

计算方法实验报告

姓名: 杨杰睿

学号: 200110428

院系: 计算机科学与技术学院

专业: 计算机类

班级: 计算机类 4 班

实验题目1 拉格朗日(Lagrange)插值

实验简介

本实验为拉格朗日插值实验，需要完成拉格朗日插值的代码编写并通过求解实验题目的答案。

本次实验过程中，主要为学习拉格朗日插值的代码编写，同时深入的理解、体会教材所言拉格朗日插值代码编写简洁的优点，同时从程序执行的流程理解其算法本身的缺陷。

实验的目的即为使用拉格朗日插值法求解函数的近似值。

该实验报告主要分为7个部分，大纲罗列如下：

- **实验简介**: 即本部分的所有内容
- **数学原理**: 对拉格朗日插值算法的数学原理进行阐述
- **代码实现**: 使用 `Julia` 语言，根据数学原理，编写实验代码
- **实验题目**: 实验指导书中所要求完成的实验题目，作有便于直观观察的**示意图**，同时题目答案已输出为**表格**展示。

注：详细执行过程请参考**附录：执行代码**部分

- **问题1**: 探究插值的阶数和准确性的关系，观察Runge现象，切身体会为什么不建议使用高阶的多项式插值函数
- **问题2**: 探究插值的区间长度选取和待插值函数的关系，从两个不同的函数对于不同阶数插值、在不同的区间长度下插值对于拟合结果的影响，理解进行插值时选取合适的插值区间来达到期望的拟合精度和对计算资源的节省，是基于对函数本身特点的认识的
- **问题4**: 探究插值外推和内插的相对可靠性，从直觉上很容易意识到内插是比外推可靠的，但这一未经数学证明的结论需要更深入的数学知识，我们通过对部分实例进行探索，从实例呈现的插值结果来看，这一结论成立的条件是基于函数的性质的。
- **思考题**: 本部分为实验指导书中所要求的完成的思考题解答
- **参考资料**: 本次实验过程中查阅的参考资料
- **附录**: 本实验的非主体部分
 - **执行代码**: 本部分是对于各个问题求解的过程进行封装，对于外界只需要传入问题所需参数，在封装好的代码内部会调用函数，完成插值求解多项式、绘制示意图以及打印实验结果的部分
 - **测试代码**: 对于程序的运行、输出进行测试的部分

Test 1 - Simple: 使用教材例题对程序运行进行简单的测试，确保基本的程序流程的正确性

Test 2 - Performance: 本实验中主要是对于不同代码实现性能的测试，最终选择了耗时更短的实现，同时由于运行较为耗时，最终会处于被注释的状态

数学原理

插值基函数

令 $l_j(x) (j = 0, 1, 2, \dots, n)$ 表示 n 次多项式，满足条件

$$l_j(x_i) = \begin{cases} 0, & i \neq j, \\ 1, & i = j, \end{cases} j, i = 0, 1, \dots, n.$$

我们称 $l_j(x) (j = 0, 1, 2, \dots, n)$ 为多项式的插值基函数。

Lagrange插值公式

显然,存在n次多项式

$$y(x) = \sum_{j=0}^n f(x_j)l_j(x). \quad (*)$$

满足插值条件式,故问题可以归结为构造满足插值基函数的n次多项式 $l_j(x)$ ($j = 0, 1, 2, \dots, n$)

很容易得知, $l_j(x)$ 应该有n个零点 $x_0, \dots, x_{j-1}, x_{j+1}, \dots, x_n$,又因为 $l_j(x)$ 是n次多项式, 所以一定具有形式

$$l_j(x) = A_j(x - x_0) \cdots (x - x_{j-1})(x - x_{j+1}) \cdots (x - x_n),$$

其中, A_j 是与 x 无关的数,由 $l_j(x_j) = 1$ 可以确定,即

$$l_j(x_j) = 1 = A_j(x_j - x_0) \cdots (x_j - x_{j-1})(x_j - x_{j+1}) \cdots (x_j - x_n),$$

故有

$$l_j(x) = \frac{(x - x_0)(x - x_1) \cdots (x - x_{j-1})(x - x_{j+1}) \cdots (x - x_n)}{(x_j - x_0)(x_j - x_1) \cdots (x_j - x_{j-1})(x_j - x_{j+1}) \cdots (x_j - x_n)}, j = 0, 1, 2, \dots, n. \quad (**)$$

综上所述,当n次多项式 $l_j(x)$ ($j = 0, 1, 2, \dots, n$)由(**)方程确定时,n次多项式满足插值条件式.可以证明,这样的多项式是唯一的.

我们称式(*)为Lagrange插值公式, (**)为Lagrange插值多项式,记为 $L_n(x)$.

代码实现

此处的代码, 实际上就是将上述的(*)和(**)式用编程语言重新表示, 较为直接, 用于编程对照的公式如下:

$$y(x) = \sum_{j=0}^n f(x_j)l_j(x). \quad (*)$$

$$l_j(x) = \frac{(x - x_0)(x - x_1) \cdots (x - x_{j-1})(x - x_{j+1}) \cdots (x - x_n)}{(x_j - x_0)(x_j - x_1) \cdots (x_j - x_{j-1})(x_j - x_{j+1}) \cdots (x_j - x_n)}, j = 0, 1, 2, \dots, n. \quad (**)$$

首先, 导入需要使用的包

```
using Printf
using Plots
using Statistics
using LinearAlgebra
using LaTeXStrings
using PrettyTables
```

然后, 在此处定义lagrange插值函数, 这里对其进行函数重载, 用于适应传入的测试参数为单个数值和一组数值(向量)的情形

```
# Lagrange Interpolation method
# 对函数进行重载, 既可以对单个的点进行测试, 也可以测试一系列的点
function lagrange(xs, fxs, x::Number)
    num = size(xs, 1)
    y, i = 0.0, 1
    while i <= num
        li = 1.0
        for j in 1:num
            if j == i
                continue
            end
            li *= (x - xs[j]) / (xs[i] - xs[j])
        end
        y += fxs[i] * li
    end
    return y
end
```

```

    end
    y += li * fxs[i]
    i += 1
end
x, y
end

function lagrange(xs, fxs, x::Vector)
    num = size(xs, 1)
    y, i = zeros(size(x, 1)), 1
    while i <= num
        li = fill(1.0, size(x, 1))
        for j in 1:num
            if j == i
                continue
            end
            li = li .* (x .- xs[j]) / (xs[i] - xs[j])
        end
        y = y + li .* fxs[i]
        i += 1
    end
    x, y
end

```

实验题目

问题 1

拉格朗日插值多项式的次数n越大越好吗？

不是，若是次数过高，会出现Runge现象，插值多项式在距离已知点位置较远处会剧烈震荡，直观呈现可见下列问题所作的示意图，20阶的方法最明显。

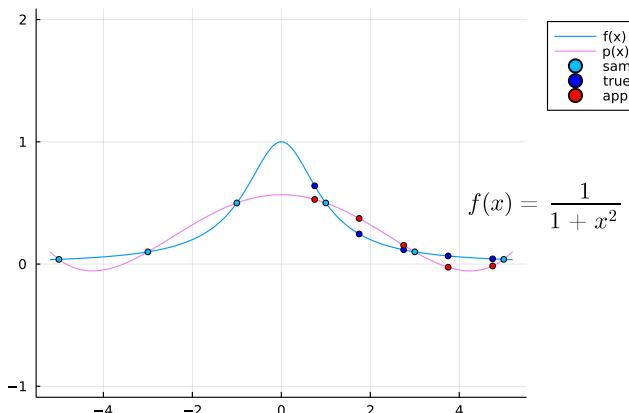
1.1

```

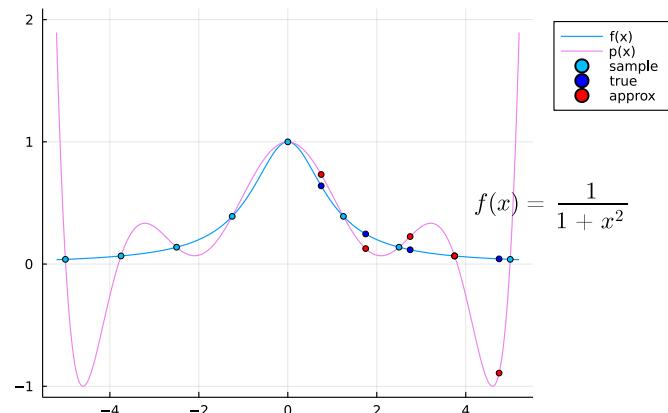
f(x) = 1 / (1 + x^2)
split_nums = [5, 8, 10, 20] # 多加了一个8阶的结果
test_x = [0.75, 1.75, 2.75, 3.75, 4.75]
xlim = [-5, 5]
ylim = [-1, 2]
println("f(x) = 1 / (1 + x^2)")
prefix = "Problem 1.1"
text = L"f(x)=\frac{1}{1+x^2}"
show_result(f, split_nums, test_x, xlim, ylim, prefix, text)

```

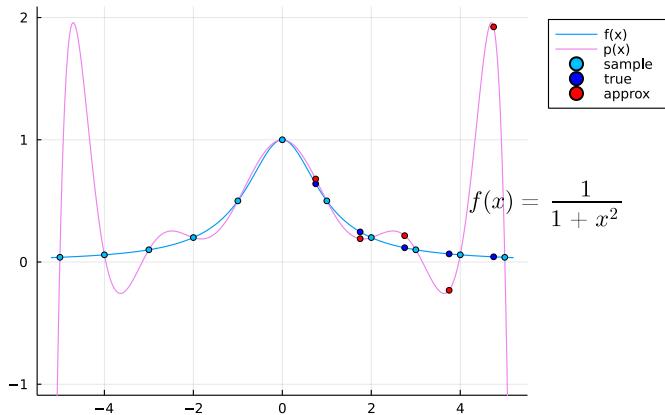
Problem 1.1 5-Order Interpolation



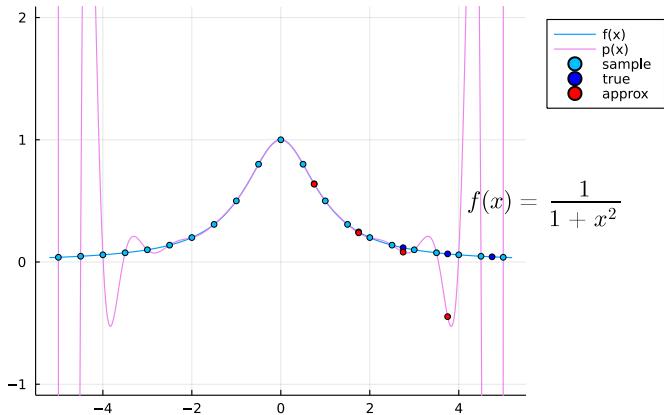
Problem 1.1 8-Order Interpolation



Problem 1.1 10-Order Interpolation



Problem 1.1 20-Order Interpolation



$$f(x) = 1 / (1 + x^2)$$

Problem 1.1 5-Order Interpolation:

