$3\pi/10$	$501\pi/1000$
1	-3.10356246330 -1.79184262363i	0.0946281329889 +0.0546335780541i	0.000432689074695 +0.000249813153751i
L/100	0.000127373521632 -0.000220617411006i	-0.170161983756 +0.294729201382i	0.159159141899 -0.275671720249i
L/10	-2.40198398180e-138 +4.16035829545e-138i	0.174571415886 -0.302366561863i	0.159539306304 -0.276330184323i
L-1	0 -3.38240809650e-2499i	0 -4.96734844172e-91i	0 +0.235628131378i

【运行结果】

所有数据均满足题目精度要求, 标红的只是和 Wolfram Mathematica 参考数据的 11~12 位有效位不同的位置.

L = 100:

M \ θ	$\pi/1000$	$3\pi/10$	$501\pi/1000$
1	-0.53998524390**8** -0.311760625929i	0.0939196480195 +0.0542245340663i	-0.0855954419091 -0.0494185514276i
L/100			
L/10	5.198874592**11**e-15 -9.004714935**71**e-15i	-0.17742028097**3** +0.307300940940i	-0.151741358911 +0.262823743243i

L-1	0 -0i	0 -6.09501379277e-9i	0 +0.0421585998443i

L = 500:

M \ θ	$\pi/1000$	$3\pi/10$	$501\pi/1000$
1	-4.38252794051 -2.5302536861**9**i	0.094548908164**8** +0.0545878375805i	-0.2756649951557 -0.1591552591593i
L/100	0.017437087784**4** -0.010067307326**2**i	-0.0894595944097 +0.0516495209140i	0.275671630157 -0.159159089879i
L/10	7.9659054410**5**e-70 +1.37973529**519**e-69i	-0.011581253088**2** -0.0200593187640i	0.0010042693493**1** +0.00173944553750i
L-1	0 +0i	2.686221470824**8**e-45 +1.550890689283**0**e-45	-0.121657308310 -0.0702388797017i

L = 1000:

M \ θ	$\pi/1000$	$3\pi/10$	$501\pi/1000$
1	-3.1035624633**6** -1.7918426236**7**i	0.094628132989**0** +0.0546335780541i	0.00043268907470**2** +0.00024981315375**4**i
L/100	0.00012737352163**2** -0.00022061741100**0**i	-0.170161983756 +0.294729201382i	0.15915914189**3** -0.275671720249i
L/10	-2.401983981**21**e-138 +4.16035829545**5**e-138i	0.174571415886 -0.302366561863i	0.159539306304 -0.276330184323i
L-1	0 +0i	0 -4.96734844172e-91i	0 +0.235628131378i

经上述表比较, 在 double 精度的范围内, 所有给定数据结果满足要求.

如果希望提高精度的话事实上可以从浮点数库入手, 中间过程中有很多数相乘相除, 误差会得到一定程度的放大. 如果采用一个任意精度的库或者 "基于整数的科学计数法" (参见第一题中的实现方法.) 应该就可以得到一个很合理的结果, 类似 Mathematica 中的 precision 设置. 但是这样的实现超出了本题的讨论范围, 故省略.

【源代码】

该代码不依赖于任何外界声明, 是完全 *self-contained* 的.

```
#####################################################################
# Computational Physics, 2017-18 Sem1
# HW-1 Ex-4
#
# (c) 2017 Haipeng Lin <linhaipeng@pku.edu.cn>
# All Rights Reserved.
#
# This program is written as Homework for the Computational Physics
# Course in Peking University, School of Physics.
# NO WARRANTIES, EXPRESS OR IMPLIED, ARE OFFERED FOR THIS PRODUCT.
# Copying is strictly prohibited and its usage is solely restricted
# to uses permitted by the author and homework grading.
#
# This program is Python 3 (3.6.2 on darwin)
# compatible, and does not support Python 2.
#
# Now Playing: Wait - M83
#              Beautiful Now - Zedd
#              Until The End (Acoustic Special Edition) - Quietdrive
#####################################################################

#####################################################################
# Numerical Differentiation
# Given a function f(x), get a numerical f'(x) result
# The timestep is dynamically assigned.
# f(x) must accept continuous values.
#####################################################################

# Arbitrarily defined timestep
h_default = 0.0001

# 中心差商近似
# derivF_c(f = lambda x: f(x), x, h)
def derivF_c(f, x, h = None):
    if(h == None):
        h = h_default
    return (f(x + h) - f(x - h))/(2 * h)

# 二阶差商近似
# deriv2F_c(f = lambda x: f(x), x, h)
def deriv2F_c(f, x, h = None):
    if(h == None):
        h = h_default
    return (f(x + h) - 2 * f(x) + f(x - h))/(h * h)


# get f(level) derivative
# this is VERY inaccurate and should only be used for 1-3 levels
def derivF_c_cycle(f, x, level = 1, h = None):
    if(level == 0):
        return f(x)
    elif(level == 2):
        return deriv2F_c(f, x, h)
    elif(level == 1):
        return derivF_c(f, x, h)

    return derivF_c_cycle(lambda x: derivF_c(f, x), x, level - 1, h)

#####################################################################
# Computing Legendre Polynomials
#####################################################################
# Utility: Factorial Function
# factorial(n) returns n! (for n in natural number)
# With Caching w/ Automatic Compilation (to avoid RecursionError)
factorial_cache = {0: 1, 1: 1}
def factorial(n, isCache = None):
    if(n == 0):
        return 1

    if(n > 500 and isCache == None):
        for i in range(249, n//2):
            factorial(i*2, True)

    cache = factorial_cache.get(n, None)
    if(cache == None):
        factorial_cache[n] = n * factorial(n - 1, isCache)
    return factorial_cache[n]

# Utility: Double Factorial Function
# doubleFactorial(n) returns n!! (double factorial) (for n in natural number)
# If acc (accumulator) is given, then the result is multiplied from the acc.
#   * WARNING: If you provide a acc, tail recursion is disabled (so is caching)
#       (doesn't really matter, Python's tail recursion is basically crap anyway)
doubleFactorial_cache = {0: 1, 1: 1, 2: 2}
def doubleFactorial(n, acc = 1, isCache = None):
    if(n == 0 or n == 1):
        return 1
    elif(n == 2):
        return 2

    if(acc != 1):
        while(n != 0 and n != 1):
            # print("* doubleFactorial diag: intermediate result = ", acc, flush=True)
            acc = acc * n
            n = n - 2
        return acc

    if(n > 250 and isCache == None):
        for i in range(124, n//2):
            doubleFactorial(i*2+n%2, isCache=True)

    cache = doubleFactorial_cache.get(n, None)
    if(cache == None):
        doubleFactorial_cache[n] = n * doubleFactorial(n - 2, isCache=isCache)
        #print("* doubleFactorial diag: cache missed n = ", n, flush=True)
    return doubleFactorial_cache[n]

# Utility: FactorialFromTo
# If from > to (this is an acceptable usage.), then the inverse is given and precisely calculated, float-
first
# If acc (accumulator) is given, then the result is multiplied from the acc.
def factorialFromTo(a, b, acc = 1):
    result = acc
    if(a == b):
        return a

    if(a < b):
        for i in range(a + 1, b + 1):
            result = result * i
    elif(a > b):
        for i in range(b + 1, a + 1):
            result = result / i
            # print("* factorialFromTo diag: intermediate result at i = ", i, " result = ", result,
flush=True)

    return result

# LegendrePolyNA (Non-Associated)
# With Caching w/ Pre-Compilation for large N (supports N up to 10000 as tested)
legendrePolyNA_cache = {0: {0: 1}, 1: {0: 0}}
def legendrePolyNA(n, x, isCache = None):
    global legendrePolyNA_cache
    # By definition (don't use this, it's completely not good)
    # return 1/(2**n * factorial(n)) * derivF_c_cycle(lambda x: (x**2 - 1)**n, x, n)

    # Using the recursive relation for Pn(x):
    # (n)P_n = (2n-1)xP_n-1 - (n-1)P_n-2, P0=1, P1=x
    # we can have better results..

    # If n is very large, maximum Recursion Depth will throw an error
    # so a cache should be built first, gradually
    if(n > 500 and isCache == None):
        for i in range(249, n//2):
            # print("* legendrePolyNA diag: building cache for n = ", n, isCache, flush=True)
            legendrePolyNA(i*2, x, True)
    if(n == 0):
        return 1
    elif(n == 1):
        return x

    cache = legendrePolyNA_cache.get(n, None)
    if(cache == None):
        legendrePolyNA_cache[n] = {}
        # print("* legendrePolyNA diag: cache missed for n = ", n, flush=True)
        legendrePolyNA_cache[n][x] = (2*n-1)/n*x*legendrePolyNA(n-1, x, isCache) - (n-
1)/n*legendrePolyNA(n-2, x, isCache)
    else:
        cache = legendrePolyNA_cache[n].get(x, None)
        if(cache == None):
            # print("* legendrePolyNA diag: cache missed for n = ", n, flush=True)
            legendrePolyNA_cache[n][x] = (2*n-1)/n*x*legendrePolyNA(n-1, x, isCache) - (n-
1)/n*legendrePolyNA(n-2, x, isCache)

    # print("* legendrePolyNA diag: n = ", n, ", x = ", x, flush=True)
    return legendrePolyNA_cache[n][x]

# LegendrePolyA (Associated Legendre Polynomial)
#     m
#   P    = ...
#     l
# With Caching w/ Automatic Pre-Compilation
# Caching structure: _cache[l][m][x]
#
# An improvement in 10-24-2017 uses Heuristic methods for choosing a recursion strategy, but it is not
functional yet.
# Improved 10-24-2017: Supports m < 0 using factorialFromTo
legendrePolyA_cache = {}
def legendrePolyA(l, m, x, isCache = None):
    global legendrePolyA_cache

    # Define a few existing formulae we can calculate directly,
    # helpful for ending recursive results
    if(abs(m) > l):
        return 0

    if(m == 0):
        return legendrePolyNA(l, x)
    elif(m < 0):
        return (-1)**m * factorialFromTo(l-m, l+m, legendrePolyA(l, (-1)*m, x, isCache=isCache))

    if(l == m):
        return (-1)**l * doubleFactorial(2*l - 1) * ((1 - x**2)**(l / 2))
    #if(l-1 == m):
    #   return x*(2*l-1) * legendrePolyA(l-1, l-1, x, isCache)

    if(m == 1):
        if(l == 1):
            return (-1)*(1-x*x)**(1/2)
        if(l == 2):
            return (-1)*3*x*(1-x*x)**(1/2)

    # print("* legendrePolyA diag: l = ", l, ", m = ", m, ", x = ", x, ", isCache = ", isCache,
flush=True)
    # Traditional Recursion:
    # This is a bit slow because it only decrements L
    # (2*l-1)/(l-m)*x*legendrePolyA(l-1, m, x) - (l+m-1)/(l-m)*legendrePolyA(l-2, m, x)
    # But it works for small L very nicely with higher precision,
    # also it is used for caching on large L
    # (2*l-1)/(l-m)*x*legendrePolyA(l-1, m, x) - (l+m-1)/(l-m)*legendrePolyA(l-2, m, x)

    # Pre-compiled Caching is used
    # to reduce recursion exponential depth growth.
    if(l > 25 and isCache == None):
        for i in range(24, l):
            # print("* legendrePolyA diag: building cache for l = ", i, " m = ", m, isCache, flush=True)
            legendrePolyA(i, m, x, True)


    # Another Recursion does incrementing & incrementing of M, while decrementing L
    # smartly, making use of legendryPolyNA,
    # used when M is sufficiently large.
    # and the fact that Plm=0 when |m|>l
    # ((1-x**2)**(1/2))*(-1)/(2*m)*(legendrePolyA(l-1, m+1, x)+(l+m-1)*(l+m)*legendrePolyA(l-1,m-1,x))

    # Heuristic for L/M-combo-values (if sufficiently large, use different strategy)
    #if(abs(l - m) < 3):
    #   # strategy = "dLiM" # decrement L, incrementing M *BRANCHES OUT EXPONENTIALLY
    #else:
    #   strategy = "dL_M" # decrement L only
    strategy = "dL_M"

    cache = legendrePolyA_cache.get(l, None)
    if(cache == None): # w/o l
        legendrePolyA_cache[l] = {}
        legendrePolyA_cache[l][m] = {}
        if(strategy == "dL_M"):
            legendrePolyA_cache[l][m][x] = (2*l-1)/(l-m)*x*legendrePolyA(l-1, m, x, isCache) - (l+m-
1)/(l-m)*legendrePolyA(l-2, m, x, isCache)
        elif(strategy == "dLiM"):
            legendrePolyA_cache[l][m][x] = ((1-x**2)**(1/2))*(-1)/(2*m)*(legendrePolyA(l-1, m+1, x)+(l+m-
1)*(l+m)*legendrePolyA(l-1,m-1,x))
        # print("* legendrePolyA diag: cache missed l = ", l, " strategy=", strategy, flush=True)
    else: # w l
        cache = legendrePolyA_cache[l].get(m, None)
        if(cache == None): # w/o m
            legendrePolyA_cache[l][m] = {}
            if(strategy == "dL_M"):
                legendrePolyA_cache[l][m][x] = (2*l-1)/(l-m)*x*legendrePolyA(l-1, m, x, isCache) - (l+m-
1)/(l-m)*legendrePolyA(l-2, m, x, isCache)
            elif(strategy == "dLiM"):
                legendrePolyA_cache[l][m][x] = ((1-x**2)**(1/2))*(-1)/(2*m)*(legendrePolyA(l-1, m+1,
x)+(l+m-1)*(l+m)*legendrePolyA(l-1,m-1,x))
            # print("* legendrePolyA diag: cache missed l = ", l, ", m = ", m, flush=True)
        else:
            cache = legendrePolyA_cache[l][m].get(x, None)
            if(cache == None): # w/o x
                if(strategy == "dL_M"):
                    legendrePolyA_cache[l][m][x] = (2*l-1)/(l-m)*x*legendrePolyA(l-1, m, x, isCache) -
(l+m-1)/(l-m)*legendrePolyA(l-2, m, x, isCache)
                elif(strategy == "dLiM"):
                    legendrePolyA_cache[l][m][x] = ((1-x**2)**(1/2))*(-1)/(2*m)*(legendrePolyA(l-1, m+1,
x)+(l+m-1)*(l+m)*legendrePolyA(l-1,m-1,x))
                # print("* legendrePolyA diag: cache missed l = ", l, ", m = ", m, ", x = ", x, ",
isCache = ", isCache, flush=True)
            #else:
            #   print("* legendrePolyA diag: cache hit")

    # print("* legendrePolyNA diag: n = ", n, ", x = ", x, flush=True)
    return legendrePolyA_cache[l][m][x]

    # Given the identity:
    # (l-m)P(m,l) = (2l-1)xP(m,l-1) - (l+m-1)P(m,l-2)
    # We can lower the l-parameters until it is equal to m
    # return ((1-x**2)**(1/2))*(-1)/(2*m)*(legendrePolyA(l-1, m+1, x)+(l+m-1)*(l+m)*legendrePolyA(l-1,m-
1,x))

    # By definition:
    # return (1 - x**2)**(m/2) * derivF_c_cycle(lambda x: legendrePolyNA(l, x), x, m)

####################################################################
# Computing Spherical Harmonics
####################################################################
pi = 3.141592653589793
# Utility: Cosine Function
# Uses Taylor Series Approximation for cosine
# Depends on:
#    factorial
# FIXME: 64 items is also an arbitrary precision limit.
# Improved 10-23-2017: If there is a modulo 2pi, remove it for higher
#   precision.
def cos(theta):
    result = 1
    theta  = abs(theta) # symmetry helps
    while(theta > 2*pi):
        theta += (-1)*2*pi
    if(abs(theta-pi/2) < 1e-11 or abs(theta - 3*pi/2) < 1e-11):
        return 0
    elif(abs(theta-pi) < 1e-11):
        return (-1)
    elif(abs(theta) < 1e-11 or abs(theta-2*pi) < 1e-11):
        return 1

    if(theta > pi):
        return (-1) * cos(theta - pi) # reduce...
    if(theta > pi/2):
        return (-1) * sin(theta - pi/2)

    for i in range(1, 64):
        result += (-1)**i * theta**(i*2) / factorial(i*2)
    return result

# Utility: Sine Function
# Uses Taylor Series Approximation for cosine
# Depends on:
#     factorial
# FIXME: 64 items is also an arbitrary precision limit.
# Improved 10-23-2017: If there is a modulo 2pi, remove it for higher
#    precision.
def sin(theta):
    result = 0
    if(theta < 0):
        while(theta < (-1)*2*pi):
            theta += 2*pi
    else:
        while(theta > 2*pi):
            theta += (-1)*2*pi

    for i in range(1, 64):
        result += (-1)**(i+1) * theta**(i*2-1) / factorial(i*2-1)
    return result

# Utility: Exponential Function
# Uses Taylor Series Expansion
# Depends on:
#     factorial
# FIXME: 32 items an arbitrary precision limit
# Improved 10-23-2017: For purely imaginary inputs, the algorithm uses Euler.
def exp(z):
    if(z.imag != 0 and z.real == 0):
        return cos(z.imag) + 1j*sin(z.imag)
    else:
        result = 1
        for i in range(1, 32):
            result += z**i / factorial(i)
        return result

# _irSpFactorial(l)
# Special Heuristic Shim for M=L-1
# Supports large L, up to 1M
def _irSpFactorial(l):
    # (2l-3)!!(2l-3)!!/(2l-1)! = ((2l-3)(2l-5)...)^2/(2l-1)(2l-2)(2l-3)...
    # u*u/(u+2)
    result = 1
    ptr2 = 2*l - 3
    ptr1 = 2*l - 1
    while(ptr2 != -1):
        result = result * ptr2 / ptr1
        if(ptr1 % 2 == 0):
            ptr2 = ptr2 - 2
        ptr1 = ptr1 - 1
    return result

# SphericalHarmonics(l, m, theta, phi)
# Computes spherical harmonics Ylm(theta, phi) for given parameters.
# Depends on the following functions:
#     legendrePolyA
#     cos (cosine function)
#     exp (exponential function)
#
```

```
# Notes:
#    ir prefix means "Intermediate Result" (Hungarian Notation)
def SphericalHarmonics(l, m, theta, phi):
    # Special Heuristic Shim for large M
    if(m == l-1):
        # print("* SphericalHarmonics diag: strategy 4 for irRS shimming")
        x = cos(theta)
        irRS = (2*l+1)/(4*pi)*_irSpFactorial(l)*x*x*(2*l-1)**2*(1-x**2)**(l-1)
        return (-1)**(l-1) * (-1 if x < 0 else 1) * (irRS)**(1/2) * exp(1j * m * phi)

    # factorial(l-m)/factorial(l+m) replace with 1/factorialFromTo(l-m, l+m) should be more accurate
    lPA = legendrePolyA(l, m, cos(theta))

    # power 1/2 of complex numbers / negative numbers is completely inaccurate in Python,
    # reaching absurd levels of error in the Real part (up to 1/2 of number)
    # Which is why we need a "safe", official, sanctioned square root.
    if(lPA < 0):
        lPA_ssqrt = ((-1)*lPA)**(1/2) * 1j
    else:
        lPA_ssqrt = lPA**(1/2)

    # print("* SphericalHarmonics diag: l=", l, " m=", m, " theta=", theta, " phi=", phi, flush=True)
    # print("* SphericalHarmonics diag: lPA intermediate result lPA=", lPA, " (sqrt)=", lPA_ssqrt,
flush=True)


    #if(m > 50): # Heuristic for factorial explosion and custom accumulator
    #    ilr_factorialFromTo = factorialFromTo(l-m, l+m)
    #    if(ilr_factorialFromTo == 1e500): # +infty

    # Rely on some heuristics depending on intermediate results,
    # depending on what we obtain as the intermediate legendrePolyA result (e+150 is the maximum)
    if(abs(lPA) > 1e+150):
        irBR = 1
        irRR = 1
        irRS = factorialFromTo(l+m, l-m, (2*l+1) * lPA * irBR)/(4*pi)

        if(irRS == 0):
            # Underflow
            # Extract an exponent from lPA... for the maximum acceptable
            # This is to ensure there's no underflow at factorialFromTo
            # By "borrowing" an accumulator based on irBR (BoRrow), then Return it (irRR)
            # from the final result.
            irBE = int((1e+301 / lPA).__str__().split("e+")[1])
            irBE += irBE % 2 - 2 # convert to an even number for sqrt-ng
            irBH = irBE // 2
            irBR = float("1e+" + irBE.__str__())
            irRR = float("1e-" + irBH.__str__())
            irRS = factorialFromTo(l+m, l-m, (2*l+1) * lPA * irBR)/(4*pi)
            # print("* SphericalHarmonics diag: lPA strategy 3, irBR =", irBR, flush=True)

        if(irRS < 0):
            irRS_ssqrt = ((-1)*irRS)**(1/2) * irRR * 1j
        else:
            irRS_ssqrt = irRS**(1/2) * irRR

        return irRS_ssqrt * lPA_ssqrt * exp(1j * m * phi)
    #elif(abs(lPA) < 1e-50 and m > 55): # Factorials grow large VERY fast and accumulators have to be
used
    #    return (factorialFromTo(l+m, l-m, (2*l+1) * 1e300)/(4*pi))**(1/2) * lPA * 1e-300 * exp(1j * m *
phi)
    else:
        irRS = factorialFromTo(l+m, l-m, (2*l+1) * lPA**2)/(4*pi)
        return (irRS)**(1/2) * (1 if lPA > 0 else (-1)) * exp(1j * m * phi)


#import time
#start_time = time.time()
#elapsed_time = time.time() - start_time
#print("Execution time is ", elapsed_time, " seconds.")

l = 1000

print(SphericalHarmonics(l, 1, pi/1000, pi/6))
print(SphericalHarmonics(l, 1, 3*pi/10, pi/6))
print(SphericalHarmonics(l, 1, 501*pi/1000, pi/6))
print("\n");
print(SphericalHarmonics(l, int(l/100), pi/1000, pi/6))
print(SphericalHarmonics(l, int(l/100), 3*pi/10, pi/6))
print(SphericalHarmonics(l, int(l/100), 501*pi/1000, pi/6))
print("\n");

print(SphericalHarmonics(l, int(l/10), pi/1000, pi/6))
print(SphericalHarmonics(l, int(l/10), 3*pi/10, pi/6))
print(SphericalHarmonics(l, int(l/10), 501*pi/1000, pi/6))
print("\n");

print(SphericalHarmonics(l, l-1, pi/1000, pi/6))
print(SphericalHarmonics(l, l-1, 3*pi/10, pi/6))
print(SphericalHarmonics(l, l-1, 501*pi/1000, pi/6))
print("\n");
```

