实验五 常微分方程初值问题的数值解法

一、实验目的

① 掌握常微分方程数值解的常用算法；
② 培养编程与上机调试能力。

二、实验过程和结果

1. 改进欧拉算法

常微分方程的初值问题：$\begin{cases} y'(x) = f(x, y) \\ y(a) = y_0 \end{cases} \quad (a \le x \le b)$

改进欧拉算法：

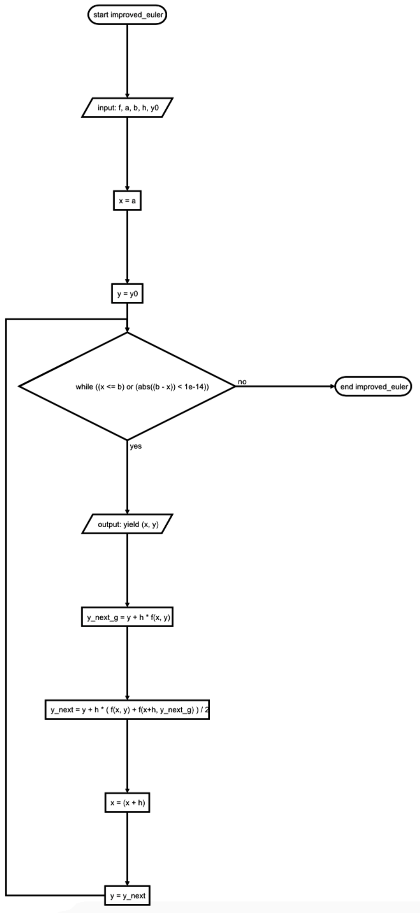
$$\begin{cases} \bar{y}_{n+1} = y_n + hf(x_n, y_n) \\ y_{n+1} = y_n + \frac{h}{2}\left[f(x_n, y_n) + f(x_{n+1}, \bar{y}_{n+1})\right] \end{cases} \quad n = 0,1,2,\dots$$

实现程序[1]：

`improved_euler(f, a, b, h, y0)`

输入参数：

- `f`: 二元函数：$y'(x) = f(x, y)$
- `a, b`: float, x 的区间
- `h`: float , x 迭代步：
  $x_0 = a, x_1 = x_0 + h, x_2 = x_1 + h, \dots, x_n = x_{n-1} + h = b$
- `y0`: float, 初值 $y(a)$



【图1】改进欧拉算法

返回输出：

`(x, y)`： 方程的解 for x from a to b, step h.

---

1 改进欧拉法源码：https://github.com/cdfmlr/NumericalAnalysis/blob/master/ex5/src/improved_euler.py

由于这个东西可以需要返回很多值（当区间$[a,b]$比较大，步长 h 又比较小时），如果用传统的方法返回一个列表（向量）储存全部的值，会消耗大量内存，计算所有结果也需要程序 CPU 密集运行一段时间。而我们真实使用的时候（例如画图时），只需在结果上做迭代，逐次取用结果，一次只使用一个，很少需要同时使用完整的全部结果。

这种情况下，我把该函数实现为一个 Python 的生成器（Generator），实现惰性计算。每次取用 `next(improved_euler(f, a, b, h, y0))` 时，才完成一步的计算，并输出这一步的值。

改进欧拉法的调用实例[2]:

```
rs = improved_euler(lambda x, y: y - x**2 + 1, a=0, b=0.5, h=0.1, y0=0.5)
for r in rs:
    print(r[0],r[1], sep='\t')
```

```
0	0.5
0.1	0.657
0.2	0.828435
0.30000000000000004	1.013720675
0.4	1.2122113458750001
0.5	1.4231935371918751
```

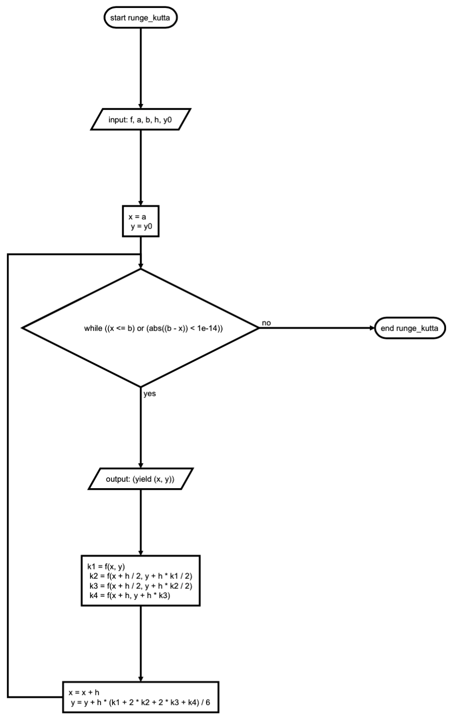
2. 四阶龙格-库塔算法

经典的四阶 Runge-Kutta 算法（RK4）：

$$
\begin{cases}
y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\
k_1 = f\left(x_n, y_n\right) \\
k_2 = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right) \\
k_3 = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right) \\
k_4 = f\left(x_n + h, y_n + hk_3\right)
\end{cases}
$$