Test x	Test y	Pred y
0.750000	0.640000	0.528974
1.750000	0.246154	0.373325
2.750000	0.116788	0.153733
3.750000	0.066390	-0.025954
4.750000	0.042440	-0.015738

Problem 1.1 8-Order Interpolation:

Test x	Test y	Pred y
0.750000	0.640000	0.732842
1.750000	0.246154	0.126609
2.750000	0.116788	0.225141
3.750000	0.066390	0.066390
4.750000	0.042440	-0.891667

Problem 1.1 10-Order Interpolation:

Test x	Test y	Pred y
0.750000	0.640000	0.678990
1.750000	0.246154	0.190580
2.750000	0.116788	0.215592
3.750000	0.066390	-0.231462
4.750000	0.042440	1.923631

Problem 1.1 20-Order Interpolation:

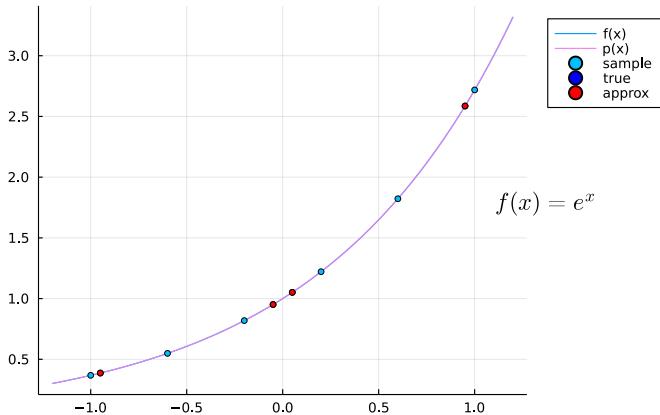
Test x	Test y	Pred y
0.750000	0.640000	0.636755
1.750000	0.246154	0.238446
2.750000	0.116788	0.080660
3.750000	0.066390	-0.447052
4.750000	0.042440	-39.952449

```

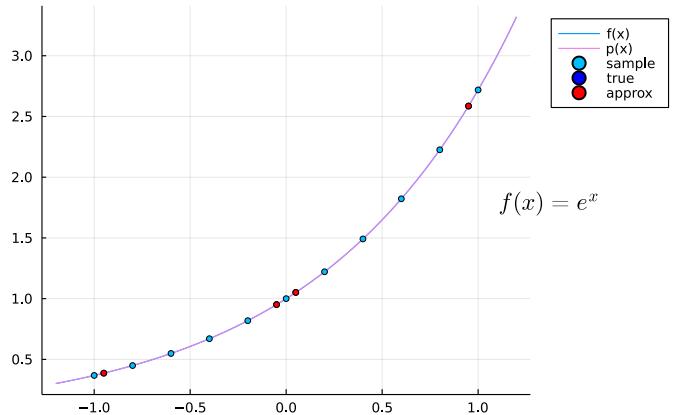
f(x) = exp(x)
split_nums = [5, 10, 20]
test_x = [-0.95, -0.05, 0.05, 0.95]
xlim = [-1, 1]
# ylim = [-1, 10] # the good-looking ylim is defined manually
ylim = []
println("f(x) = exp(x)")
prefix = "Problem 1.2 "
text = L"f(x)=e^x"
show_result(f, split_nums, test_x, xlim, ylim, prefix, text)

```

Problem 1.2 5-Order Interpolation



Problem 1.2 10-Order Interpolation

 $f(x) = \exp(x)$

Problem 1.2 5-Order Interpolation:

Test x	Test y	Pred y
-0.950000	0.386741	0.386798
-0.050000	0.951229	0.951248
0.050000	1.051271	1.051290
0.950000	2.585710	2.585785

Problem 1.2 10-Order Interpolation:

Test x	Test y	Pred y
-0.950000	0.386741	0.386741
-0.050000	0.951229	0.951229
0.050000	1.051271	1.051271
0.950000	2.585710	2.585710

Problem 1.2 20-Order Interpolation:

Test x	Test y	Pred y
-0.950000	0.386741	0.386741
-0.050000	0.951229	0.951229
0.050000	1.051271	1.051271
0.950000	2.585710	2.585710

问题 2

插值区间越小越好吗?

不一定，从精度上考虑虽然有一定的合理性，但插值节点过于密集时，一方面计算量增大却没提高对于精度计算的收益，另一方面区间缩短、节点增加并不能保证两节点间能很好的逼近函数，反而有可能出现Runge现象。但合理的对区间长度进行选择，同时采用低次插值来避免Runge现象，能够得到较好的拟合效果。

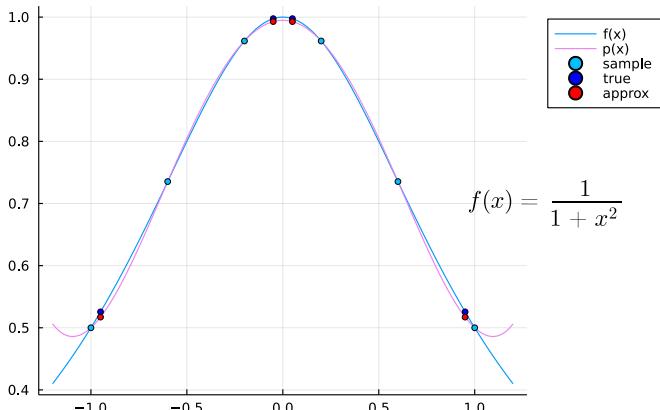
不过，实例中对于函数 $f(x) = \frac{1}{1+x^2}$ ，较短区间的插值效果比长区间插值更好

而函数 $f(x) = e^x$ 无论是长区间还是短区间插值，都能得到相对较好的拟合效果，但短区间插值相对误差更低

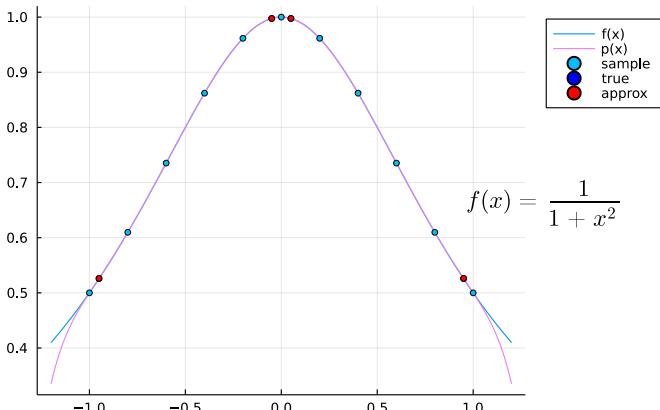
2.1

```
f(x) = 1 / (1 + x^2)
split_nums = [5, 10, 20]
test_x = [-0.95, -0.05, 0.05, 0.95]
xlim = [-1, 1]
# ylim = [-1, 2]
ylim = []
println("f(x) = 1 / (1 + x^2)")
prefix = "Problem 2.1"
text = L"f(x) = \frac{1}{1+x^2}"
show_result(f, split_nums, test_x, xlim, ylim, prefix, text)
```

Problem 2.1 5-Order Interpolation



Problem 2.1 10-Order Interpolation



$$f(x) = 1 / (1 + x^2)$$

Problem 2.1 5-Order Interpolation:

Test x	Test y	Pred y
-0.950000	0.525624	0.517147
-0.050000	0.997506	0.992791
0.050000	0.997506	0.992791
0.950000	0.525624	0.517147

Problem 2.1 10-Order Interpolation:

Test x	Test y	Pred y
-0.950000	0.525624	0.526408
-0.050000	0.997506	0.997507
0.050000	0.997506	0.997507
0.950000	0.525624	0.526408

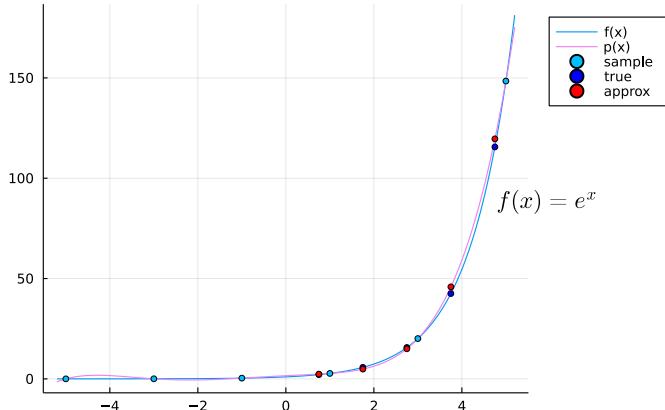
Problem 2.1 20-Order Interpolation:

Test x	Test y	Pred y
-0.950000	0.525624	0.525620
-0.050000	0.997506	0.997506
0.050000	0.997506	0.997506
0.950000	0.525624	0.525620