作者声明该子程序针对特定的边界情况进行了 *shimming*.

计算物理学 “大作业” 文档 (五)

二维, 二阶常微分方程的数值求解

【题目】

（五）【10分】炮弹发射的初始速率为$v_0 = 700$米/秒，考虑三种发射角度，即30，40，50度：(1) 若考虑空气阻力$F = -B_2 v^2$，给定B_2/M为$4 \times 10^{-5} M^{-1}$，其中M为炮弹的质量，请编写程序数值计算炮弹在竖直x-y平面内运动的轨迹，直至炮弹落地，并分别给出三种发射角下落地时的时间、速度、以及距发射点的直线距离x_f。(2) 若不考虑空气阻力(设$B_2 = 0$即可)，请重复(1)的计算。将(1)、(2)两种情况的炮弹轨迹画在同一张图上。

【编写思路与算法描述】

该题的基本数值问题就是求解常微分方程组的初值问题. 因为炮弹处在二维平面上, 因而需要分两个方向处理问题, 设竖直方向 y, 水平方向 x, 初始点为 (0, 0). 我们可以将问题化简成如下的二阶常微分方程组:

$$\begin{cases} \ddot{y} = -g - B_2 v^2 \hat{y} \\ \ddot{x} = -B_2 v^2 \hat{x} \end{cases}$$

事实上高阶常微分方程(组)的求解方法是类似一阶的情况的, 在过程中还可以求出各部分的导数值. 因为我们的输入参量是加速度, 首先我们可以求解如下的常微分方程组:

$$\begin{cases} \dfrac{dv_y}{dt} = -g - B^2 v^2 \widehat{v_y} = -g - B_2 \sqrt{v_x^2 + v_y^2} * v_y \\ \dfrac{dv_x}{dt} = -B^2 v^2 \widehat{v_x} = -B_2 \sqrt{v_x^2 + v_y^2} * v_x \\ v_y(t=0) = v_0 sin\theta \\ v_x(t=0) = v_0 \cos\theta \end{cases}$$

然后再积分得到位置就可以了.

因为第二题中我们编写了 “Euler 方法” 求解常微分方程, 我们只需要对这个一元的情形做一些改动来支持 x, y 方向值的耦合就可以. 从而从原先的 eulerForwardSolve 函数中, 可以派生出耦合版 eulerForwardSolveXY:

```
    List eulerForwardSolve(lambda x, y: f(x, y), a, b, y0, dt = None)
    List eulerForwardSolveXY(lambda x, yx, yy: fx(x, yx, yy), lambda x, yy, yx: fy(x, yy, yx), a, b,
yx0, yy0, dt = None)
```

注意两个 lambda 函数的参量顺序依赖于主参量. eulerForwardXY 是一次循环同步进行的, 效率和 eulerForwardSolve 基本一致. 注意区分 “x” 和 “yx”, “x” 是指的本题目中的 t.

其他的问题就比较好解决了. 由于题目的限制, 我们需要自行编写 sin, cos 函数并进行角度-弧度转换. 其中三角函数我们已经在第四题中完成了编写, 然后 deg2Rad 可以乘上预计算的系数 0.017453292519943295 完成.