看上去比较复杂，但写程序十分容易实现[3]:

`runge_kutta(f, a, b, h, y0)`



【图2】RK4

---

[2] 调用实例节选自： https://github.com/cdfmlr/NumericalAnalysis/blob/master/ex5/src/ex5.ipynb

[3] RK4 实现源码：https://github.com/cdfmlr/NumericalAnalysis/blob/master/ex5/src/rk4.py

该函数输入输出都与前面的改进欧拉法相同，内部实现也很类似，同样实现为生成器，做惰性计算。

3. RKF 算法

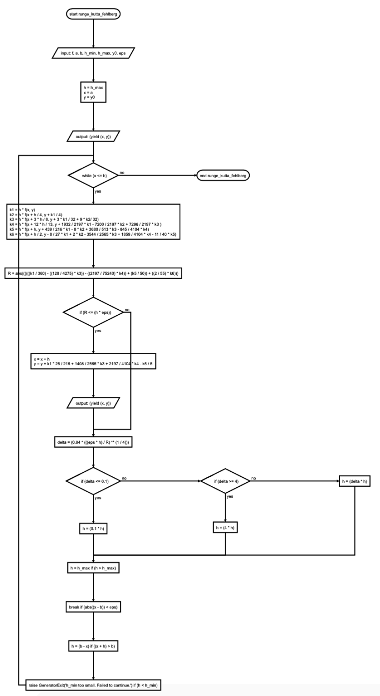
RKF 是自适步长的 Runge-Kutta 方法，在 Runge-Kutta 算法的基础上，增加了动态调整步长大小。

具体来说，RKF 方法使用四阶的 Runge-Kutta 方法来计算结果，并使用五阶 Runge-Kutta 的结果与四阶的结果之差的绝对值估计截断误差，从而预测步长。

实现程序[4]:

`runge_kutta_fehlberg(f, a, b, h_min, h_max, y0, eps)`

这个函数的输入和之前的略有不同，把之前确定的步长 `h` 改成了步长取值区间 `[h_min, h_max]`，并且要求容许误差 `eps`。



【图3】RKF

调用实例，可以看到计算时动态的步长调整：

```
rs = runge_kutta_fehlberg(lambda x, y: y - x**2 + 1, a=0, b=2, h_min=0.01, h_max=0.5, y0=0.5, eps=1e-6)
for r in rs:
    print(r[0],r[1], sep='\t')
```

```
0	0.5
0.13654362227564565	0.7185790374592727
0.26801219081330985	0.954173517401745
0.4012266303587918	1.2166084183124863
0.5366174380061863	1.5060873731856972
0.6747628968898941	1.823047493996752
0.8164946373311293	2.1683758740655024
0.9630852685641695	2.5438210903808947
1.116667783480386	2.952954119903974
1.2813817240537375	3.4038972431729375
1.467877999563017	3.9204151719053937
1.662648091115601	4.453068120471034
1.8154680775168273	4.854886481689374
1.967706858383927	5.230159855113913
2.0	5.30547384382613
```

[4] RKF 源码：https://github.com/cdfmlr/NumericalAnalysis/blob/master/ex5/src/rkf.py

4. 实验内容题目

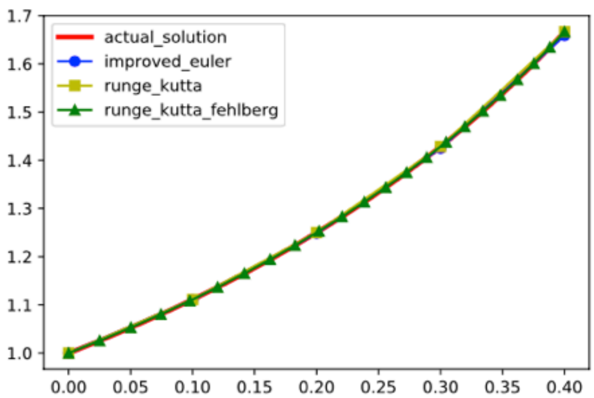
a) $\begin{cases} \dfrac{dy}{dx} = y^2 \\ y(0) = 1 \end{cases} \quad 0.1 \le x \le 0.4 \quad h = 0.1$