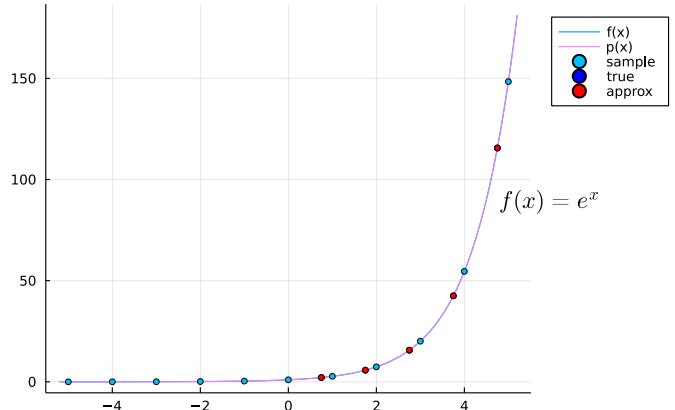
2.2

```
f(x) = exp(x)
split_nums = [5, 10, 20]
test_x = [0.75, 1.75, 2.75, 3.75, 4.75]
xlim = [-5, 5]
# ylim = [-1, 10] # the good-looking ylim is defined manually
ylim = []
println("f(x) = exp(x)")
prefix = "Problem 2.2 "
text = L"f(x) = e^x"
show_result(f, split_nums, test_x, xlim, ylim, prefix, text)
```

Problem 2.2 5-Order Interpolation



Problem 2.2 10-Order Interpolation



$$f(x) = \exp(x)$$

Problem 2.2 5-Order Interpolation:

Test x	Test y	Pred y
0.750000	2.117000	2.373957
1.750000	5.754603	4.871635
2.750000	15.642632	15.008061
3.750000	42.521082	45.862257
4.750000	115.584285	119.621007

Problem 2.2 10-Order Interpolation:

Test x	Test y	Pred y

0.750000	2.117000	2.117136
1.750000	5.754603	5.754367
2.750000	15.642632	15.643248
3.750000	42.521082	42.518431
4.750000	115.584285	115.607360

Problem 2.2 20-order Interpolation:

Test x	Test y	Pred y
0.750000	2.117000	2.117000
1.750000	5.754603	5.754603
2.750000	15.642632	15.642632
3.750000	42.521082	42.521082
4.750000	115.584285	115.584285

问题 4

考虑拉格朗日插值问题，内插比外推更可靠吗？

不一定，这取决于函数的性质，但通常我们认为对于连续函数内插的可靠程度更高。

外推等价于根据已知点预测完全未知点的函数值，但我们所得的插值多项式不含有任何有关待拟合函数的已知点外的信息，根据多项式函数的特性进行外推是不合理的。

而考虑到连续函数，内插则不会对于函数的拟合存在无根据的外推过程，有更高的可靠程度。

从实验结果来看，第一个实例体现的是外推的严重错误，尽管第二个实例中外推所得误差稍小于内插结果，但在事实上这只是所选区间拟合的巧合，而内插误差虽然略高，却也具有相当低的误差和相当高的可靠程度。

```
f(x) = sqrt(x)
split_xs = [1, 4, 9]
test_x = [5, 50, 115, 185]
xlim = [0, 200]
# ylim = [-1, 2]
ylim = []
println("Problem 4.1 f(x) = sqrt(x)")
prefix = "Problem 4.1 "
text = L"f(x) = \sqrt{x}"
comment = "This is a test comment"
comment = L"last~3~points~are~extrapolation~results"
show_result(f, nothing, split_xs, test_x, xlim, ylim, prefix, text, comment)

f(x) = sqrt(x)
split_xs = [36, 49, 64]
test_x = [5, 50, 115, 185]
xlim = [0, 200]
# ylim = [-1, 2]
ylim = []
println("Problem 4.2 f(x) = sqrt(x)")
prefix = "Problem 4.2 "
text = L"f(x) = \sqrt{x}"
comment = L"only~the~second~one~is~interpolation~result"
show_result(f, nothing, split_xs, test_x, xlim, ylim, prefix, text, comment)

f(x) = sqrt(x)
split_xs = [100, 121, 144]
```

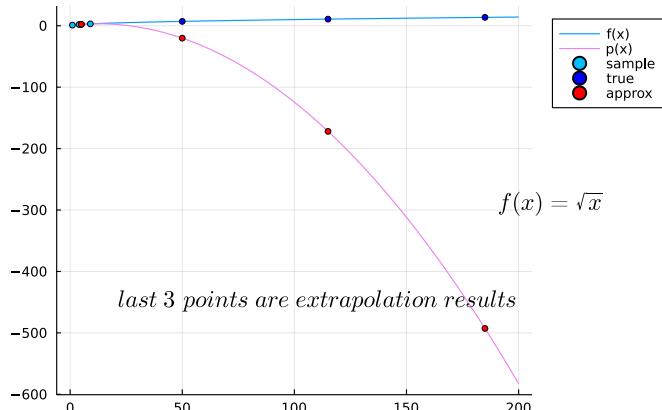
```

test_x = [5, 50, 115, 185]
xlim = [0, 250]
# ylim = [-1, 2]
ylim = []
println("Problem 4.3 f(x) = sqrt(x)")
prefix = "Problem 4.3 "
text = L"f(x) = \sqrt{x}"
comment = L"only~the~third~one~is~interpolation~result"
show_result(f, nothing, split_xs, test_x, xlim, ylim, prefix, text, comment)

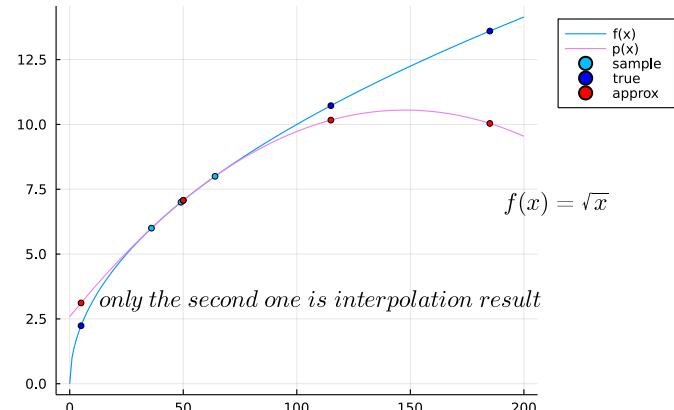
f(x) = sqrt(x)
split_xs = [169, 196, 225]
test_x = [5, 50, 115, 185]
xlim = [0, 250]
# ylim = [-1, 2]
ylim = []
println("Problem 4.4 f(x) = sqrt(x)")
prefix = "Problem 4.4 "
text = L"f(x) = \sqrt{x}"
comment = L"first~3~points~are~extrapolation~results"
show_result(f, nothing, split_xs, test_x, xlim, ylim, prefix, text, comment)

```

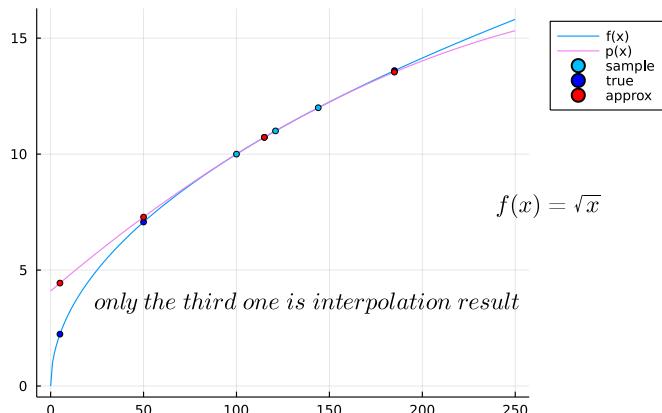
Problem 4.1 3-Order Interpolation



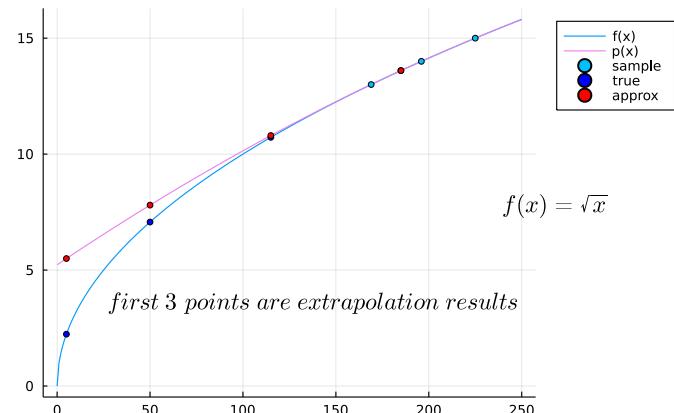
Problem 4.2 3-Order Interpolation



Problem 4.3 3-Order Interpolation



Problem 4.4 3-Order Interpolation



Problem 4.1 f(x) = sqrt(x)

Test x	Test y	Pred y
5.000000	2.236068	2.266667
50.000000	7.071068	-20.233333
115.000000	10.723805	-171.900000
185.000000	13.601471	-492.733333

Problem 4.2 $f(x) = \sqrt{x}$

Test x	Test y	Pred y
5.000000	2.236068	3.115751
50.000000	7.071068	7.071795
115.000000	10.723805	10.167033
185.000000	13.601471	10.038828

Problem 4.3 $f(x) = \sqrt{x}$

Test x	Test y	Pred y
5.000000	2.236068	4.439112
50.000000	7.071068	7.284961
115.000000	10.723805	10.722756
185.000000	13.601471	13.535667

Problem 4.4 $f(x) = \sqrt{x}$

Test x	Test y	Pred y
5.000000	2.236068	5.497172
50.000000	7.071068	7.800128
115.000000	10.723805	10.800493
185.000000	13.601471	13.600620