具体的实现请参照源代码.

【算法性能】

在给定的数据范围内近实时运算完毕. Euler 迭代法求解常微分线性方程组的过程的复杂度是线性的.

【运行方式】

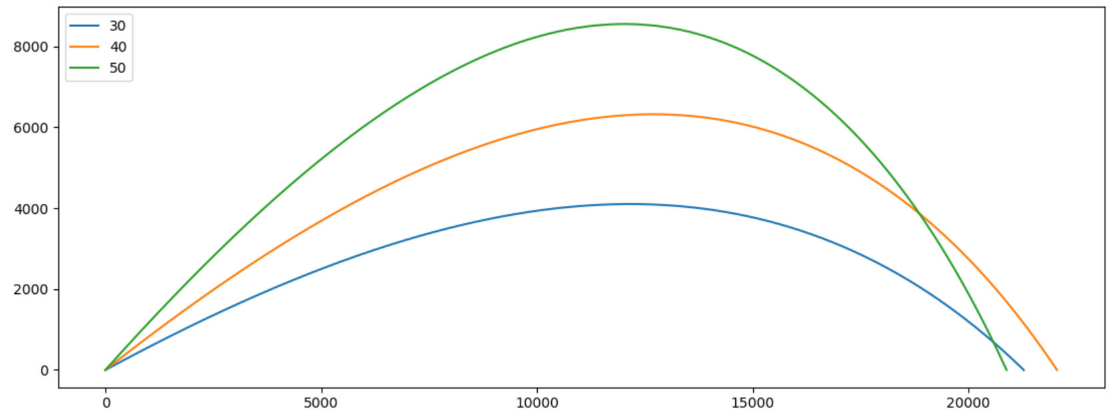
第五题的内容内置在源代码内, 可以自由调整参数. 运行一次可以完成 30, 40, 50° 的计算, 但需要更改源代码中的 boolean 参量 *hasFriction* 为 False 来关闭摩擦力 (即 (2) 问).

绘图需要用外部程序完成, 代码中也内置了利用 *matplotlib* 绘图的样例.

【运行结果】

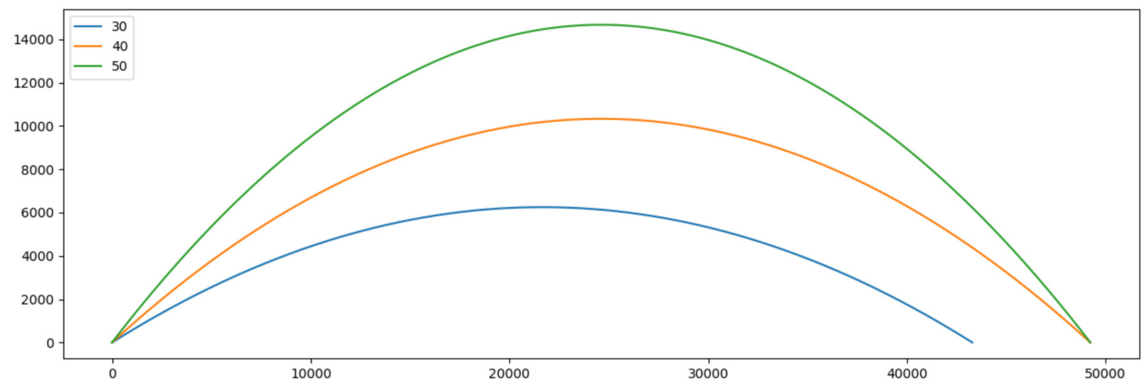
第 (1) 问 (存在非零摩擦力系数)

```
Calculating for theta =  30 deg
* Fall time t = 57.365000000010895 s
* Velocity vx = 238.94932652079044 m/s, vy = -248.37334798779113 m/s
* Speed v = 344.65359512908566 m/s
* Distance xf = 21280.926715367234 m

Calculating for theta =  40 deg
* Fall time t = 71.24500000000769 s
* Velocity vx = 187.42509693281272 m/s, vy = -299.69275913029117 m/s
* Speed v = 353.4740681229685 m/s
* Distance xf = 22051.944101112123 m

Calculating for theta =  50 deg
* Fall time t = 83.03999999999697 s
* Velocity vx = 145.64680783942504 m/s, vy = -338.917575594995 m/s
* Speed v = 368.88767352813994 m/s
* Distance xf = 20882.609784250006 m
```



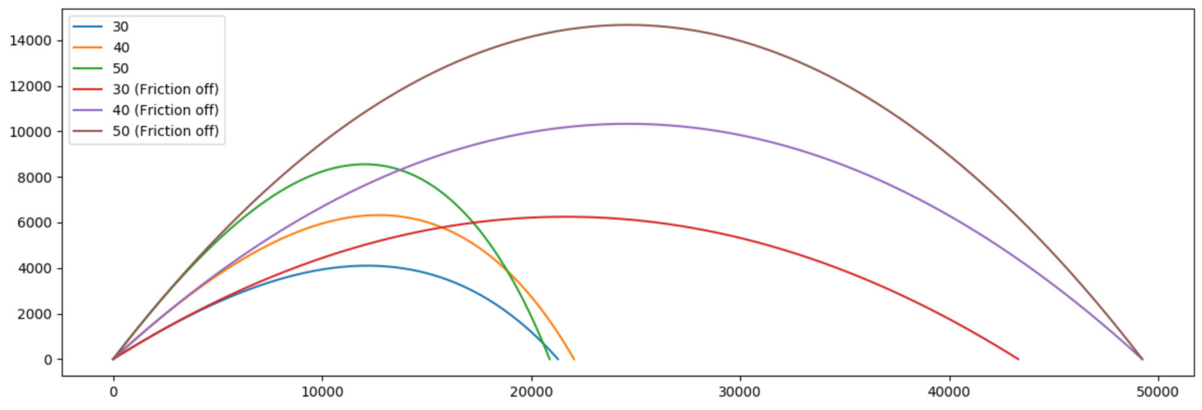
第 (2) 问 (摩擦力为零)

```
** WARNING: Friction set to 0 **
Calculating for theta =  30 deg
* Fall time t = 71.42500000000753 s
* Velocity vx = 606.217782649107 m/s, vy = -349.9649999999669 m/s
* Speed v = 699.9825006562498 m/s
* Distance xf = 43299.10512570805 m

Calculating for theta =  40 deg
* Fall time t = 91.82499999998898 s
* Velocity vx = 536.2311101832846 m/s, vy = -449.9336732193003 m/s
* Speed v = 699.9886526401767 m/s
* Distance xf = 49239.421692592136 m

Calculating for theta =  50 deg
* Fall time t = 109.42999999997296 s
* Velocity vx = 449.95132678057746 m/s, vy = -536.1828898165164 m/s
* Speed v = 699.9630617422555 m/s
* Distance xf = 49238.173689590985 m
```



所有图的坐标单位为 m.

将 (1) (2) 两种情况绘制在同一张图上的结果:



【源代码】

```
#####################################################################
# Computational Physics, 2017-18 Sem1
# HW-1 Ex-5
#
# (c) 2017 Haipeng Lin <linhaipeng@pku.edu.cn>
# All Rights Reserved.
#
# This program is written as Homework for the Computational Physics
# Course in Peking University, School of Physics.
# NO WARRANTIES, EXPRESS OR IMPLIED, ARE OFFERED FOR THIS PRODUCT.
# Copying is strictly prohibited and its usage is solely restricted
# to uses permitted by the author and homework grading.
#
# This program is Python 3 (3.6.2 on MSYS_NT)
# compatible, and does not support Python 2.
#
# Now Playing: My Heart Will Go On - Celine Dion
#              Not Another Song About Love - Hollywood Ending
#####################################################################

# The below is mostly adapted from the code in Exercise 2
# Solves a linear differential equation with an initial condition
# dy/dx = f(x, y)  a <= x <= b
# y(a) = y_0

# Euler Forward Method, for XY coupled situation
# eulerForwardSolve(\t, yx, yy -> f(t,yx), \t, yy, yx -> f(t,yy), a, b, x0, y0, dt = None)
# If timestep dt is none, then a default value will be used given N = 1000 partitions
# Returns the values for all the partitioned f(t_i) in a tuple (t_i, x_i, y_i)
#
# FIXME: default 1000 partitions is completely arbitrary
def eulerForwardSolveXY(fx, fy, a, b, x0, y0, dt = None):
    # Euler forward method
    k = 0
    if(dt != None):
        h = dt    # timestep given
    else:
        h = (b-a)/1000
    xn = a
    ys = [(a,x0,y0)]

    while((xn) <= b): # do a little nudging because python's double is a little too sensitive
        xn += h
        ys.append((xn, ys[-1][1] + h * fx(xn, ys[-1][1], ys[-1][2]), ys[-1][2] + h * fy(xn, ys[-1][2],
ys[-1][1])))

    return ys


# Euler Forward Method
# eulerForwardSolve(\x, y -> f(x,y), a, b, y0, dt = None)
# If timestep dt is none, then a default value will be used given N = 1000 partitions
# Returns the values for all the partitioned f(x_i) = y_i
#
# FIXME: 1000 partitions is completely arbitrary
def eulerForwardSolve(f, a, b, y0, dt = None, terminateCondition = None):
    k = 0
    if(dt != None):
        h = dt    # timestep given
    else:
        h = (b-a)/1000
    xn = a
    ys = [(a,y0)]

    while((xn) <= b): # do a little nudging because python's double is a little too sensitive
        xn += h
        ys.append((xn, ys[-1][1] + h * f(xn, ys[-1][1])))
        if(terminateCondition != None): # then is a lambda
            if(terminateCondition(xn, ys[-1][1]) == True):
                break

    return ys
```

```python
# The below is (mostly) copy-pasted from the code in Exercise 4
# Trigonometry Functions
pi = 3.141592653589793

# Utility: Factorial Function
# factorial(n) returns n! (for n in natural number)
# With Caching w/ Automatic Compilation (to avoid RecursionError)
factorial_cache = {0: 1, 1: 1}
def factorial(n, isCache = None):
    if(n == 0):
        return 1

    if(n > 500 and isCache == None):
        for i in range(249, n//2):
            factorial(i*2, True)

    cache = factorial_cache.get(n, None)
    if(cache == None):
        factorial_cache[n] = n * factorial(n - 1, isCache)
    return factorial_cache[n]

# Utility: Cosine Function
# Uses Taylor Series Approximation for cosine
# Depends on:
#     factorial
# FIXME: 64 items is also an arbitrary precision limit.
# Improved 10-23-2017: If there is a modulo 2pi, remove it for higher
#    precision.
def cos(theta):
    result = 1
    theta  = abs(theta) # symmetry helps
    while(theta > 2*pi):
        theta += (-1)*2*pi

    if(abs(theta-pi/2) < 1e-11 or abs(theta - 3*pi/2) < 1e-11):
        return 0
    elif(abs(theta-pi) < 1e-11):
        return (-1)
    elif(abs(theta) < 1e-11 or abs(theta-2*pi) < 1e-11):
        return 1

    if(theta > pi):
        return (-1) * cos(theta - pi) # reduce...
    if(theta > pi/2):
        return (-1) * sin(theta - pi/2)

    for i in range(1, 64):
        result += (-1)**i * theta**(i*2) / factorial(i*2)
    return result

# Utility: Sine Function
# Uses Taylor Series Approximation for sine
# Depends on:
#     factorial
# FIXME: 64 items is also an arbitrary precision limit.
# Improved 10-23-2017: If there is a modulo 2pi, remove it for higher
#    precision.
def sin(theta):
    result = 0
    if(theta < 0):
        while(theta < (-1)*2*pi):
            theta += 2*pi
    else:
        while(theta > 2*pi):
            theta += (-1)*2*pi

    for i in range(1, 64):
        result += (-1)**(i+1) * theta**(i*2-1) / factorial(i*2-1)
    return result

def deg2Rad(deg):
    return deg * 0.017453292519943295

## Exercise 5 Code
# Configure parameters below:
```

```
# degTheta = 30      # deg
v0 = 700            # m/s
hasFriction = False
fFactor = 4e-5      # M^(-1) as friction coefficient
g = 9.8             # m/s^(-2) as gravity constant
dt = 0.01           # dt as timestep
te = 10000          # Arbitrarily defined maximum time_end (not actually used, as lambda t, y: y <= 0 
termination rule will)
#                   # control the result from breaking the floor, heh

if(hasFriction == False):
    fFactor = 0
    print("** WARNING: Friction set to 0 **")

degThetas = [30, 40, 50]
#import matplotlib.pyplot as plt
#fig, ax = plt.subplots()
for i in range(len(degThetas)):
    degTheta = degThetas[i]
    print("Calculating for theta = ", degTheta, "deg")

    # Solve for dvy/dt, dvx/dt first
    vs = (eulerForwardSolveXY(lambda t, vx, vy: (-1) * fFactor * (vx**2 + vy**2) ** (1/2) * vx, lambda t, 
vy, vx: (-1) * g - fFactor * (vx**2 + vy**2) ** (1/2) * vy, 0, te, v0 * cos(deg2Rad(degTheta)), v0 * 
sin(deg2Rad(degTheta)), dt))

    # Then simply solve for x and y...
    # Translate vs to an appropriate data structure (dictionary)
    vxs = {}
    vys = {}
    for i in range(len(vs)):
        vxs[vs[i][0]] = vs[i][1]
        vys[vs[i][0]] = vs[i][2]

    xs_c = (eulerForwardSolve(lambda t, x: vxs[t], 0, te, 0, dt))
    ys_c = (eulerForwardSolve(lambda t, y: vys[t], 0, te, 0, dt, lambda t, y: y <= 0))

    xs = []
    ys = []

    for i in range(len(ys_c)):
        xs.append(xs_c[i][1])
        ys.append(ys_c[i][1])

    # Slice xs to fit ys...
    xs = xs[0:len(ys)]

    # Compute parameters as required by the exercise.
    print("* Fall time t =", xs_c[len(ys_c)-1][0], "s")
    print("* Velocity vx =", vs[len(ys_c)-1][1], "m/s, vy =", vs[len(ys_c)-1][2], "m/s")
    print("* Distance xf =", xs_c[len(ys_c)-1][1], "m")
    print("\n")

    #ax.plot(xs, ys, label=degTheta.__str__())

#plt.legend(loc='upper left')
#plt.show()
```