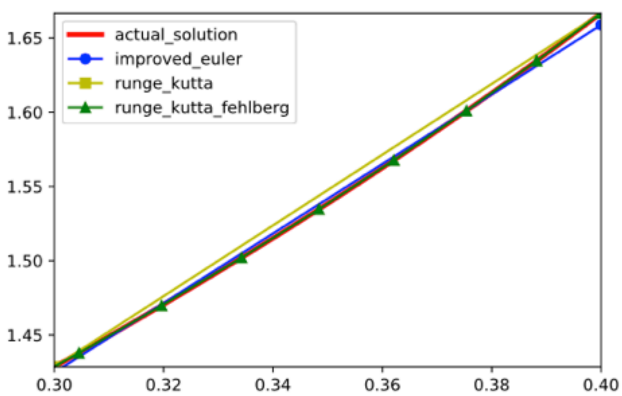
分别使用改进欧拉算法、RK4 和 RKF 算法进行求解[5]，这里封装了一个类来调用各种算法求解问题，并作图比较。调用即可完成求解：

```
def f(x, y):
    return y ** 2

a, b = 0, 0.4
h = 0.1
y0 = 1
h_min=0.001
h_max=0.5
eps=1e-8

def actual_solution(x):
    return 1 / (1 - x)

#  def __init__(self, f, a, b, h, y0, h_min, h_max, eps, actual_solution):
q1 = Question(f, a, b, h, y0, h_min, h_max, eps, actual_solution)
q1.solve()
```

最终等到的结果：

```
Improved Euler:
0        1
0.1      1.1105
0.2      1.2482762285866027
0.30000000000000004      1.4247601260213614
0.4      1.6587363946557603

RK4:
0        1
0.1      1.1111104900521944
0.2      1.2499979920470152
0.30000000000000004      1.4285661863014445
0.4      1.6666532572503225

RKF:
0        1
0.025039979795246234     1.0256830838146338
0.05023788001716733      1.052895224207251
0.07449021044767089      1.080485600141923
0.09781031904296066      1.1084143630136087
0.12024490661926406      1.1366799785401573
0.1418387901059162       1.1652822211726135
0.16263399033424697      1.194220913397017
0.1826699094328344       1.2234958832046705
0.20198352501286038      1.253106962683577
0.22060956834055725      1.2830539876206315
0.23858068763984974      1.3133367976484365
0.2559275973202766       1.3439552358985725
0.27267921507337145      1.3749091491644632
0.2888627873258841       1.4061983873896555
0.3045040049273295       1.4378228039555951
0.31962710915217313      1.4697822551801247
0.33425498973396817      1.5020766007259563
0.3484092747167351       1.5347057029264364
0.362110413792854 6      1.5676694270969493
0.37537775507854887      1.6009676414571847
0.38822961593386085      1.634600216668164
0.4      1.666666670706118 4
```





[5] 具体的求解过程见： https://github.com/cdfmlr/NumericalAnalysis/blob/master/ex5/src/ex5.ipynb

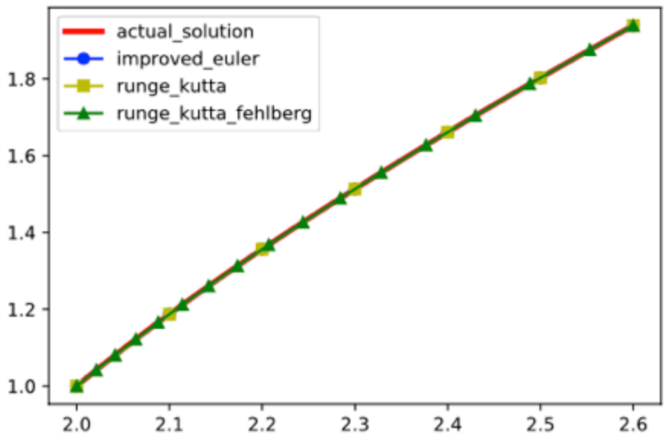
b) $\begin{cases} \frac{dy}{dx} = x/y \\ y(2.0) = 1 \end{cases} \quad 2.0 \le x \le 2.6 \quad h = 0.1$