思考题

1. 对实验 1 存在的问题，应如何解决？

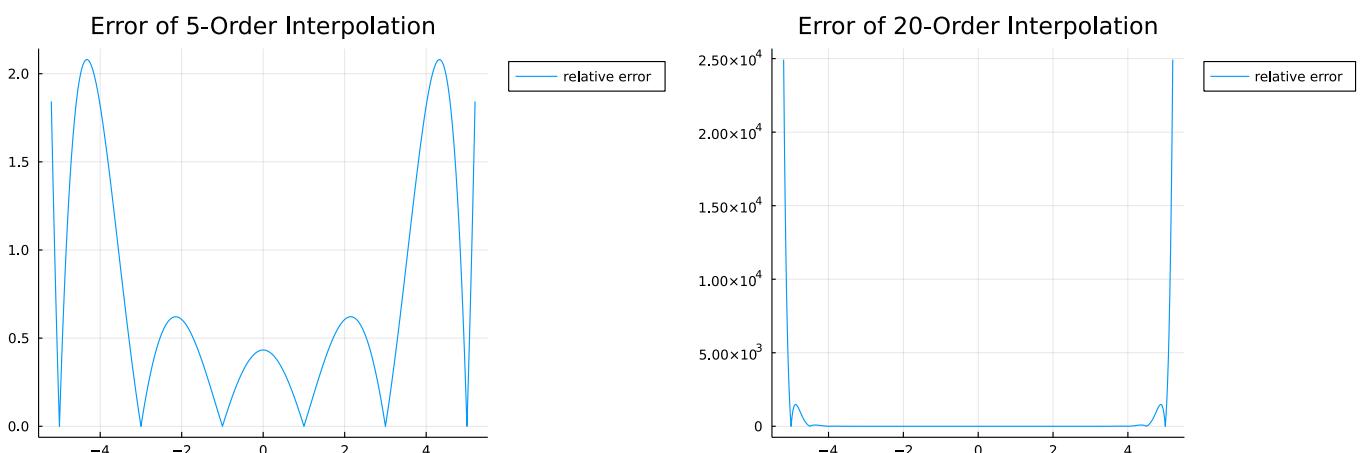
当插值多项式次数过高的时候会出现Runge现象，插值多项式在距离已知点位置较远处会剧烈震荡，越靠近端点，逼近的效果越差，这表明了节点的密集不一定能保证在两节点间插值函数逼近程度的上升。

对函数 $f(x) = \frac{1}{1+x^2}$ 进行5阶、20阶插值的相对误差的变化，如下图所示：

很明显看到，误差在距离已知点之外的急剧增长，相对误差高达 10^4 量级，结果已经严重失真。目前而言，这一问题的解决方案主要有两类：

一个是从插值函数的二阶导数剧烈变化出发，修改插值条件对插值函数的二阶导数进行限制，如使用Hermite型插值；

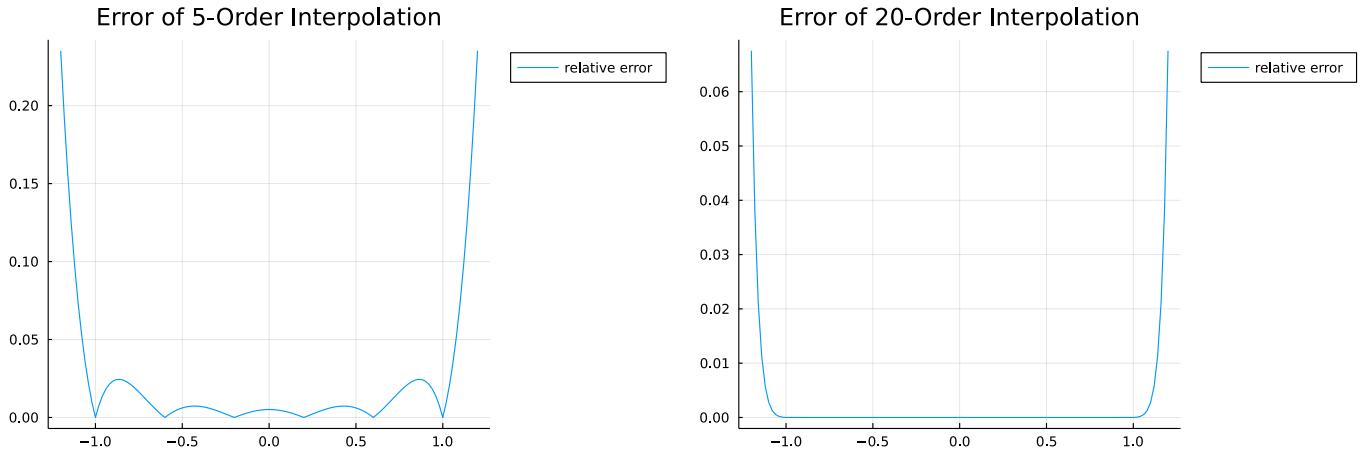
另一种是将长区间划分为若干个小区间，在每一个小区间上分别做低次插值来避免Runge现象，逼近效果要比在整个区间上用高阶光滑差值效果更好，即使用分段插值和样条插值。



2. 对实验 2 存在的问题的回答，试加以说明

首先不一定，从精度上考虑虽然有一定的合理性，但插值节点过于密集时，一方面计算量增大却没提高对于精度计算的收益，另一方面区间缩短、节点增加并不能保证两节点间能很好的逼近函数，反而有可能出现Runge现象。但合理的对区间长度进行选择，同时采用低次插值来避免Runge现象，能够得到较好的拟合效果。

不过，实例中对于函数 $f(x) = \frac{1}{1+x^2}$ ，较短区间的插值效果比长区间插值更好，而且短区间插值甚至缓解了Runge现象对于精度的影响，虽然相对而言边界点仍有较大的误差，但其相对误差的绝对大小也明显低于低阶插值的结果，如下图所示；而函数 $f(x) = e^x$ 无论是长区间还是短区间插值，都能得到相对较好的拟合效果，但短区间插值相对误差更低，此处不再赘述。



3. 略

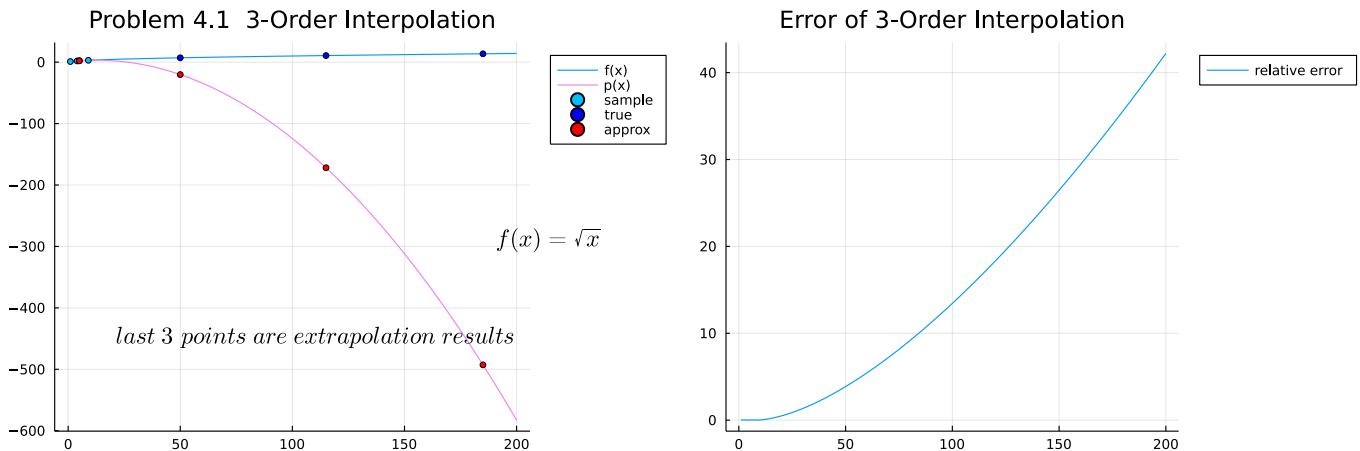
4. 如何理解插值问题中的内插和外推？

通常我们认为对于连续函数内插的可靠程度高于外推，因为对于未知的连续函数而言，我们无法预知任何在已知点信息之外的有关函数的信息，无法简单通过多项式插值来对函数趋势进行判断。

外推等价于根据已知点预测完全未知点的函数值，但我们所得的插值多项式不含有任何有关待拟合函数的已知点外的信息，我们没有理由认为实际函数的变化必须符合多项式函数的变化，根据多项式函数的特性进行外推是不合理的，故这里我们可以简单的认为即时出现外推的结果相比于内插更好的情况，也很大程度上是函数本身特性导致的巧合，当然，或许也可以根据函数的性质来决定函数是否适合进行多项式差值的外推。

认为内插比外推可靠的原因是，在内插的区间我们认为连续函数在相邻点间的变化不会过于剧烈，从而可以简单的认为内插更加可靠，而外推时我们没有任何而先验知识可以用于对完全未知的区间函数值进行推断。

从实验结果来看，第一个实例体现的是外推的严重错误（如下图所示），接下来的几个实例中外推所得误差和内插结果相差不明显，但在即便如此也只是所选区间拟合的巧合，因为第一个实例中的极端例子就说明了，不加考量的直接外推会造成的灾难性后果。



参考资料

1. julia plot xlim and ylim <https://stackoverflow.com/questions/53230969/how-to-scale-a-plot-in-julia-using-plots-jl>
2. julia fill array with specific value <https://www.geeksforgeeks.org/fill-an-array-with-specific-values-in-julia-array-filling-method/>
3. julia prettystables <https://ronisbr.github.io/PrettyTables.jl/stable/>
4. 《数值分析原理》吴勃英 105-106,123
5. markdown多张图片并排显示 <https://www.cnblogs.com/jaycethanks/p/12201959.html>
6. markdown图片大小设定 <https://www.cnblogs.com/jaycethanks/p/12202169.html>
7. 《计算方法实验指导》实验题目 1 拉格朗日(Lagrange)插值

附录

执行代码

这一部分的代码是将展示结果的部分进行封装，运行时只需要调用一个封装后的函数，传入不同例题所给定的不同参数即可运行得到展示的结果。

首先定义的是展示误差图像的函数，该部分在最终运行时被注释处理，以简化结果呈现。

然后是两个展示结果的函数，由于问题1,2和问题4略有不同，重载了不同参数列表的同名函数。

```
function show_error(f::Function, title::String, series_x, series_y)
    errors = abs.(f.(series_x) - series_y) ./ f.(series_x)
    #plot(series_x, errors, label="relative error", title=title, legend=:outertopright)
end

function show_result(f::Function, split_nums::Vector, test_x::Vector, xlim::Vector,
ylim::Vector, prefix, text)
    for n in split_nums
        # initialization
        x_min, x_max = xlim
        x_range = x_min-0.2:0.02:x_max+0.2
        xs = x_min:(x_max-x_min)/n:x_max
        ys = f.(xs)

        plot(x_range, f.(x_range), label="f(x)") # plot f(x)
        plot!(legend=:outertopright, title=prefix * " $n-Order Interpolation")
        series_x = Vector(x_range)
        _, series_y = lagrange(xs, ys, series_x) # compute the interpolation function points
        plot!(series_x, series_y, color=:violet, label="p(x)") # add p(x) function curve
        plot!(ylim=ylim, yflip=false) # add ylim
        # add sample for lagrange interpolation
        plot!(xs, ys, seriestype=:scatter, markersize=3, msw=1, color=:deepskyblue,
label="sample")
        test_y = f.(test_x)
        # add test x & y, plot true points
        p = plot!(test_x, test_y, seriestype=:scatter, markersize=3, msw=1, color=:blue,
label="true")
        _, pred_y = lagrange(xs, ys, test_x)
        println()
        println(prefix * " $n-Order Interpolation:")

        data = [test_x test_y pred_y]
        header = (["Test x", "Test y", "Pred y"])
        pretty_table(
            data;
```

```

        alignment=[:c, :c, :c],
        header=header,
        header_crayon=crayon"bold",
        formatters=ft_printf("%11.6f"))

    xmin, xmax = xlims(p)
    x = xmax + (xmax - xmin) * 0.04
    y = mean(ylims(p))
    ymax = ylims(p)[2]
    annotate!(x, y, text, :black)
    # add pred_y by lagrange interpolation
    display(plot!(test_x, pred_y, seriestype=:scatter, markersize=3, msw=1, color=:red,
label="approx"))
end
end