计算物理学 "大作业" 文档 (六)

正定对称矩阵的共轭梯度法解法

【题目】

（六）【10分】设有稀疏矩阵A，即对角元均为3，两个次对角元均为-1，除此三对角线之外在$(i, n+1-i)$ $(i = 1, 2, ..., n)$ 的位置元素为$\frac{1}{2}$，其余元素为0。形式如下：

$$A = \begin{bmatrix} 3 & -1 & 0 & \cdots & 0 & \frac{1}{2} \\ -1 & 3 & -1 & \cdots & \frac{1}{2} & 0 \\ 0 & -1 & 3 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & \frac{1}{2} & 0 & \cdots & 3 & -1 \\ \frac{1}{2} & 0 & 0 & \cdots & -1 & 3 \end{bmatrix} \tag{7}$$

设$b = (2.5, 1.5, \cdots, 1.5, 1.0, 1.0, 1.5, \cdots, 2.5)^T$。请用共轭梯度法求解线性方程组$Ax = b$，相邻两次的迭代误差要求小于$\varepsilon = 10^{-6}$。分别取$n = 100$和$n = 10000$两个算例。

【编写思路与算法描述】

首先需要制作一矩阵库, 方便进行共轭梯度法所需要的相关矩阵操作: 矩阵加减, 乘积, 数乘 (fmap). 这些功能可以独立于共轭梯度法的程序进行编写, 并进行封装, 我称之为 *pylitematrix* (之后的题目中也会利用到这个库.) 该库完全由自己编写, 函数操作模式类似于函数式编程语言 *Haskell*, 减小代码的副作用并使之数学表述更清晰.

这个封装完毕的矩阵库源代码见下附录 1.

其余部分需要 Implement 共轭梯度法. 注意, 共轭梯度法的具体写法可以有未优化和优化版, 在讲义中均有表述. 在下复述:

共轭梯度法的基本形式, 求解正定对称矩阵 Ax=b

给定初始向量 x_0, 第一步选择负梯度的方向为搜索方向, 即 $p_o = r_0$, 于是有:

$$\alpha_0 = \frac{\boldsymbol{r}_0^T \boldsymbol{r}_0}{\boldsymbol{p}_0^T \boldsymbol{A} \boldsymbol{p}_0}, \qquad \boldsymbol{x}_1 = \boldsymbol{x}_0 + \alpha_0 \boldsymbol{p}_0, \qquad \boldsymbol{r}_1 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_1$$

对于以后各步, 都类似进行以下的操作:

1) 计算搜索步长

$$\alpha_k = \frac{\boldsymbol{r}^T \boldsymbol{p}}{\boldsymbol{r}^T \boldsymbol{A} \boldsymbol{p}}$$

2) 更新解

$$\boldsymbol{x} = \boldsymbol{x} + \alpha \boldsymbol{p}$$

3) 计算新的残向量

$$\boldsymbol{r} = \boldsymbol{r} - \alpha \boldsymbol{A} \boldsymbol{p}$$

4) 计算新的搜索方向

$$\beta = -\frac{\boldsymbol{r}^T \boldsymbol{A} \boldsymbol{p}}{\boldsymbol{p}^T \boldsymbol{A} \boldsymbol{p}}$$
$$\boldsymbol{p} = \boldsymbol{r} + \beta \boldsymbol{p}$$

直到满足判停标准为止.

优化的共轭梯度法

共轭梯度法可以根据一些性质 (见讲义第 88 页) 进行如下的优化: (第 k 步)

$$\alpha_k = \frac{\boldsymbol{r}_k^T \boldsymbol{r}_k}{\boldsymbol{p}_k^T \boldsymbol{A} \boldsymbol{p}_k} \qquad \beta_{k-1} = \frac{\boldsymbol{r}_k^T \boldsymbol{r}_k}{\boldsymbol{r}_{k-1}^T \boldsymbol{r}_{k-1}}$$

这两种方法我们都可以分别书写 (源代码中为 conjGradient 和 conjGradient2 两个函数), 请参见源代码.

【算法性能】

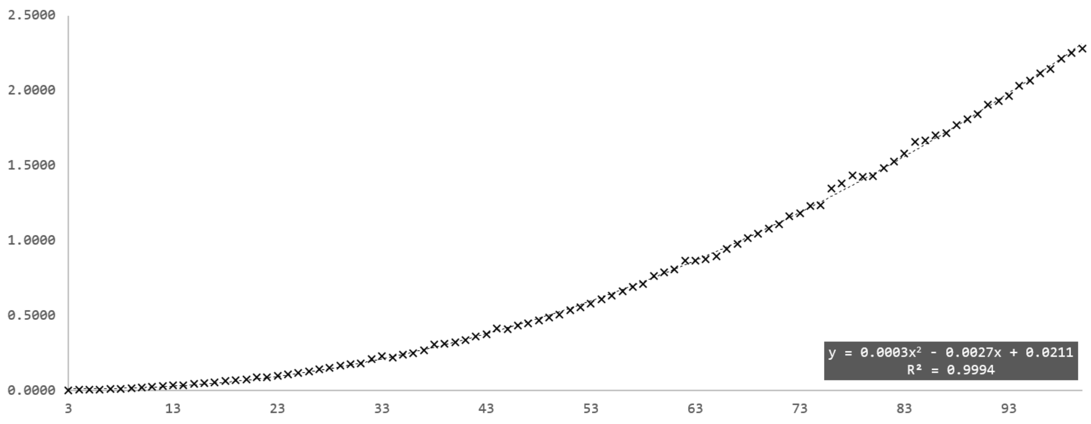
运行环境: Darwin 11.0, Python 3.6.2 AMD64 on Intel® Core i7-4770HQ.

运行时间: n = 100 为 2.297809362411499 秒钟, n = 10000 为 53137.34716796875 秒钟.

如果对于不同的 n 进行时间测试:

```
timings = []
for i in range(3, 101):
    start_time = time.time()
    ex6_2_noprint(i) # don't print
    elapsed_time = time.time() - start_time
    timings.append(elapsed_time)
    # print("Execution time is i=", i, " t=", elapsed_time, " seconds")
    print(elapsed_time)
```

可以得到如下 t – n 图, 可知算法的运算量约为 n^2.



对于本题目而言直接使用常规的矩阵乘法是很不划算的. N = 10000 的情况显然需要过多的时间, 这是由于我们的算法是 n^2 尺度导致的. 更合理的方法是通过发现**矩阵 A 是一个稀疏矩阵**, 因此它的乘法不需要使用点乘组合的方式完成, 而是编写一个特殊的函数对它进行运算. 这可以大大加快运算的速度, 但是我们选择不这样做, 因为这**打破了 "算法" 应该和 "数据" 相独立的界限**. 我们不当对特定的数据特征进行优化, 至少说我们在写共轭梯度法的算法的时候, 我们应该仅针对正定、对称矩阵性质进行讨论, 而不是对于矩阵 A 进行讨论.

【运行方式】

更改代码中的 ex8 函数参量指定 n, 直接运行即可.

【运行结果】

在精度范围内, 解为 $(1, \ldots, 1)^t$ (n 维). 精度的主要限制为 double 类型本身的精度, 一般迭代 20 次左右即可收敛到要求的 10^{-6} 的结果, 如果迭代 300 次左右即可收敛到 double 精度的最小误差限 (大约 19 位有效位数).

【源代码】

注意, 此源代码依赖于我所编写的 *pylitematrix* 矩阵库. 请根据附录 1 的代码参阅矩阵库的使用和编写细节, 一并阅读此代码.

```
########################################################################
# Computational Physics, 2017-18 Sem1
# HW-1 Ex-8
#
# (c) 2017 Haipeng Lin <linhaipeng@pku.edu.cn>
# All Rights Reserved.
#
# This program is written as Homework for the Computational Physics
# Course in Peking University, School of Physics.
# NO WARRANTIES, EXPRESS OR IMPLIED, ARE OFFERED FOR THIS PRODUCT.
# Copying is strictly prohibited and its usage is solely restricted
# to uses permitted by the author and homework grading.
#
# This program is Python 3 (3.6.2 on darwin)
# compatible, and does not support Python 2.
#
# Now Playing: Forgettable - Project 46
########################################################################

# Include my own lightweight matrix library
from pylitematrix.pylitematrix import Matrix

def matrixA(n):
    return Matrix(n, n, lambda i, j: -1 if (abs(j-i) == 1) else (3 if j == i else (0.5 if j == (n+1-i)
else 0)))

def vectorB(n):
    if(n % 2):
        return Matrix(n, 1, lambda m, _: 2.5 if(m == 1 or m == n) else (1.0 if (m == n//2+1) else 1.5))
    else:
        return Matrix(n, 1, lambda m, _: 2.5 if(m == 1 or m == n) else (1.0 if (m == n/2 or m == n/2+1)
else 1.5))

# The Conjugate Gradient Method, w/o Improved Algorithm
def conjGradient(A, b, x0):
    rc = b - A * x0
    if(rc.max() == 0):
        return x0
    pc = rc
    k  = 0
    x  = x0
    while(k < 1000):
        alpha_factor = (rc.transpose() * A * pc).elem(1, 1)
        if(alpha_factor == 0):
            break
        alpha = (rc.transpose() * pc).map(lambda x: x/alpha_factor).elem(1, 1)
        x = x + pc.map(lambda x: x * alpha)
        beta_factor  = (pc.transpose() * A * pc).elem(1, 1) * -1
        rc = rc - (A * pc).map(lambda x: x * alpha)
        if(beta_factor == 0):
            break
        beta = (rc.transpose() * A * pc).map(lambda x: x/beta_factor).elem(1, 1)
        pc = rc + pc.map(lambda x: x * beta)
        k = k + 1
    print("debug from conjGradient: did ", k, "iterations")
    return x


# The Conjugate Gradient Method, w/ Improved Algorithm
# Generally is 50% faster than conjGradient
def conjGradient2(A, b, x0):
    rc = b - A * x0
    if(rc.max() == 0):
        return x0
    pc = rc
    k  = 0
    x  = x0
    while(k < 1000):
        # print("*", end="", flush=True) # iterator diagnostic
        alpha_factor = (pc.transpose() * A * pc).elem(1, 1)
        if(alpha_factor == 0):
            break
        alpha = (rc.transpose() * rc).map(lambda x: x/alpha_factor).elem(1, 1)
        x = x + pc.map(lambda x: x * alpha)
        beta_factor  = (rc.transpose() * rc).elem(1, 1)
        rc = rc - (A * pc).map(lambda x: x * alpha)
        if(beta_factor == 0):
            break
        beta = (rc.transpose() * rc).map(lambda x: x/beta_factor).elem(1, 1)
        pc = rc + pc.map(lambda x: x * beta)
        k = k + 1
    print("debug from conjGradient2: did ", k, "iterations")
    return x

# Actual Code for Exercise 6
def ex6(n):
    print(conjGradient(matrixA(n), vectorB(n), Matrix(n, 1, lambda i, j: 0)))

def ex6_2(n):
    print(conjGradient2(matrixA(n), vectorB(n), Matrix(n, 1, lambda i, j: 0)))
```

计算物理学 “大作业” 文档 (七)

带状矩阵的直接解法

【题目】

（七）【10分】课堂上讲过，带宽为$2m+1$的对称正定带状矩阵A，可以不必选主元分解为$A=LDL^T$，其中L为下半带宽为m的带状阵。请根据课堂上讨论的一般对称正定带状系数矩阵方程组$Ax=b$的直接解法编写程序，注意计算时空复杂度要做到最小。给定对阵带状矩阵A，$m=2$，其非零元素: $a_{11}=a_{nn}=5$；$a_{ii}=6\ (i=2,\dots,n-1)$；$a_{i,i-1}=4\ (i=2,\dots,n)$；$a_{i,i-2}=1\ (i=3,\dots,n)$。右端向量$b=(60,120,\dots,120,60)^T$。请取$n=100$和10000两个算例。

【编写思路与算法描述】

为了得到更直接的解题思路, 不如我们将矩阵写出来: (未标注的元素为 0)

$$A=\begin{pmatrix}5&&&&\\4&6&&&\\1&4&\ddots&&\\&1&4&6&\\&&1&4&5\end{pmatrix}$$

所以如果我们要求解

$$\begin{pmatrix}5&&&&\\4&6&&&\\1&4&\ddots&&\\&1&4&6&\\&&1&4&5\end{pmatrix}\begin{pmatrix}x_1\\x_2\\\vdots\\x_{n-1}\\x_n\end{pmatrix}=\begin{pmatrix}60\\120\\\vdots\\120\\60\end{pmatrix}$$

事实上可以直接观察到

$$x_1=\frac{60}{5}=12$$

然后就可以求解

$$4x_1+6x_2=120\Rightarrow x_2=\cdots$$

所以事实上, 这个问题确实是可以直接求解的.