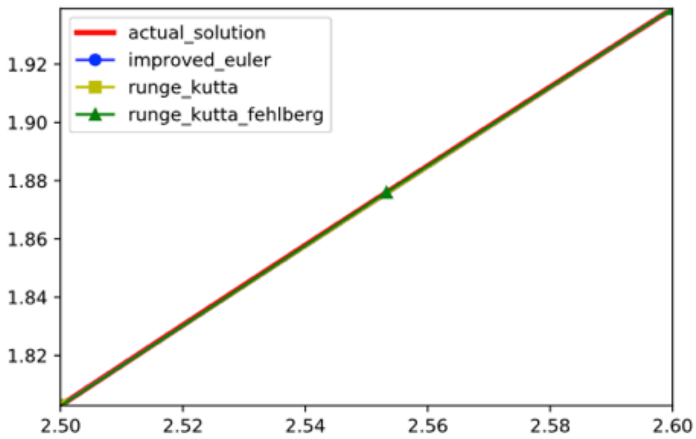
同样使用改进欧拉算法、RK4 和 RKF 算法进行求解：

```python
def f(x, y):
    return x / y

a, b = 2.0, 2.6
h = 0.1
y0 = 1
h_min=0.001
h_max=0.5
eps=1e-8

def actual_solution(x):
    return (x**2 - 3) ** (1/2)

#  def __init__(self, f, a, b, h, y0, h_min, h_max, eps, actual_solution):
q2 = Question(f, a, b, h, y0, h_min, h_max, eps, actual_solution)
q2.solve()
```

最终等到的结果（由于结果差距很小，在图像上，先画的改进欧拉法被其他算法盖住了）：

```
Improved Euler:
2.0	1
2.1	1.1875
2.2	1.3565459923557568
2.3000000000000003	1.5133558178802287
2.4000000000000004	1.6614034116615422
2.5000000000000004	1.802850617838112
2.6000000000000005	1.9391431095639506

RK4:
2.0	1
2.1	1.1874362471788942
2.2	1.3564683099780293
2.3000000000000003	1.513276851413264
2.4000000000000004	1.6613269035807452
2.5000000000000004	1.8027776370912958
2.6000000000000005	1.9390738201189597

RKF:
2.0	1
2.0211724308032437	1.0416995703232401
2.0415542156639184	1.080714400351875
2.063772665098076	1.1221219241586564
2.0878610104513116	1.16583171952646
2.1140161190991194	1.212049566189454
2.142450034889673	1.2609885609556355
2.173401878648306	1.3128883138785854
2.2071414713667057	1.3680180820976793
2.243974231145123	1.4266815860724658
2.2842469848065763	1.4892227117613641
2.3283549439053477	1.5560323717778588
2.3767501221468326	1.627556800810627
2.429951552025518	1.7043076428256803
2.488557762018454	1.786874290623765
2.553262116784645	1.875939080006759
2.6	1.9390719416220605
```





三、思考题分析解答

1. 用常微分方程初值问题的数值方法计算 $\int_0^1 \frac{sint}{t}dt.$

问题转化为求 $\begin{cases} y'(x) = \frac{\sin x}{x} \\ y(0) = 0 \end{cases}$ $0 \le x \le 1$ 在 1 处的值。用四阶龙格库塔求解：

```
import math
import sympy as sp

def f(x, y):
    return math.sin(x) / x

a, b = 1e-64, 1
h = 0.1
y0 = 1e-64

# 用 RK4 求解
rs = runge_kutta(f, a, b, h, y0)
I = list(rs)[-1][-1]
print("By RK4:", I)

# 用 sympy 求解析解
x = sp.Symbol('x')
actual = sp.integrate(sp.sin(x) / x, (x, 0, 1))
print(f"Actual: {actual}={float(actual)}")

abs(actual - I) < 1e-6
```

```
By RK4: 0.946083076517732
Actual: Si(1)=0.946083070367183
True
```

2. RKF 算法的实现

详见 实验内容 3. RKF 算法。

四、重点难点分析

重点：

① 掌握常微分方程数值解的常用算法；
② 培养编程与上机调试能力。

难点：

RKF 方法的编程实现，其中有大量常数的输入、计算，容易错，可以考虑用一个矩阵储存 k1 ~ k6 计算过程中的各种常数，这样可以大幅降低打字错误的风险。这种方法的实现在这一次 Git 提交：10a08ee。