```

```

function show_result(f::Function, split_nums::Nothing, split_xs::Vector, test_x, xlim, ylim,
prefix, text, comment)
    x_min, x_max = xlim
    x_range = x_min:1:x_max+0.2 # x_min cannot be negative
    xs = split_xs
    ys = f.(xs)

    plot(x_range, f.(x_range), label="f(x)") # plot f(x)
    plot!(legend=:outertopright, title=prefix * " $(size(split_xs,1))-Order Interpolation")
    series_x = Vector(x_range)
    _, series_y = lagrange(xs, ys, series_x) # compute the interpolation function points
    plot!(series_x, series_y, color=:violet, label="p(x)") # add p(x) function curve
    plot!(ylim=ylim, yflip=false) # add ylim
    # add sample for lagrange interpolation
    plot!(xs, ys, seriestype=:scatter, markersize=3, msw=1, color=:deepskyblue, label="sample")

    test_y = f.(test_x)
    # add test x & y, plot true points
    p = plot!(test_x, test_y, seriestype=:scatter, markersize=3, msw=1, color=:blue,
label="true")
    _, pred_y = lagrange(xs, ys, test_x)
    println()
    data = [test_x test_y pred_y]
    header = ([["Test x", "Test y", "Pred y"]])
    pretty_table(
        data;
        alignment=[:c,:c,:c],
        header=header,
        header_crayon=crayon"bold",
        # tf = tf_markdown,
        formatters=ft_printf("%11.6f"))

    xmin, xmax = xlims(p)
    x = xmax + (xmax - xmin) * 0.04
    y = mean(ylims(p))
    ymin, ymax = ylims(p)
    x2 = xmin + (xmax - xmin) * 0.55
    y2 = y - (ymax - ymin) * 0.25
    annotate!(x, y, text, :black)
    annotate!(x2, y2, comment, :black)
    # add pred_y by lagrange interpolation
    display(plot!(test_x, pred_y, seriestype=:scatter, markersize=3, msw=1, color=:red,
label="approx"))
end

```

测试代码

Test 1 - Simple

首先，使用教材例题作为简单的测试，用于代码正确性基本的检验，检查对于已给点的正确拟合，以及对内插和外插的分别简单测试。

```
test_x = xs = [0, 2, 3, 5, 6]
test_y = ys = [1, 3, 2, 5, 6]
@time _,pred_y = lagrange(xs, ys, test_x)
data = [test_x test_y pred_y]
header = ("Test x", "Test y", "Pred y")
pretty_table(
    data;
    alignment=[:c, :c, :c],
    header=header,
    header_crayon=crayon"bold",
    formatters=ft_printf("%11.6f"))

test_x = [-1, 0, 1, 2, 3, 4, 5, 6, 7]
test_y = [NaN, 1, NaN, 3, 2, NaN, 5, 6, NaN]
@time _, pred_y = lagrange(xs, ys, test_x)
data = [test_x test_y pred_y]
pretty_table(
    data;
    alignment=[:c, :c, :c],
    header=header,
    header_crayon=crayon"bold",
    formatters=ft_printf("%11.6f"))

test_x = xs = [0.4, 0.55, 0.65, 0.80]
test_y = ys = [0.41075, 0.57815, 0.69675, 0.88811]
@time _, pred_y = lagrange(xs, ys, test_x)
data = [test_x test_y pred_y]
pretty_table(
    data;
    alignment=[:c, :c, :c],
    header=header,
    header_crayon=crayon"bold",
    formatters=ft_printf("%11.6f"))
```

0.074727 seconds (52.44 k allocations: 2.636 MiB, 99.91% compilation time)

Test x	Test y	Pred y
0.000000	1.000000	1.000000
2.000000	3.000000	3.000000
3.000000	2.000000	2.000000
5.000000	5.000000	5.000000
6.000000	6.000000	6.000000

0.000022 seconds (59 allocations: 7.094 KiB)

Test x	Test y	Pred y
-1.000000	Nan	-12.200000
0.000000	1.000000	1.000000
1.000000	Nan	4.000000
2.000000	3.000000	3.000000

3.000000	2.000000	2.000000
4.000000	NaN	2.800000
5.000000	5.000000	5.000000
6.000000	6.000000	6.000000
7.000000	NaN	1.000000

0.078833 seconds (52.32 k allocations: 2.653 MiB, 21.31% gc time, 99.93% compilation time)

Test x	Test y	Pred y
0.400000	0.410750	0.410750
0.550000	0.578150	0.578150
0.650000	0.696750	0.696750
0.800000	0.888110	0.888110

Test 2 - Performance

接着，以下测试是为了选用更高效率代码而进行的，用大数组对代码的性能进行评判。

这里由于已经经过了测试，并且本部分运行耗时长，在 Jupyter Notebook 中将代码注释但不删除用以存档。

```
xs = [i for i in -10:0.1:10]
ys = [i^2 for i in -10:0.1:10]
test_x1 = [i for i in -1000:0.01:100]
display(@time xs, ys = lagrange(xs, ys, test_x1))
xs = [i for i in -10:0.1:10]
ys = [i^3 for i in -10:0.1:10]
test_x1 = [i for i in -1000:0.01:100]
display(@time xs, ys = lagrange(xs, ys, test_x1))
xs = [i for i in -10:0.1:10]
ys = [i^4 for i in -10:0.1:10]
test_x1 = [i for i in -1000:0.01:100]
display(@time xs, ys = lagrange(xs, ys, test_x1))
xs = [i for i in -10:0.1:10]
ys = [i^5 for i in -10:0.1:10]
test_x1 = [i for i in -1000:0.01:100]
display(@time xs, ys = lagrange(xs, ys, test_x1))
# display(plot(xs, ys, seriestype=:scatter, markersize=1, msw=0, legend=:outertopright))
```

before code changes:

14.301368 seconds (161.61 k allocations: 66.232 GiB, 8.95% gc time)
14.055990 seconds (161.61 k allocations: 66.232 GiB, 9.19% gc time)
14.121750 seconds (161.61 k allocations: 66.232 GiB, 6.94% gc time)
12.479890 seconds (161.61 k allocations: 66.232 GiB, 7.44% gc time)

after code changes:

13.299191 seconds (162.01 k allocations: 66.396 GiB, 7.29% gc time)
13.303492 seconds (162.01 k allocations: 66.396 GiB, 7.25% gc time)
12.577300 seconds (162.01 k allocations: 66.396 GiB, 6.15% gc time)
12.780377 seconds (162.01 k allocations: 66.396 GiB, 6.49% gc time)

实验题目2 龙贝格(Romberg)积分法

实验简介

本实验为龙贝格积分法，需要完成龙贝格积分代码的编写，并对实验题目进行求解。

本次实验过程中，主要是对龙贝格积分法的算法流程进行调试，从数学原理直接构造算法流程用于计算，充分体会了 Julia 编程语言的流畅性，感受到了编写代码时同 Python 一样自如，却能拥有和 C 相比的循环结构。

实验的目的为使用龙贝格积分法计算定积分，并输出 T 数表。

该实验报告主要分 7 个部分，大纲罗列如下：

- **实验简介**: 即本部分分所有内容
- **数学原理**: 即龙贝格积分法的数学公式，用于改写为算法流程
- **代码实现**: 使用 Julia 编程语言，根据数学原理，编写实验代码
- **实验题目**: 实验指导书中所要求的题目
 - **问题1**: 直接使用龙贝格积分法计算定积分的值
- **思考题**: 本部分为实验指导书中所要求完成的思考题解答
- **参考资料**: 本部分为本次实验过程中查阅的参考资料
- **附录**: 本实验的非主体部分
 - **测试代码**: 教材上的例题用于对程序结果进行初步的检验

数学原理

教材中给出的计算公式如下

$$\begin{cases} T_{0,0} &= \frac{b-a}{2}[f(a) + f(b)], \\ T_{0,i} &= \frac{1}{2}T_{0,i-1} + \frac{1}{2}\frac{b-a}{2^{i-1}}\sum_{j=1}^{2^{i-1}}f\left[a + \left(j - \frac{1}{2}\right)\cdot\frac{b-a}{2^{i-1}}\right], i = 1, 2, 3, \dots, \\ T_{m,k} &= \frac{4^m T_{m-1,k+1} - T_{m-1,k}}{4^m - 1}, m = 1, 2, \dots, i; k = i - m. \end{cases}$$

因 Julia 语言数组类下标的起点为 1，同时实验指导书所给 T 数表为下三角形，故将原公式改写如下

$$\begin{cases} T_{1,1} &= \frac{b-a}{2}[f(a) + f(b)], \\ T_{i+1,1} &= \frac{1}{2}T_{i,1} + \frac{1}{2}\frac{b-a}{2^{i-1}}\sum_{j=1}^{2^{i-1}}f\left[a + \left(j - \frac{1}{2}\right)\cdot\frac{b-a}{2^{i-1}}\right], i = 1, 2, 3, \dots, n \\ T_{i+1,m+1} &= \frac{4^m T_{i+1,m} - T_{i,m}}{4^m - 1}, m = 1, 2, \dots, i. \end{cases}$$