我们需要做到时空复杂度最小. 根据讲义 2 中的 “最小” 存法, 我们可以用三个数组储存 A 的三条对角线. 选用这样三个数组:

- $M[i]\equiv a_{ii}\ (i=1,2,3,\dots n)$ (注意我们舍去了元素 0, 这是一个固定的空间负担, 不需要担心.)
- $A[i]\equiv a_{i,i-1}\ (i=2,3,\dots n)$ (同上)
- $B[i]\equiv a_{i,i-2}\ (i=3,4,\dots n)$

每次循环只需要计算

$$B[i]X[i-2]+A[i]X[i-1]+M[i]X[i]=120\ (i=2,3,\dots,n-1)$$

我们可以把 $X[i]$ 储存在 $M[i]$ 中, 因为之后不会再用到这些主元素了. 所以空间复杂度可以做到 O(3n), 时间复杂度 O(n).

【运行方法】

设定文件中的 n 参数, 直接运行即可输出结果.

【算法效率】

上述已分析, 空间复杂度 O(3n) = O(n), 时间复杂度 O(n).

在给定的数据样本下可实时运算.

【运行结果】

对 n =100 的结果为

x = (12, 12, 10, 11.333..., 10.777...,10.925925925925926, 10.919753086419755, 10.89917695473251, 10.9139231824417, 10.907521719250115, 10.909090 ...., ..., 10.909090....)^t

(可以参见 *ex7-n-100.txt*)

对 n =10000 的结果请参见 *ex7-n-10000.txt*.

精度限制主要是 double 类型固有的精度.

【源代码】

```
########################################################################
# Computational Physics, 2017-18 Sem1
# HW-1 Ex-7
#
# (c) 2017 Haipeng Lin <linhaipeng@pku.edu.cn>
# All Rights Reserved.
#
# This program is written as Homework for the Computational Physics
# Course in Peking University, School of Physics.
# NO WARRANTIES, EXPRESS OR IMPLIED, ARE OFFERED FOR THIS PRODUCT.
# Copying is strictly prohibited and its usage is solely restricted
# to uses permitted by the author and homework grading.
#
# This program is Python 3 (3.6.2 on darwin)
# compatible, and does not support Python 2.
#
# Now Playing: Forgettable - Project 46
#              喜欢恋爱 - 卢巧音
########################################################################

n = 10000

# Initialize the matrices *using MATHEMATICAL INDEXES*
# This is to prevent index confusion.

# Primary Diagonal & X Solution Array
M = [6] * (n + 1)
M[0] = 0 # padding index
M[1] = 5
M[n] = 5

# Secondary Diagonal
A = [4] * (n + 1)
A[0] = 0 # non-significant padding index
A[1] = 0 # non-significant padding index

# Tertiary Diagonal
B = [1] * (n + 1)
B[0] = 0 # non-significant padding index
B[1] = 0 # non-significant padding index
B[2] = 0 # non-significant padding index

# Perform first step calculation
M[1] = 60 / M[1]
for i in range(2, n):
    # B[i]X[i-2]+A[i]X[i-1]+M[i]X[i]=120
    # X[i]=1/M[i] * (120-B[i]X[i-2]-A[i]X[i-1])
    M[i] = 1/M[i] * (120 - B[i] * M[i-2] - A[i] * M[i-1])
M[n] = 1/M[n] * (60 - B[n] * M[n-2] - A[n] * M[n-1])

print(M[1:])
```

计算物理学 “大作业” 文档 (八)

自然边界条件的样条插值方法

【题目】

（八）【10分】在飞机设计中，已知机翼下轮廓上数据如下，加工时，需要x每改变0.1米时的y值，利用自然边界条件，试用三次样条插值估计y的值，并画出轮廓曲线。

x	0	3	5	7	9	11	12	13	14	15
y	0	1.2	1.7	2.0	2.1	2.0	1.8	1.2	1.0	1.6

【编写思路与算法描述】

三次样条差值是一种分段三次多项式插值, 在每个插值节点处比分段三次 Hermite 更光滑, 具有二阶连续导数, 而且不需要节点导数信息. 根据《计算方法》(周铁等, 2006, 清华大学出版社), 三次样条差值方法的确切提法如下:

给定 [a, b] 区间上的 n+1 个互异的插值节点 x_i, 以及被插函数在这些节点上的函数值 $y_i = f(x_i)$, 求一个函数 $S_h(x)$, s.t.

(1) $S_h(x)$ 在每个区间 $[x_i, x_{i+1}]$ 上是不超过三次的多项式;

(2) $S_h(x_i) = y_i, i = 0, 1, 2, \dots, n$ (不需要导数信息)

(3) $S_h(x) \in C^2[a, b]$ (二阶连续)

$S_h(x)$ 的构造方法可以利用分段三次 Hermite 插值的基函数 $\alpha_i(x), \beta_i(x)$ 从而得到

$$S_h(x) = \sum_{i=0}^{n} \left(y_i \alpha_i(x) + m_i \beta_i(x)\right)$$

采用自然边界条件

$$S_h''(x_0) = S_h''(x_n) = 0$$

可以得到关于 m_i 的封闭线性代数方程组, 形式为 “强对角占优的三对角方程组”, 可以利用追赶法求解, 形式如下

$$\begin{bmatrix} 2 & \lambda_0 & 0 & \cdots & 0 \\ 1-\lambda_1 & 2 & \lambda_1 & & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & 1-\lambda_{n-1} & 2 & \lambda_{n-1} \\ 0 & \cdots & 0 & 1-\lambda_n & 2 \end{bmatrix} \begin{bmatrix} m_0 \\ m_1 \\ m_2 \\ \vdots \\ m_{n-1} \\ m_n \end{bmatrix} = \begin{bmatrix} \mu_0 \\ \mu_1 \\ \mu_2 \\ \vdots \\ \mu_{n-1} \\ \mu_n \end{bmatrix}$$

其中参数可以由下决定 (其中, $h_i = x_{i+1} - x_i$)

$$\lambda_0 = 1 \qquad \lambda_i = \frac{h_{i-1}}{h_{i-1} + h_i} \qquad \lambda_n = 0$$

$$\mu_0 = \frac{3(y_1 - y_0)}{h_0} \qquad \mu_i = 3\left(\frac{1-\lambda_i}{h_{i-1}}(y_i - y_{i-1}) + \frac{\lambda_i}{h_i}(y_{i+1} - y_i)\right) \qquad \mu_n = \frac{3(y_n - y_{n-1})}{h_{n-1}}$$

之前提到的 Hermite 基函数定义如下:

$$\alpha_0(x) = \begin{cases} \alpha_{\mathrm{L}}(x; x_0, x_1), & x \in [x_0, x_1], \\ 0, & x \in [x_1, x_n], \end{cases} \qquad \alpha_n(x) = \begin{cases} 0, & x \in [x_0, x_{n-1}], \\ \alpha_{\mathrm{R}}(x; x_{n-1}, x_n), & x \in [x_{n-1}, x_n], \end{cases}$$

$$\beta_0(x) = \begin{cases} \beta_{\mathrm{L}}(x; x_0, x_1), & x \in [x_0, x_1], \\ 0, & x \in [x_1, x_n], \end{cases} \qquad \beta_n(x) = \begin{cases} 0, & x \in [x_0, x_{n-1}], \\ \beta_{\mathrm{R}}(x; x_{n-1}, x_n), & x \in [x_{n-1}, x_n]. \end{cases}$$

$$\alpha_i(x) = \begin{cases} \alpha_{\mathrm{R}}(x; x_{i-1}, x_i), & x \in [x_{i-1}, x_i], \\ \alpha_{\mathrm{L}}(x; x_i, x_{i+1}), & x \in [x_i, x_{i+1}], \\ 0, & x \notin [x_{i-1}, x_{i+1}], \end{cases} \qquad 1 \leqslant i \leqslant n-1,$$

$$\beta_i(x) = \begin{cases} \beta_{\mathrm{R}}(x; x_{i-1}, x_i), & x \in [x_{i-1}, x_i], \\ \beta_{\mathrm{L}}(x; x_i, x_{i+1}), & x \in [x_i, x_{i+1}], \\ 0, & x \notin [x_{i-1}, x_{i+1}], \end{cases} \qquad 1 \leqslant i \leqslant n-1,$$

$$\alpha_{\mathrm{L}}(x;a,b) = \left(1 + 2\frac{x-a}{b-a}\right)\left(\frac{x-b}{a-b}\right)^2,$$
$$\alpha_{\mathrm{R}}(x;a,b) = \left(1 + 2\frac{x-b}{a-b}\right)\left(\frac{x-a}{b-a}\right)^2,$$
$$\beta_{\mathrm{L}}(x;a,b) = (x-a)\left(\frac{x-b}{a-b}\right)^2,$$
$$\beta_{\mathrm{R}}(x;a,b) = (x-b)\left(\frac{x-a}{b-a}\right)^2.$$

从而现在剩下的问题是求解强对角占优的三对角方程组. 选用 *Thomas* 算法 (参见 Wikipedia: “Tridiagonal matrix algorithm”, retrieved 10-25-2017) 求解这个方程组. 对于如下的线性方程组

$$\begin{bmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ 0 & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{bmatrix}$$

进行两次系数转化, 分别为

$$c'_i = \begin{cases} \dfrac{c_i}{b_i} & i = 1 \\ \dfrac{c_i}{b_i - a_i c'_{i-1}} & i = 2, 3, \dots, n-1 \end{cases} \qquad d'_i = \begin{cases} \dfrac{d_i}{b_i} & i = 1 \\ \dfrac{d_i - a_i d'_{i-1}}{b_i - a_i c'_{i-1}} & i = 2, 3, 4, \dots, n \end{cases}$$

然后进行一次逆向替换, 就可以得到解:

$$x_n = d'_n \qquad x_i = d'_i - c'_i x_{i+1} \quad (i = n-1, n-2, \dots, 1)$$

为了最终的代码更 “数学”, 我们将数组 0 元素搁置不用而从 1 开始标号, 这样指标就不需要转化了, 避免了很多书写习惯不统一的 bug. 最终源代码见本部分末.

【运行方式】

调用 spline 函数, 返回的是一个 *lambda*, 可以给定 x 得到 y. 参数如下:

lambda spline(xs, ys)

其中 xs 是 x 坐标集, ys 是 y 坐标集.

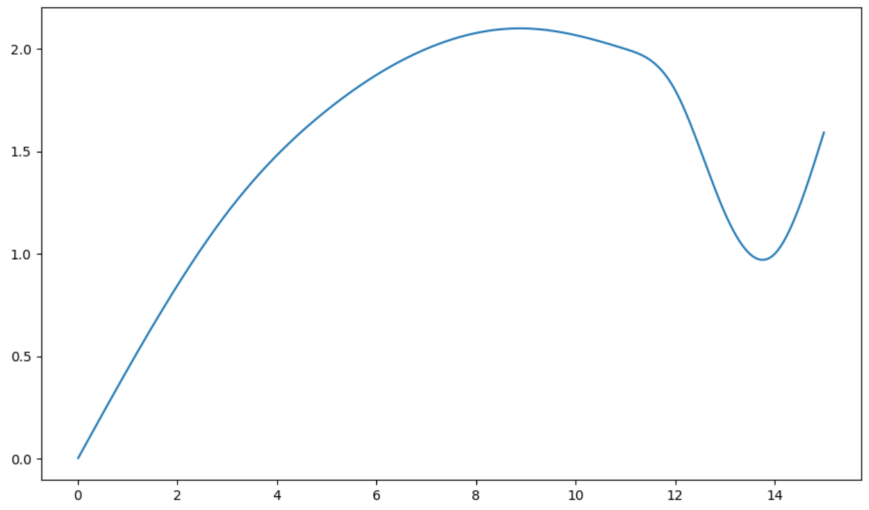
如果不接受使用 Python 的功能 *functools.partial*, 则可以直接使用 *spline_* 函数:

float spline_(xs, ys, x)

注意这个性能会更慢, 因为每次调用都会重新求解方程组.