随后可对照公式完成代码的编写

代码实现

使用 Julia 编程语言，根据上述数学原理，编写 romberg 积分法实验代码。

以下部分为 romberg() 函数定义：

```
using Printf
function romberg(f::Function, xlim, n, ε)
    a, b = xlim
    h = b - a
    T = zeros(n, n)
    T[1, 1] = 1 / 2 * h * (f(a) + f(b))
    for i = 1:n
        tmpsum = 0
```

```

jmax = 2^(i - 1)
for j = 1:jmax
    tmpsum += f(a + (j - 1 / 2) * h)
end
T[i+1, 1] = 1 / 2 * T[i, 1] + 1 / 2 * h * tmpsum

for m = 1:i
    T[i+1, m+1] = (4^m * T[i+1, m] - T[i, m]) / (4^m - 1)
end
for m = 1:i
    @printf("%12.9f\t", T[i, m])
end
@printf("\n")
if i > 1 && abs(T[i+1, i+1] - T[i, i]) < ε
    @printf("Accuracy requirement satisfied.\n\n")
    break
end
h /= 2
end
end

```

实验题目

问题 1

```

iter_num = 30

f(x) = x^2 * exp(x)
ε = 1e-6
xlim = 0, 1
println("f(x) = x^2 * exp(x)")
romberg(f, xlim, iter_num, ε)

f(x) = exp(x)sin(x)
ε = 1e-6
xlim = 1, 3
println("f(x) = exp(x)sin(x)")
romberg(f, xlim, iter_num, ε)

f(x) = 4 / (1 + x^2)
ε = 1e-6
xlim = 0, 1
println("f(x) = 4 / (1 + x^2)")
romberg(f, xlim, iter_num, ε)

f(x) = 1 / (x + 1)
ε = 1e-6
xlim = 0, 1
println("f(x) = 1 / (x + 1)")
romberg(f, xlim, iter_num, ε)

```

```

f(x) = x^2 * exp(x)
1.359140914
0.885660616   0.727833850
0.760596332   0.718908238   0.718313197
0.728890177   0.718321459   0.718282340   0.718281850
Accuracy requirement satisfied.

```

```

f(x) = exp(x)sin(x)
5.121826420
9.279762907 10.665741736
10.520554284 10.934151409 10.952045388
10.842043468 10.949206529 10.950210203 10.950181074
10.923093890 10.950110697 10.950170975 10.950170352 10.950170310
Accuracy requirement satisfied.

```

```

f(x) = 4 / (1 + x^2)
3.000000000
3.100000000 3.133333333
3.131176471 3.141568627 3.142117647
3.138988494 3.141592502 3.141594094 3.141585784
3.140941612 3.141592651 3.141592661 3.141592638 3.141592665
Accuracy requirement satisfied.

```

```

f(x) = 1 / (x + 1)
0.750000000
0.708333333 0.694444444
0.697023810 0.693253968 0.693174603
0.694121850 0.693154531 0.693147901 0.693147478
Accuracy requirement satisfied.

```

思考题

1. 略
2. 在实验 1 中二分次数和精度的关系如何?

二分次数越多所求的精度越高，通常预设较大的二分次数来确保计算结果有足够的精度，同时也设定早停需要满足的精度要求，避免达到所需精度之后继续计算导致增加的运算量

3. 略
4. 略

参考资料

1. julia 数值积分 https://blog.csdn.net/m0_37816922/article/details/103475445
2. Romberg Integration-Numerical Analysis <http://homepages.math.uic.edu/~jan/mcs471/romberg.pdf>
3. 《数值分析》吴勃英 196-199

附录

测试代码

本部分使用教材上的例题用于对程序结果进行初步的检验，计算结果和教材给出数表类似，可以认为测试通过。

```

f(x) = x^2 * exp(x)
f(x) = 1 / x
ε = 1e-6
xlim = 1, 3
romberg(f, xlim, 10, ε)

```

```

1.333333333
1.166666667 1.111111111
1.116666667 1.100000000 1.099259259
1.103210678 1.098725349 1.098640372 1.098630548
1.099767702 1.098620043 1.098613022 1.098612588 1.098612518
Accuracy requirement satisfied.

```

实验题目3 四阶龙格-库塔(Runge-Kutta)方法

本实验完成了Runge-Kutta方法的编写，充分学习了微分方程求解、(隐)函数求导库等的用法，巩固了相关理论知识。

实验简介

本实验为Runge-Kutta方法实验，需要完成使用Runge-Kutta方法求解常微分方程初值问题数值解的任务，求解本次各个实验题目问题。

本次实验过程中，主要为对Runge-Kutta方法代码完成编写，并充分体会Runge-Kutta方法的简洁性和相比于Euler方法的在准确性上的优点，同时从绘制的数值解图像注意到n的取值对于结果的重要影响。

实验的目的即为使用Runge-Kutta方法求解常微分方程初值问题的数值解。

该实验报告主要分为7个部分，大纲罗列如下：

- **实验简介**: 即本部分的所有内容
- **数学原理**: 即常微分方程初值问题的数学定义，和对Runge-Kutta方法的基本数学原理进行阐述
- **代码实现**: 使用Julia语言，根据数学原理，编写实验代码
- **实验题目**: 实验指导书中所要求的完成的实验题目，作有便于对照使用Runge-Kutta方法的lib solver 和 my solver 与真实结果 true result 的曲线图，各题目均同时使用lib solver 和 my solver 进行求解，熟悉了Julia库 DifferentialEquations 求解ODE问题的使用流程。

注：详细执行过程请参考附录：执行代码部分

- **问题1**: 探究数值解法与解析解的关系，通过对于解为线性函数和非线性函数的常微分方程的数值求解，体会求出的数值解用于反推解析解的困难程度。
- **问题2&问题3**: 探究n的大小对于求解精度的影响，首先是问题2变化的n对于求解精度的影响几乎可以不计，很容易求得精度较高的解，而在求解问题3时过小的n却根本无法对方程进行求解。这一定程度上说明了，求解的精度和n的选取很大程度上依赖于方程本身的性质。
- **思考题**: 本部分为实验指导书中所要求的完成的思考题解答
- **参考资料**: 本部分为完成实验过程中查阅的参考资料
- **附录**: 本实验的非主体部分
 - **执行代码**: 本部分是实验代码进行运行时封装的部分，将函数的调用细节隐藏在show_result()函数内部，便于直接从外部使用特定参数对函数进行调用。
 - **测试代码**: 对程序的运行、输出进行测试的部分

Test 1 - Simple: 使用教材上的例题对程序的正确性进行简单的测试，确保所写代码能完成实验任务。

数学原理

给定常微分方程初值问题

$$\begin{cases} \frac{dy}{dx} = f(x, y), a \leq x \leq b \\ y(a) = \alpha, h = \frac{b-a}{N} \end{cases}$$

记 $x_n = a + n \cdot h, n = 0, 1, \dots, N$, 利用四阶Runge-Kutta方法，有

$$\begin{aligned} K_1 &= h \cdot f(x_n, y_n) \\ K_2 &= h \cdot f\left(x_n + \frac{h}{2}, y_n + \frac{K_1}{2}\right) \\ K_3 &= h \cdot f\left(x_n + \frac{h}{2}, y_n + \frac{K_2}{2}\right) \\ K_4 &= h \cdot f(x_n + h, y_n + K_3) \end{aligned}$$

$$y_{n+1} = y_n + \frac{1}{6}(K_1 + 2K_2 + 2K_3 + K_4), n = 0, 1, \dots, N-1$$

可以逐次求出微分方程初值问题的数值解 $y_n, n = 1, 2, \dots, N$ 。

代码实现

首先导入需要的包。

`DifferentialEquations.jl` 是用于求解微分方程的标准库，本例中用于获取 `lib solver` 所需的数值解；

`ImplicitEquations.jl` 是用于支持隐函数的标准库，本例中仅在 `Test 1 - Simple` 部分用于支持绘制隐函数图像。

```
using DifferentialEquations
using Plots
using LaTeXStrings
using Statistics
using ImplicitEquations
using PrettyTables
```

根据数学原理和代码流程，可以很容易写出如下代码：

```
function rungekutta(f::Function, xspan, y0, num)
    a, b = xspan
    x0 = a
    h = (b - a) / num
    xs, ys = zeros(num), zeros(num)
    for n = 1:num
        K1 = h * f(x0, y0)
        K2 = h * f(x0 + h / 2, y0 + K1 / 2)
        K3 = h * f(x0 + h / 2, y0 + K2 / 2)
        K4 = h * f(x0 + h, y0 + K3)
        x1 = x0 + h
        y1 = y0 + 1 / 6 * (K1 + 2K2 + 2K3 + K4)
        xs[n], ys[n] = x0, y0 = x1, y1
    end
    xs, ys
end
```

实验题目

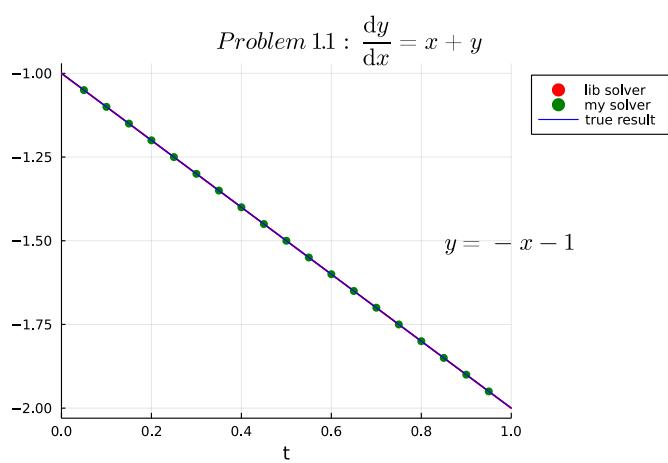
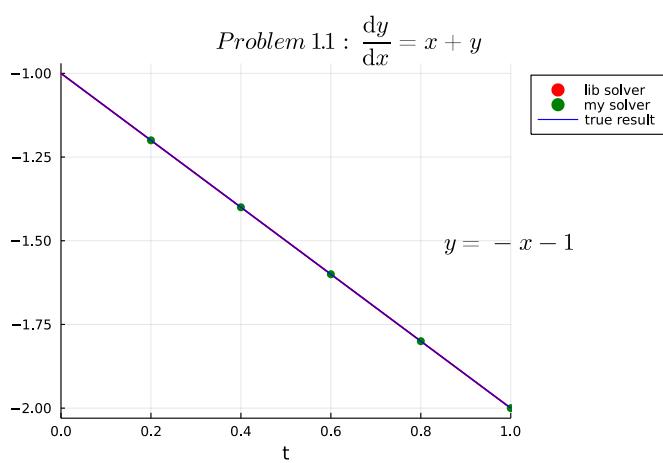
问题 1