【运行结果】

得到的函数轮廓如下图所示.



按照题目要求, 可以得出 x 变化 0.1 对应的 y 值:

```
x = 0.0            y = 0.0
x = 0.1            y = 0.04407260340885021
x = 0.2            y = 0.08811802592731877
x = 0.3            y = 0.13210908666502408
x = 0.4            y = 0.17601860473158443
x = 0.5            y = 0.21981939923661814
x = 0.6            y = 0.26348428928974366
x = 0.7            y = 0.30698609400057925
x = 0.8            y = 0.35029763247874335
x = 0.9            y = 0.3933917238338543
x = 1.0            y = 0.4362411871755302
x = 1.1            y = 0.47881884161338994
x = 1.2            y = 0.5210975062570515
x = 1.3            y = 0.5630500002161329
x = 1.4            y = 0.6046491426002534
x = 1.5            y = 0.6458677525190305
x = 1.6            y = 0.6866786490820829
x = 1.7            y = 0.7270546513990293
x = 1.8            y = 0.7669685785794871
x = 1.9            y = 0.8063932497330759
x = 2.0            y = 0.8453014839694127
x = 2.1            y = 0.8836661003981173
x = 2.2            y = 0.9214599181288067
x = 2.3            y = 0.9586557562710999
x = 2.4            y = 0.9952264339346156
x = 2.5            y = 1.0311447702289713
x = 2.6            y = 1.066383584263786
x = 2.7            y = 1.1009156951486778
x = 2.8            y = 1.1347139219932656
x = 2.9            y = 1.1677510839071663
x = 3.0            y = 1.2
x = 3.1            y = 1.2314412908653536
x = 3.2            y = 1.2620867830326923
x = 3.3            y = 1.2919561045154513
x = 3.4            y = 1.3210688833270656
x = 3.5            y = 1.3494447474809694
x = 3.6            y = 1.3771033249905982
x = 3.7            y = 1.4040642438693864
x = 3.8            y = 1.4303471321307692
x = 3.9            y = 1.455971617788181
x = 4.0            y = 1.4809573288550568
x = 4.1            y = 1.5053238933448316
x = 4.2            y = 1.52909093927094
x = 4.3            y = 1.5522780946468169
x = 4.4            y = 1.5749049874858974
x = 4.5            y = 1.5969912458016158
x = 4.6            y = 1.6185564976074076
x = 4.7            y = 1.6396203709167063
x = 4.8            y = 1.6602024937429487
x = 4.9            y = 1.6803224940995682
x = 5.0            y = 1.7
x = 5.1            y = 1.7192507827798167
x = 5.2            y = 1.738075187063141
x = 5.3            y = 1.7564697007962342
x = 5.4            y = 1.7744308119253562
x = 5.5            y = 1.7919550083967684
x = 5.6            y = 1.8090387781567312
x = 5.7            y = 1.8256786091515047
x = 5.8            y = 1.8418709893273508
x = 5.9            y = 1.8576124066305293
x = 6.0            y = 1.872899349007301
x = 6.1            y = 1.8877283044039266
x = 6.2            y = 1.902095760766667
x = 6.3            y = 1.915998206041783
```

```
x = 6.4            y = 1.9294321281755347
x = 6.5            y = 1.9423940151141832
x = 6.6            y = 1.9548803548039893
x = 6.7            y = 1.9668876351912128
x = 6.8            y = 1.9784123442221155
x = 6.9            y = 1.9894509698429577
x = 7.0            y = 2.0
x = 7.1            y = 2.01005557801538
x = 7.2            y = 2.0196124687147434
x = 7.3            y = 2.028665092299612
x = 7.4            y = 2.0372078689715094
x = 7.5            y = 2.045235218931957
x = 7.6            y = 2.0527415623824776
x = 7.7            y = 2.0597213195245936
x = 7.8            y = 2.0661689105598278
x = 7.9            y = 2.0720787556897022
x = 8.0            y = 2.077445275115739
x = 8.1            y = 2.0822628890394617
x = 8.2            y = 2.0865260176623917
x = 8.3            y = 2.0902290811860516
x = 8.4            y = 2.0933664998119643
x = 8.5            y = 2.095932693741652
x = 8.6            y = 2.097922083176637
x = 8.7            y = 2.099329088318442
x = 8.8            y = 2.100148129368589
x = 8.9            y = 2.100373626528601
x = 9.0            y = 2.1
x = 9.1            y = 2.0990269051586634
x = 9.2            y = 2.0974749380778857
x = 9.3            y = 2.095369930005317
x = 9.4            y = 2.0927377121886064
x = 9.5            y = 2.089604115875404
x = 9.6            y = 2.085994972313358
x = 9.7            y = 2.08193611275012
x = 9.8            y = 2.0774533684333383
x = 9.9            y = 2.0725725706106624
x = 10.0            y = 2.067319550529742
x = 10.1            y = 2.0617201394382265
x = 10.2            y = 2.055800168583767
x = 10.3            y = 2.0495854692140103
x = 10.4            y = 2.0431018725766084
x = 10.5            y = 2.0363752099192096
x = 10.6            y = 2.0294313124894634
x = 10.7            y = 2.0222960115350204
x = 10.8            y = 2.0149951383035285
x = 10.9            y = 2.0075545240426385
x = 11.0            y = 2.0
x = 11.1            y = 1.9922335248272014
x = 11.2            y = 1.9836615667915916
x = 11.3            y = 1.9735667215644572
x = 11.4            y = 1.9612315848170867
x = 11.5            y = 1.945938752220767
x = 11.6            y = 1.9269708194467856
x = 11.7            y = 1.903610382166431
x = 11.8            y = 1.87514003605099
x = 11.9            y = 1.8408423767717506
x = 12.0            y = 1.8
x = 12.1            y = 1.7522217135891476
x = 12.2            y = 1.6984211741210944
x = 12.3            y = 1.639838250359861
x = 12.4            y = 1.5777128110694694
x = 12.5            y = 1.5132847250139405
x = 12.6            y = 1.4477938609572956
x = 12.7            y = 1.382480087663558
x = 12.8            y = 1.3185832738967487
```

```
x = 12.9            y = 1.2573432884208884
x = 13.0            y = 1.2
x = 13.1            y = 1.1476796208162041
x = 13.2            y = 1.101053736724028
x = 13.3            y = 1.0606802769960961
x = 13.4            y = 1.0271171709050355
x = 13.5            y = 1.0009223477234706
x = 13.6            y = 0.9826537367240282
x = 13.7            y = 0.9728692671793342
x = 13.8            y = 0.9721268683620142
x = 13.9            y = 0.9809844695446942
x = 14.0            y = 1.0
x = 14.1            y = 1.029459803146033
x = 14.2            y = 1.068563878982792
x = 14.3            y = 1.1162406416557524
x = 14.4            y = 1.171418505310389
x = 14.5            y = 1.2330258840921764
x = 14.6            y = 1.299991192146591
x = 14.7            y = 1.3712428436191053
x = 14.8            y = 1.445709252655195
x = 14.9            y = 1.5223188334003348
x = 15.0            y = 1.6
```

【源代码】

注意, 此源代码依赖于我所编写的 *pylitematrix* 矩阵库. 请根据附录 1 的代码参阅矩阵库的使用和编写细节, 一并阅读此代码.

此源代码**可选**包含 Python 的函数 *functools.partial*, 可以允许 *spline* 函数返回 *lambda*, 而不需要事先给定 x 的值. 这不影响程序的核心功能, 因而, 如果不允许外接此功能, 则可以使用 *spline_*函数来代替.

```python
########################################################################
# Computational Physics, 2017-18 Sem1
# HW-1 Ex-8
#
# (c) 2017 Haipeng Lin <linhaipeng@pku.edu.cn>
# All Rights Reserved.
#
# This program is written as Homework for the Computational Physics
# Course in Peking University, School of Physics.
# NO WARRANTIES, EXPRESS OR IMPLIED, ARE OFFERED FOR THIS PRODUCT.
# Copying is strictly prohibited and its usage is solely restricted
# to uses permitted by the author and homework grading.
#
# This program is Python 3 (3.6.2 on darwin)
# compatible, and does not support Python 2.
#
# Now Playing: Remember the Name - Fort Minor
#
# Reticulating Splines requires the solving of a tri-diagonal matrix
# using whatever method of our choice. I have chosen Thomas' Algorithm,
# which is similar to 追赶法 because its easier and more lightweight
# to implement.
# The implementation here is similar to a previous
# masterpiece of mine written in Haskell,
# https://github.com/jimmielin/HW-Numerical-Analysis-Spring-2016/
#  blob/master/20160305-extrapolation/splines.hs
# but re-implemented in Python.
########################################################################

from functools import partial

# Include my own lightweight matrix library & its prototype
from pylitematrix.pylitematrix import Matrix, mp


########################################################################
# The Tridiagonal Matrix Algorithm "Thomas Algorithm"
# https://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm
#
# An algorithm stable for solving tri-diagonal, positive semi-definite
# or diagonally dominant matrices
# in O(n) instead of O(n^2) as Gauss.

# solveTri(M, b) = x solves Mx=b
# when M is tri-diagonal, positive semi-definite || diagonal dominant
#
# Notes of the author (10/05/2017):
# 1. This implementation IS space-optimal as the matrix is decomposed in
# the first step and stored in an internal representation similar to the
# course description (but since we are not symmetric, we use arrays)
# 2. Watch out for the indexes. Some have been padded for mathematical
# understanding, but some have not for implementation reasons.
#
def solveTri(M, b):
    # Extract the {c_i}, {b_i} and {a_i} in the NxN matrix
    n = M.nrows()
    if(M.ncols() != n):
        raise ValueError("Tridiagonal Matrix to-solve must be square.")

    # The arrays below have been padded so the indeces are mathematically sound.
    # One doesn't have to do this, but it makes for more coding convenience.
    A = [0, 0] # lower diagonal A, i=2,3..n
    B = [0] # diagonal B, i=1,2,..n
    C = [0] # upper diagonal C, i=1,2,..n-1
    D = b.toList() # right-b vector d1, d2, ... n
    D.insert(0, 0) # left-padd, * minor performance concern

    # The solution array is *NOT* padded, as it'll be used for
    # fromList_ by the Matrix prototype
    # It's even reversed as part of back-substitution
    # so watch out for the index.
    X = [None] * n

    # .elem is now O(1) so you can use that for safety,
    # but I'll tap into .internal represenation for speed and less overhead
    # since we already checked the indexes above
    for i in range(1, n+1):
        if(i != 1): # make sure i=1 case does not run for A
            A.append(M.internal[i-1][i-2])
        if(i != n): # make sure i=n case does not run for C
            C.append(M.internal[i-1][i])
        B.append(M.internal[i-1][i-1])

    # Now perform the Thomas Algorithm Forward Sweep
    C[1] = C[1] / B[1]
    D[1] = D[1] / B[1]
    for i in range(2, n+1):
        if(i != n): # make sure i=n case does not run for C=..n-1
            C[i] = C[i] / (B[i] - A[i] * C[i-1])
        D[i] = (D[i] - A[i] * D[i-1]) / (B[i] - A[i] * C[i-1])

    # Now do a back-subsitution...
    X[0] = D[n]
    for i in range(2, n+1):
        j = n+1-i # back indexes
        X[i-1] = D[j] - C[j] * X[i-2]

    X.reverse()

    return mp.fromList_(n, 1, X)

# Some testing data.
#testtrim = mp.fromList_(4, 4, [1,5,0,0,8,2,6,0,0,9,3,7,0,0,10,4])
#testtrib = mp.fromList_(4, 1, [1,1,1,1])
#print(solveTri(testtrim, testtrib))

####################################################################
# Reticulator of Splines
# “三次样条差值是一种分段三次多项式插值，在每个插值节点处比分段三次 Hermite 更光滑，
#  具有二阶连续导数，而且不需要节点导数信息.”
# 据王鸣老师所述，在早期工程学上三次样条插值的实现方法就是在坐标纸上画点，然后用
# 一个铁条固定在节点处弯曲，从而获得自然边界条件。
#
# 王教授于 2016 年夏因病逝世，我们成为最后一批《计算方法 B》的学生.
# R.I.P.
#

# Base Hermite Functions alpha, beta
# "分段三次 Hermite 插值的基函数"
#
# for partial applications we may need functools->partial, but this implementation
# is so much lamer than Haskell...
aL = lambda a, b, x: (1 + 2*(x-a)/(b-a)) * ((x-b)/(a-b))**2
aR = lambda a, b, x: (1 + 2*(x-b)/(a-b)) * ((x-a)/(a-b))**2
bL = lambda a, b, x: (x-a)*((x-b)/(a-b))**2
bR = lambda a, b, x: (x-b)*((x-a)/(a-b))**2

# Hermite Functions alpha_i, beta_i
# Generated Functions (Partially Applied Lambdas) that use aLRbLR to
# generate the respective alpha/beta functions for extrapolation
# According to 《计算方法》Page 40, Tsinghua University
#
# The Haskell Implementation is as follows:
#
# ah' :: (RealFloat a) => Int -> Int -> [a] -> a -> a
# ah' i n xs x
#     | i == 0 = if x < xs !! 0 then 0 else (if x <= xs !! 1 then aleft (xs !! 0) (xs !! 1) x else 0)
#     | i == n = if x < xs !! (n - 1) then 0 else (if x <= xs !! n then aright (xs !! (n-1)) (xs !! n) x
else 0)
```

```
#     | otherwise = if x < (xs !! (i - 1)) then 0 else (if x <= (xs !! i) then aright (xs !! (i - 1))
(xs !! i) x else (if x <= (xs !! i + 1) then aleft (xs !! i) (xs !! (i + 1)) x else 0))
#
# i = 0, 1, ... n (for a total of n+1 nodes)
# Indexes are mathematical since the mathematical definition starts with zero.
def aI(x, i, nodes):
    n = len(nodes) - 1
    if(n < 1):
        raise ValueError("There must be at least 2 nodes for Hermite Extrapolation (or any, actually.)")

    if(i == 0):
        return 0 if x < nodes[0] else (aL(a=nodes[0],b=nodes[1],x=x) if x < nodes[1] else 0)
    elif(i == n):
        return 0 if x < nodes[n-1] else (aR(a=nodes[n-1],b=nodes[n],x=x) if x <= nodes[n] else 0)
    else:
        return 0 if x < nodes[i-1] else (aR(a=nodes[i-1],b=nodes[i],x=x) if x < nodes[i] else
(aL(a=nodes[i],b=nodes[i+1],x=x) if x < nodes[i+1] else 0))

def bI(x, i, nodes):
    n = len(nodes) - 1
    if(n < 1):
        raise ValueError("There must be at least 2 nodes for Hermite Extrapolation (or any, actually.)")

    if(i == 0):
        return 0 if x < nodes[0] else (bL(a=nodes[0],b=nodes[1],x=x) if x < nodes[1] else 0)
    elif(i == n):
        return 0 if x < nodes[n-1] else (bR(a=nodes[n-1],b=nodes[n],x=x) if x <= nodes[n] else 0)
    else:
        return 0 if x < nodes[i-1] else (bR(a=nodes[i-1],b=nodes[i],x=x) if x < nodes[i] else
(bL(a=nodes[i],b=nodes[i+1],x=x) if x < nodes[i+1] else 0))

# Generate Equations
# Given xs, ys, generate the matrices to be solved in order to get the appropriate m-factors
# in the Splined Hermite representation Sh(x).
#
# Note: This is *not* very efficient as it stores a whole matrix.
# But it is generalized enough so we can abstract a solver out of the equation generator,
# so its for the greater good.
#
# Arrays xs and ys are mathematical indexes i=0..n
def genSplineMatrix(xs, ys):
    n = len(xs) - 1
    if(n != len(ys) - 1):
        raise ValueError("You must provide as many y-points as x-points to genSplineMatrix")

    M = mp.diag_(n+1, [2] * (n+1))

    # arrays lambdas & mus are mathematical indexes i=0..n
    lambdas = [None] * (n+1)
    lambdas[0] = 1
    lambdas[n] = 0
    M.setElem_(1, 2, lambdas[0])
    M.setElem_(n+1, n, 1)
    # h_i = x_i+1 - x_i

    mus = [None] * (n+1)
    mus[0] = 3 * (ys[1] - ys[0]) / (xs[1] - xs[0])
    mus[n] = 3 * (ys[n] - ys[n-1]) / (xs[n] - xs[n-1])

    for i in range(1, n):
        lambdas[i] = (xs[i] - xs[i-1]) / (xs[i+1] - xs[i-1])
        mus[i]     = 3 * ((ys[i] - ys[i-1]) * (1 - lambdas[i]) / (xs[i] - xs[i-1]) + (ys[i+1] - ys[i]) *
lambdas[i] / (xs[i+1] - xs[i]))
        # and commit using the side-effect setElem_ function
        M.setElem_(i+1,i+2,lambdas[i])
        M.setElem_(i+1,i,1-lambdas[i])

    B = mp.fromList_(n+1,1,mus)

    return (M, B)

# Reticulate Splines (Actual Code)
# Generate Equations, Solve Equations, Construct Functions, and returns a Lambda
# representing the splined function.
#
```

```python
# Luckily, Python does not do lazy evaluation, meaning that this lambda will be speedy
# instead of being tri-solved every call. :)
def spline_(xs, ys, x):
    gendMatrix = genSplineMatrix(xs, ys)
    mSolution  = solveTri(gendMatrix[0], gendMatrix[1]).toList() # ms...

    # To do a sum, we need parrying & foldl which Python does not do very well.
    # Hence we use functools.partial to do this in a tricky way,
    # making this spline_ function do the dirty work and wrapping it in a cleaner
    # spline function.
    # def aI(x, i, nodes) / bI
    res = 0
    for i, y in enumerate(ys):
        res += y * aI(x, i, xs) + mSolution[i] * bI(x, i, xs)

    return res

def spline(xs, ys):
    spp = partial(spline_, xs=xs, ys=ys)
    return lambda x: spp(x=x)

ex8 = spline([0,3,5,7,9,11,12,13,14,15], [0,1.2,1.7,2.0,2.1,2.0,1.8,1.2,1.0,1.6])

# For plotting the result
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
xs = [x/100 for x in range(1,1500)]
ys = [ex8(x) for x in xs]
ax.plot(xs, ys)
plt.show()

# For 0.1x variances
xs = [0,3,5,7,9,11,12,13,14,15]
for i in range(len(xs)):
    print("x =", xs[i] - 0.1, " y =", ex8(xs[i] - 0.1))
    print("x =", xs[i], " y =", ex8(xs[i]))
    print("x =", xs[i] + 0.1, " y =", ex8(xs[i] + 0.1))
```