1.1

Problem 1.1 $\frac{dy}{dx} = x + y$

```
iternums = [5, 10, 20]

f1(y, p, x) = x + y      # lib RK4() solver
xspan = (0.0, 1.0)
y0 = -1.0
f2(x, y) = x + y          # my rungekutta() solver
f3(x) = -x - 1             # true result
title = L"Problem\ 1.1: \frac{\mathrm{d} y}{\mathrm{d} x} = x + y"
text = L"y = -x - 1"
show_result(f1, f2, f3, xspan, y0, iternums, true, true, title, text) # show, dense
```



Iternum: 5

x	True y	Pred y
0.20000000	-1.20000000	-1.20000000
0.40000000	-1.40000000	-1.40000000
0.60000000	-1.60000000	-1.60000000
0.80000000	-1.80000000	-1.80000000
1.00000000	-2.00000000	-2.00000000

Iternum: 10

x	True y	Pred y
0.10000000	-1.10000000	-1.10000000
0.20000000	-1.20000000	-1.20000000
0.30000000	-1.30000000	-1.30000000
0.40000000	-1.40000000	-1.40000000
0.50000000	-1.50000000	-1.50000000
0.60000000	-1.60000000	-1.60000000
0.70000000	-1.70000000	-1.70000000
0.80000000	-1.80000000	-1.80000000
0.90000000	-1.90000000	-1.90000000
1.00000000	-2.00000000	-2.00000000

Iternum: 20

x	True y	Pred y
0.05000000	-1.05000000	-1.05000000
0.10000000	-1.10000000	-1.10000000
0.15000000	-1.15000000	-1.15000000
0.20000000	-1.20000000	-1.20000000
0.25000000	-1.25000000	-1.25000000
0.30000000	-1.30000000	-1.30000000
0.35000000	-1.35000000	-1.35000000
0.40000000	-1.40000000	-1.40000000
0.45000000	-1.45000000	-1.45000000
0.50000000	-1.50000000	-1.50000000
0.55000000	-1.55000000	-1.55000000
0.60000000	-1.60000000	-1.60000000
0.65000000	-1.65000000	-1.65000000
0.70000000	-1.70000000	-1.70000000
0.75000000	-1.75000000	-1.75000000

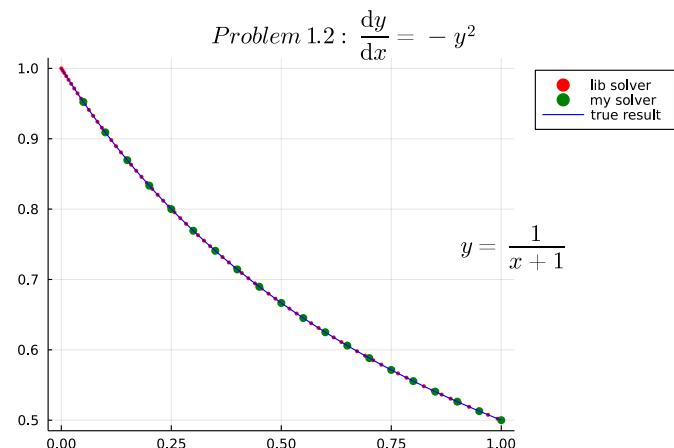
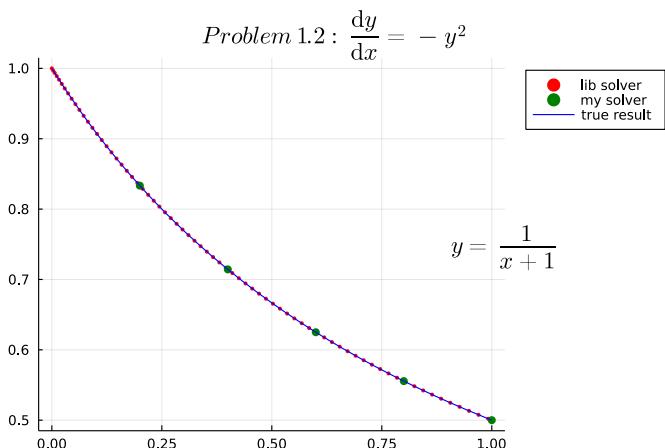
0.80000000	-1.80000000	-1.80000000
0.85000000	-1.85000000	-1.85000000
0.90000000	-1.90000000	-1.90000000
0.95000000	-1.95000000	-1.95000000
1.00000000	-2.00000000	-2.00000000

1.2

Problem 1.2 $\frac{dy}{dx} = -y^2$

```
iternums = [5, 10, 20]

f1(y, p, x) = -y^2
xspan = (0.0, 1.0)
y0 = 1.0
f2(x, y) = -y^2
f3(x) = 1 / (x + 1)
title = L"Problem\ 1.2: \frac{\mathrm{d}y}{\mathrm{d}x} = -y^2"
text = L"y = \frac{1}{x + 1}"
show_result(f1, f2, f3, xspan, y0, iternums, true, false, title, text) # show, dense
```



Iternum: 5

x	True y	Pred y
0.20000000	0.83333333	0.83333904
0.40000000	0.71428571	0.71429213
0.60000000	0.62500000	0.62500589
0.80000000	0.55555556	0.55556069
1.00000000	0.50000000	0.50000441

Iternum: 10

x	True y	Pred y
0.10000000	0.90909091	0.90909119
0.20000000	0.83333333	0.83333373
0.30000000	0.76923077	0.76923121
0.40000000	0.71428571	0.71428615
0.50000000	0.66666667	0.66666709
0.60000000	0.62500000	0.62500040
0.70000000	0.58823529	0.58823567
0.80000000	0.55555556	0.55555590

0.90000000	0.52631579	0.52631611
1.00000000	0.50000000	0.50000030

Iternum: 20

x	True y	Pred y
0.05000000	0.95238095	0.95238096
0.10000000	0.90909091	0.90909093
0.15000000	0.86956522	0.86956524
0.20000000	0.83333333	0.83333336
0.25000000	0.80000000	0.80000003
0.30000000	0.76923077	0.76923080
0.35000000	0.74074074	0.74074077
0.40000000	0.71428571	0.71428574
0.45000000	0.68965517	0.68965520
0.50000000	0.66666667	0.66666669
0.55000000	0.64516129	0.64516132
0.60000000	0.62500000	0.62500003
0.65000000	0.60606061	0.60606063
0.70000000	0.58823529	0.58823532
0.75000000	0.57142857	0.57142859
0.80000000	0.55555556	0.55555558
0.85000000	0.54054054	0.54054056
0.90000000	0.52631579	0.52631581
0.95000000	0.51282051	0.51282053
1.00000000	0.50000000	0.50000002

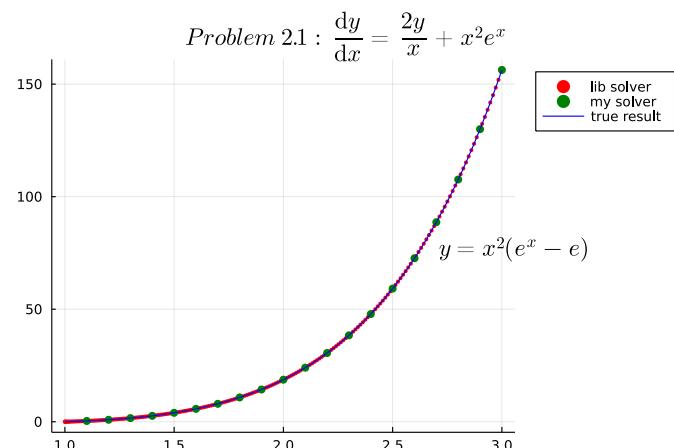
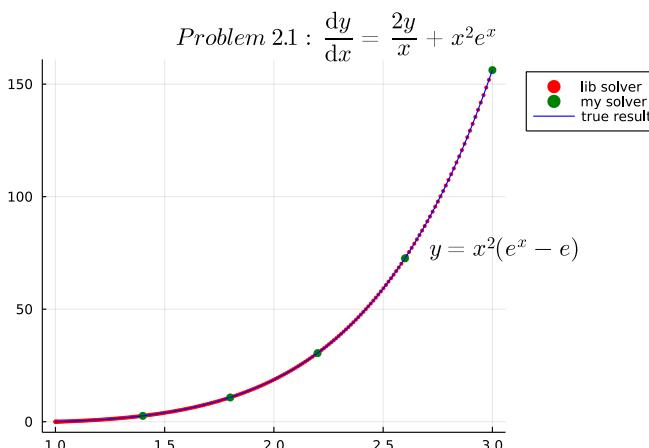
问题 2

2.1

$$Problem 2.1: \frac{dy}{dx} = \frac{2y}{x} + x^2 e^x$$

```
iternums = [5, 10, 20]

f1(y, p, x) = 2 * y / x + x^2 * exp(x)
xspan = (1.0, 3.0)
y0 = 0.0
f2(x, y) = 2 * y / x + x^2 * exp(x)
f3(x) = x^2 * (exp(x) - exp(1))
title = "Problem 2.1: \frac{dy}{dx} = \frac{2y}{x} + x^2 e^x"
text = "y=x^2(e^x - e)"
show_result(f1, f2, f3, xspan, y0, iternums, true, false, title, text) # show, dense
```



Iternum: 5

x	True y	Pred y
1.40000000	2.62035955	2.61394279
1.80000000	10.79362466	10.77631317
2.20000000	30.52458129	30.49165420
2.60000000	72.63928396	72.58559861
3.00000000	156.30529585	156.22519828