计算物理学 “大作业” 文档 (九)

微积分的数值求解

【题目】

（九）【15分】球坐标系中氢原子的哈密顿算符为$H=-\frac{1}{2}\nabla^2-\frac{1}{r}$，归一化的束缚态波函数为：

$$\Psi(r,\theta,\phi)=\sqrt{\left(\frac{2}{n}\right)^3\frac{(n-l-1)!}{2n(n+l)!}}\mathrm{e}^{-\rho/2}\rho^l L_{n-l-1}^{2l+1}(\rho)Y_l^m(\theta,\phi), \tag{8}$$

其中$\rho=\frac{2r}{n}$，$L_{n-l-1}^{2l+1}(\rho)$为广义拉盖尔多项式，$Y_l^m(\theta,\phi)$为球谐函数。球坐标系中的拉普拉斯算符为：

$$\begin{aligned}\nabla^2 &= \frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial}{\partial r}\right)+\frac{1}{r^2\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial}{\partial\theta}\right)+\frac{1}{r^2\sin^2\theta}\frac{\partial^2}{\partial\phi^2} && (9)\\ &= \frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial}{\partial r}\right)-\frac{\hat{\mathcal{L}}^2}{r^2} && (10)\end{aligned}$$

试计算当$n=3,l=1,m=1$时的体系总能量E，要求误差不大于10^{-6}。【$E=\langle\Psi|H|\Psi\rangle$，$E_{311}$准确值为：$-\frac{1}{2n^2}=-\frac{1}{18}$】

提示： 角度部分的微积分解析求出，然后在$(0,60]$区间离散r，导数利用有限差分近似。以下公式对你有用：

$$\begin{aligned}E=\langle\Psi|H|\Psi\rangle &= \iiint\Psi^*(H\Psi)r^2dr\sin\theta d\theta d\phi && (11)\\ \hat{\mathcal{L}}^2Y_l^m(\theta,\phi) &= l(l+1)Y_l^m(\theta,\phi) && (12)\\ Y_1^1 &= -\sqrt{\frac{3}{8\pi}}\sin\theta e^{i\phi} && (13)\\ L_1^\alpha(x) &= -x+\alpha+1 && (14)\end{aligned}$$

【编写思路与算法描述】

我们首先需要求出 n = 3, l = 1, m =1 时的归一化束缚态波函数. 带入参量可得:

$$\Psi(r,\theta,\phi)=\frac{\sqrt{6}}{81}re^{-\frac{r}{3}}\left(-\frac{2r}{3}+4\right)Y_1^1(\theta,\phi)$$

我们还需要计算 $\Psi^*(H\Psi)$. 注意到

$$\iint Y_1^{1*}Y_1^1\,sin\theta d\theta d\phi=1$$

所以

$$E=\langle\Psi|\hat{H}|\Psi\rangle=\int_0^{+\infty}\frac{1}{2187}\left[-r^3e^{-\frac{2}{3}r}\left(-\frac{2r}{3}+4\right)\left(-\frac{2}{27}r^2+\frac{16}{9}r-8\right)-2r^3e^{-\frac{2}{3}r}\left(-\frac{2}{3}r+4\right)^2\right]dr$$

这个式子也可以进一步化简 (我们对函数进行化简方便输入, 也会简化计算. 并不需要使用 “提示” 中的 “有限差分近似”, 那样误差会过大), 求得

$$E=\int_0^{+\infty}-\frac{1}{2187}r^4e^{-\frac{2}{3}r}\left(\frac{4}{81}r^2-\frac{16}{27}r+\frac{16}{9}\right)dr$$

这样的话我们就可以利用数值积分的方法对这个积分进行求解.

对于数值积分的公式, 我们选取复化梯形公式 (Trapezoid Formula), 把积分区间进行 n 等分, 分别使用梯形求积分公式求积分, 最后对结果求和. 算法论述如下:

设 f(x) 在 [a, b] 上有连续的二阶导数, n 是正整数. 将 [a, b] 等分为 n 个小区间, 分别使用梯形公式, 最后叠加所有的区间:

$$x_k=a+kh\qquad h=\frac{b-a}{n}\ (k=0,1,\dots,n)$$

$$T_n=\sum_{k=0}^{n-1}\frac{h}{2}[f(x_k)+f(x_{k+1})]=\frac{h}{2}\left[f(a)+2\sum_{k=1}^{n-1}f(x_k)+f(b)\right]$$

【运行方式】

程序已设置好, 直接运行.

【运行结果】

计算结果为 `-0.05555555554929977`（`1000` 等分区间）

已知精确结果

$$-\frac{1}{18} = 0.0\dot{5}$$

我们的精度在 10^{-10}, 远远高于题目精度要求.

【源代码】

```python
##############################################################################
# Computational Physics, 2017-18 Sem1
# HW-1 Ex-9
#
# (c) 2017 Haipeng Lin <linhaipeng@pku.edu.cn>
# All Rights Reserved.
#
# This program is written as Homework for the Computational Physics
# Course in Peking University, School of Physics.
# NO WARRANTIES, EXPRESS OR IMPLIED, ARE OFFERED FOR THIS PRODUCT.
# Copying is strictly prohibited and its usage is solely restricted
# to uses permitted by the author and homework grading.
#
# This program is Python 3 (3.6.2 on MSYS_NT)
# compatible, and does not support Python 2.
#
# Now Playing: 野孩子 - 杨千嬅
##############################################################################

# Euler's Number Constant
e = 2.718281828459045

# This is a simple, Python-dependent implementation of the exp(x)
# If you want to see a Taylor expansion based implementation
# (which is usually more accurate than the Pythonic one)
# Check out Ex-4's Source Code
def exp(x):
    return e**x

# Trapezoid Formula for Integration
# Num integTrapezoid(f lambda x: y, a, b, h)
def integTrapezoid(f, a, b, n):
    result = 0
    h  = (b - a)/n

    # Sum over k = 1, ... n-1 for f(x_k)
    result = h/2 * (f(a) + f(b) + 2*sum([f(a+i*h) for i in range(1,n)]))

    return result

# Ex-9 Specific Code
f = lambda r: (-1)/2187 * r**4 * exp((-1) * 2 * r / 3) * (4/81*r**2 - 16/27*r + 16/9)
print(integTrapezoid(f, 0, 60, 1000))
```

附录 1: 自主编写的 Python 矩阵库 *pylitematrix* 主代码

复现封装需要建立文件夹 pylitematrix, 创建空白的 __init__.py, 和如下的 pylitematrix.py 文件:

```
####################################################################
# The Lightweight Matrix Library, Python 3+
# (c) 2017 Haipeng Lin <linhaipeng@pku.edu.cn>
# Version 1710.02
#
# edu.jimmielin.pylitematrix
#
# This library is fully written by myself to provide for some basic
# matrix operation functions for homework writing.
#
# Loosely based on the Haskell Matrix Functions written by myself,
# used for homework assignments in the "Numerical Methods B" course
# in the 2016/17 Semester 1.
# The paradigms and crude variable naming may look like Haskell,
# e.g. fmap, constructor functions, excessive use of lambdas and
# recursion, etc...
# <3 Haskell (sadly its not allowed in our Homework Assignments)
####################################################################

import math

class Matrix:
    internal = []

    ## Matrix Generators
    # matrix(rows, cols, \i, j -> elem)
    # Returns a matrix with data filled according to the constructor lambda.
    def __init__(self, r, c, fn):
        matrix = []
        for i in range(1, r+1):
            row = []
            for j in range(1, c+1):
                row.append(fn(i, j))
            matrix.append(row)
        self.internal = matrix

    # The below are PROTOTYPE functions and return new objects
    # zeros(i, j)
    # Returns a zero filled matrix
    def zeros_(self, i, j):     return Matrix(i, j, lambda i, j: 0)

    # diag(n, list)
    # Returns a diagonal matrix with first n elems from list taken to construct
    def diag_(self, n, list):   return Matrix(n, n, lambda i, j: list[i-1] if i == j else 0)

    # fromList(rows, cols, list)
    # From a list with at least rows*cols elements generate a matrix left-to-right, top-to-down
    # fromList n m = M n m 0 0 m . V.fromListN(n*m)
    def fromList_(self, r, c, list):
        if(len(list) < r*c):
            raise ValueError("List too short for fromList Matrix Construction", r*c, len(list))
        return Matrix(r, c, lambda i, j: list[(i-1)*c+j-1])

    ## Matrix Manipulation Functions
    # map(fn)
    # equiv. fmap fn matrix, return a copy (for in-place changes, use mapInplace, which is map! ruby)
    def mapInplace(self, fn):
        for i, row in enumerate(self.internal):
            for j, val in enumerate(row):
                self.internal[i][j] = fn(val) # because we are not by reference
        return self

    def map(self, fn):
        mp = Matrix(1, 1, lambda i, j: 0)
        mp.internal = [xs[:] for xs in self.internal]
        # deepcopy because here it is a reference now by default, stupid languages with side effects
        mp.mapInplace(fn)
        return mp

    # toList(matrix)
    # Get elements of matrix stored in a list.
```

```
def toList(self):  return sum(self.internal, [])

# dotproduct_(list1, list2)
# Multiplies two *lists* of the *same* length
def dotProduct__(self, l1, l2):
    if(len(l1) != len(l2)):
        raise ValueError("Attempted to multiply two different-sized lists", l1, l2)
    x = 0
    for i, n in enumerate(l1):
        x += l1[i] * l2[i]
    return x

# Multiplies two *matrices* representing row/col vectors (Prototype)
def dotProduct(self, m1, m2):
    return self.dotProduct__(sum(m1, []), sum(m2, [])) # good taste

# nrows(matrix)
# Gets number of rows (Prototype)
def nrows_(self, m):   return len(m)

# ncols(matrix)
# Gets number of cols (Prototype)
def ncols_(self, m):   return (0 if len(m) == 0 else len(m[0]))

# nrows(matrix)
# Gets number of rows
def nrows(self):   return self.nrows_(self.internal)

# ncols(matrix)
# Gets number of cols
def ncols(self):   return self.ncols_(self.internal)

# getRow(matrix, r) O(1)
def getRow(self, r):
    if(len(self.internal) < r):
        raise ValueError("Out-of-Bounds Matrix Row Access", r, self.internal)
    return self.internal[r-1]

# getCol(matrix, c) O(r)
def getCol(self, c):
    col = []
    if(len(self.internal) == 0 and c > 0):
        raise ValueError("Out-of-Bounds Matrix Column Access", c, self.internal)
    elif(len(self.internal[0]) < c):
        raise ValueError("Out-of-Bounds Matrix Column Access", c, self.internal)

    for i, row in enumerate(self.internal):
        col.append(row[c-1])
    return col

# subMatrix(s_r, e_r, s_c, e_c) (starting/ending row/column)
# extract a submatrix given row and column limits, e.g.
# subMatrix(1, 2, 2, 3) (1 2 3) = (2 3)
#                       (4 5 6) = (5 6)
#                       (7 8 9)
# FIXME: this function is currently unsafe
def subMatrix(self, sr, er, sc, ec):
    # M nrows ncols rowOffset colOffset vcols mvect
    # M (r2-r1+1) (c2-c1+1) (ro+r1-1) (co+c1-1) w v
    return Matrix(er-sr+1, ec-sc+1, lambda i, j: self.internal[sr+i-2][sc+j-2])

# splitBlocks(self, r, c) = (TL, TR, BL, BR) 4-tuple Matrix
# makes a block-partition matrix using given element (r,c) as reference,
# with the reference element staying in the bottom-right corner of the first split.
#                    (              )   (      |      )
#                    (              )   ( ...  | ...  )
#                    (    x         )   (    x |      )
# splitBlocks(r,c) = (              ) = (-------------) , where x = a(r,c)
#                    (              )   (      |      )
#                    (              )   ( ...  | ...  )
#                    (              )   (      |      )
# Note that some blocks can end up empty. We use the following notation for these blocks:
# ( TL | TR )
# (---------)
# ( BL | BR )
```

```
    def splitBlocks(self, r, c):
        return (self.subMatrix(1, r, 1, c), self.subMatrix(1, r, c + 1, self.ncols()), self.subMatrix(r + 
1, self.nrows(), 1, c), self.subMatrix(r + 1, self.nrows(), c + 1, self.nrows()))

    # elem(i, j)
    # Get [i, j] in matrix
    def elem(self, i, j):
        if(len(self.internal) < i or len(self.internal[i-1]) < j):
            raise ValueError("Out-of-Bounds Matrix Access", i, j, self.internal)
        return self.internal[i-1][j-1]

    # setElem_(i,j,val)
    # Set [i,j] in matrix (SIDE EFFECTS, but almost everything is these days)
    def setElem_(self, i, j, val):
        if(len(self.internal) < i or len(self.internal[i-1]) < j):
            raise ValueError("Out-of-Bounds Matrix Access", i, j, self.internal)
        self.internal[i-1][j-1] = val
        return self.internal[i-1][j-1]

    # setElem(i, j, val)
    # Set [i,j] in matrix without side effects (returns a copy), immutable
    def setElem(self, i, j, val):
        mp = Matrix(1, 1, lambda i, j: 0)
        mp.internal = [xs[:] for xs in self.internal]
        mp.setElem_(i, j, val)
        return mp

    # transpose()
    # Get matrix transposition. O(c*r)
    def transpose(self):    return Matrix(self.ncols(), self.nrows(), lambda i, j: self.elem(j, i))

    ## Matrix Computational Functions
    # multStd_ a@(M n m _ _ _ _) b@(M _ m' _ _ _ _) =
    #    matrix n m' $ \(i,j) -> sum [ a !. (i,k) * b !. (k,j) | k <- [1 .. m] ]
    def __mul__(self, m2):
        res = Matrix(self.nrows(), m2.ncols(), lambda i, j: self.dotProduct__(self.getRow(i), 
m2.getCol(j)))
        return res

    # add
    # O(m*n) rather than naive implementation
    def __add__(self, m2):
        if(self.nrows() != m2.nrows() or self.ncols() != m2.ncols()):
            raise TypeError("Cannot add matrices that are not of the same size!")
        return self.fromList_(self.nrows(), self.ncols(), [a+b for a, b in zip(self.toList(), 
m2.toList())])

    # invert for unary arithmetic functions
    def __invert__(self): return self.map(lambda x: -1*x)

    # subtract
    def __sub__(self, m2):
        if(self.nrows() != m2.nrows() or self.ncols() != m2.ncols()):
            raise TypeError("Cannot add matrices that are not of the same size!")
        return self.fromList_(self.nrows(), self.ncols(), [a-b for a, b in zip(self.toList(), 
m2.toList())])

    ## Matrix Pretty-Print
    # adapted from SO#13214809, herein replicated on fair use basis as it
    # is not a crucial component to HW
    def __str__(self):
        s = [[str(n) for n in row] for row in self.internal]
        lens = [max(map(len, col)) for col in zip(*s)]
        form = '\t'.join('{{:{}}}'.format(x) for x in lens)
        table = [form.format(*row) for row in s]
        return ('\n'.join(table))

    ## Matrix Element Functions
    # max() get maximum element from matrix.
    def max(self): return max(self.toList())

    # min() get miminum element from matrix.
    def min(self): return min(self.toList())

# Initialize a Matrix "Prototype"
# Seriously, this is insane - I'm using Javascript paradigms
# in Python.
# But again, the rest is Haskell. You can't get much better than this.
mp = Matrix(1, 1, lambda i, j: 1)

######################################################################
# / end Lightweight Matrix Library
######################################################################
```

附录 2: 所有题目中实现的数值计算功能索引

IEEE Standard Number Conversion

string dec2ieee(*string* N)	ex1-double10.py
string ieee2dec(*string* N)	ex1-2double.py

Numerical Ordinary Differential Equation Solving (ODE Solver)

list (double x, double y) eulerForwardSolve(f: *x, y*, a, b, y0, dt = *None*)	ex2-euler-diffsolve.py
list double (t, x, y) eulerForwardSolveXY(fx: *t, x, y*, fy: *t, y, x*, a, b, x0, y0, dt = *None*)	ex5-throw.py

Numerical Differentiation

double derivF_c(f: *x*, x, h = *None*)	ex4-spherical-harmonics.py
double deriv2F_c(f: *x*, x, h = *None*)	ex4-spherical-harmonics.py
double derivF_c_cycle(f: *x*, x, *int* level = 1, h = *None*)	ex4-spherical-harmonics.py

Numerical Integration

double integTrapezoid(f: *x*, a, b, n (partitions))	ex9-integration.py

Matrix Manipulation

In Matrix.:

(* functions without _ suffix are safe and have no side effects)

Matrix __init__(*int* rows, *int* cols, fn: *i, j*)	pylitematrix.py
Matrix zeros_(*int* rows, *int* cols)	pylitematrix.py
Matrix diag_(*int* n, *list double|int|complex* list)	pylitematrix.py
Matrix fromList_(*int* r, *int* c, *list double|int|complex* list)	pylitematrix.py
Matrix mapInplace(*list double|int|complex* fn: *x)*	pylitematrix.py
Matrix map(*list double|int|complex* fn: x)	pylitematrix.py
list double|int|complex toList()	pylitematrix.py
double|int|complex dotProduct(*Matrix/Vector* m1, m2)	pylitematrix.py
int nrows_(*Matrix* m)	pylitematrix.py
int ncols_(*Matrix* m)	pylitematrix.py
int nrows_()	pylitematrix.py
int ncols_()	pylitematrix.py
list getRow(*int* r)	pylitematrix.py
list getCol(*int* c)	pylitematrix.py
Matrix subMatrix(*int* sr, *int* er, *int* sc, *int* ec)	pylitematrix.py
Matrix splitBlocks(*int r*, *int* c)	pylitematrix.py
double|int|complex elem(*int* i, *int* j)	pylitematrix.py
double|int|complex setElem_(*int* i, *int* j, *double|int|complex* val)	pylitematrix.py
Matrix setElem(*int* i, *int* j, *double|int|complex* val)	pylitematrix.py
Matrix transpose()	pylitematrix.py
Matrix __str__()	pylitematrix.py
Matrix max()/min()	pylitematrix.py

(Other magic functions include __add__, __invert__, __sub__)

Linear Equation Solving

Matrix conjGradient(A, b, x0)	ex6-conj-linsolve.py
Matrix conjGradient2(A, b, x0)	ex6-conj-linsolve.py
Matrix solveTri(M, b)	ex8-spline.py

Interpolation

Function double: *double x* spline(*list double* xs, *list double* ys)	ex8-spline.py
double spline_(*list double* xs, *list double* ys, *double* x)	ex8-spline.py

Special Functions

Integer Factorial/Double Factorials

int factorial(*int* n)	ex4-spherical-harmonics.py
double|int doubleFactorial(*int* n, *double|int* acc = 1)	ex4-spherical-harmonics.py
double|int factorialFromTo(*int* a, *int* b, *double|int* acc = 1)	ex4-spherical-harmonics.py

Legendre Polynomials (**A**ssociated and **N**on-**A**ssociated)

double legendrePolyNA(*int* n, *double* x)	ex4-spherical-harmonics.py
double legendrePolyA(*int* l, *int* m, *double* x)	ex4-spherical-harmonics.py

Trigonometric Functions

double cos(*double* theta)	ex4-spherical-harmonics.py
double sin(*double* theta)	ex4-spherical-harmonics.py
double|complex double exp(*double|complex double* z)	ex4-spherical-harmonics.py

Spherical Harmonics

complex double SphericalHarmonics(*int* l, *int* m, *double* theta, *double* phi)	ex4-spherical-harmonics.py

致谢

在本次大作业的最后我希望感谢:

- 我的家人和室友, 没有他们的支持我无法在漫漫的长夜中点灯完成本次大作业的编程任务;
- 感谢彭良友老师、王鸣老师的《计算物理学》、《计算方法(B)》课程和相关的课程资料, 没有他们的教诲和奉献我无法完成上述这些数值方法的编写;

此外还要感谢为我辛勤计算的电脑和工作站, 感谢 Wolfram Mathematica 提供的精确数值结果,
更感谢计算物理在最后一周居然延伸了 7 天 deadline 让我能够安心周末和车协 22 年会庆.

全文完