Iternum: 10

x	True y	Pred y
1.20000000	0.86664254	0.86637911
1.40000000	2.62035955	2.61974052
1.60000000	5.72096153	5.71989528
1.80000000	10.79362466	10.79201760
2.00000000	18.68309708	18.68085236
2.20000000	30.52458129	30.52159814
2.40000000	47.83619262	47.83236583
2.60000000	72.63928396	72.63450354
2.80000000	107.61470115	107.60885199
3.00000000	156.30529585	156.29825744

Iternum: 20

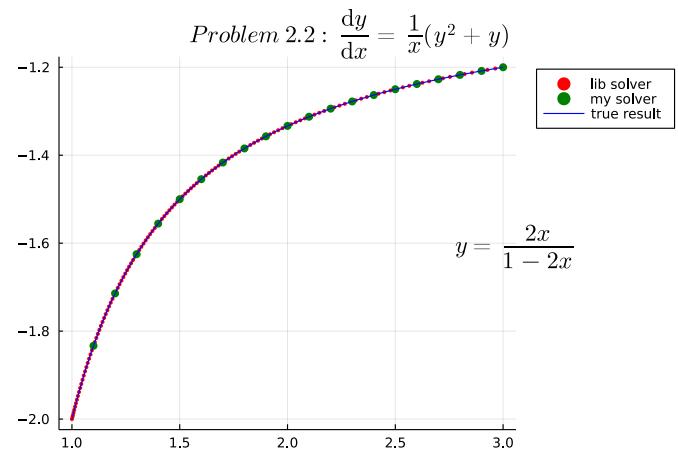
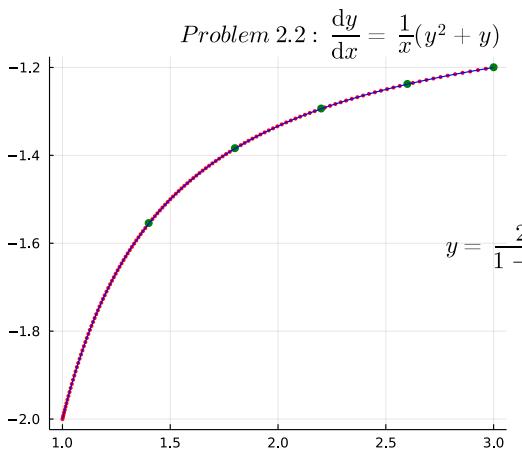
x	True y	Pred y
1.10000000	0.34591988	0.34591029
1.20000000	0.86664254	0.86662169
1.30000000	1.60721508	1.60718135
1.40000000	2.62035955	2.62031131
1.50000000	3.96766629	3.96760190
1.60000000	5.72096153	5.72087932
1.70000000	7.96387348	7.96377179
1.80000000	10.79362466	10.79350178
1.90000000	14.32308154	14.32293573
2.00000000	18.68309708	18.68292657
2.10000000	24.02518645	24.02498942
2.20000000	30.52458129	30.52435589
2.30000000	38.38371431	38.38345866
2.40000000	47.83619262	47.83590478
2.50000000	59.15132583	59.15100383
2.60000000	72.63928396	72.63892578
2.70000000	88.65696974	88.65657333
2.80000000	107.61470115	107.61426439
2.90000000	129.98381238	129.98333312
3.00000000	156.30529585	156.30477188

2.2

Problem 2.2 $\frac{dy}{dx} = \frac{1}{x}(y^2 + y)$

```
iternums = [5, 10, 20]

f1(y, p, x) = (y^2 + y) / x
xspan = (1.0, 3.0)
y0 = -2.0
f2(x, y) = (y^2 + y) / x
f3(x) = 2x / (1 - 2x)
title = "Problem 2.2: \frac{\mathrm{d}y}{\mathrm{d}x} = \frac{1}{x}(y^2+y)"
text = "y=\frac{2x}{1-2x}"
show_result(f1, f2, f3, xspan, y0, iternums, true, false, title, text) # show, dense
```



Iternum: 5

x	True y	Pred y
1.40000000	-1.55555556	-1.55398900
1.80000000	-1.38461538	-1.38361729
2.20000000	-1.29411765	-1.29340153
2.60000000	-1.23809524	-1.23754016
3.00000000	-1.20000000	-1.19954796

Iternum: 10

x	True y	Pred y
1.20000000	-1.71428571	-1.71424518
1.40000000	-1.55555556	-1.55552288
1.60000000	-1.45454545	-1.45451975
1.80000000	-1.38461538	-1.38459451
2.00000000	-1.33333333	-1.33331586
2.20000000	-1.29411765	-1.29410266
2.40000000	-1.26315789	-1.26314480
2.60000000	-1.23809524	-1.23808362
2.80000000	-1.21739130	-1.21738087
3.00000000	-1.20000000	-1.19999054

Iternum: 20

x	True y	Pred y
1.10000000	-1.83333333	-1.83333283
1.20000000	-1.71428571	-1.71428517
1.30000000	-1.62500000	-1.62499950
1.40000000	-1.55555556	-1.55555511
1.50000000	-1.50000000	-1.49999961
1.60000000	-1.45454545	-1.45454510
1.70000000	-1.41666667	-1.41666635
1.80000000	-1.38461538	-1.38461510
1.90000000	-1.35714286	-1.35714260
2.00000000	-1.33333333	-1.33333309
2.10000000	-1.31250000	-1.31249978
2.20000000	-1.29411765	-1.29411744
2.30000000	-1.27777778	-1.27777759
2.40000000	-1.26315789	-1.26315771
2.50000000	-1.25000000	-1.24999983
2.60000000	-1.23809524	-1.23809508
2.70000000	-1.22727273	-1.22727258
2.80000000	-1.21739130	-1.21739116
2.90000000	-1.20833333	-1.20833320
3.00000000	-1.20000000	-1.19999987

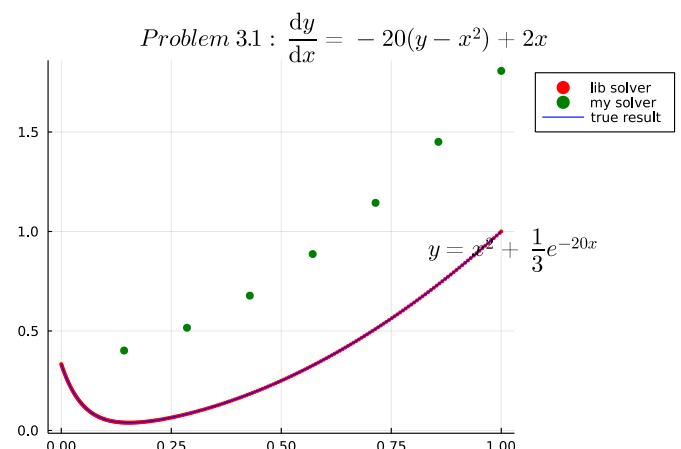
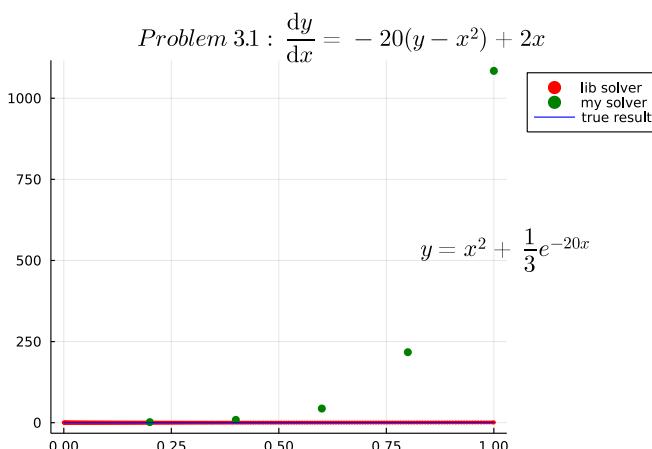
问题 3

3.1

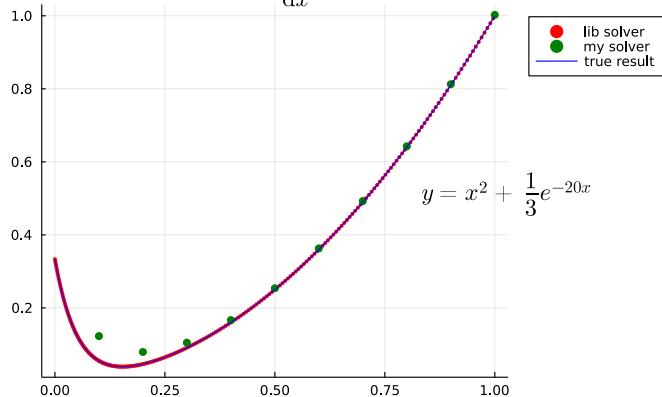
$$\text{Problem 3.1} \frac{dy}{dx} = -20(y - x^2) + 2x$$

```
iternums = [5, 7, 10, 20] # 为观察方便, 添加了n=7的作图, 表格数据仍为所求[5, 10, 20]
```

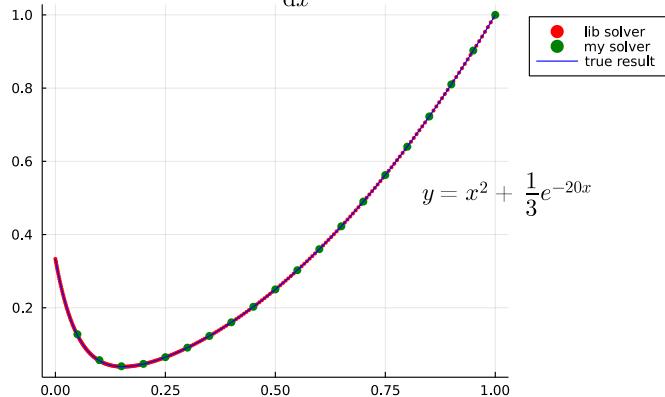
```
f1(y, p, x) = -20(y - x^2) + 2x
xspan = (0.0, 1.0)
y0 = 1 / 3
f2(x, y) = -20(y - x^2) + 2x
f3(x) = x^2 + 1 / 3 * exp(-20x)
title = L"Problem\ 3.1: \frac{\mathbf{d}y}{\mathbf{d}x}=-20(y-x^2)+2x"
text = L"y=x^2+\frac{1}{3}e^{-20x}"
show_result(f1, f2, f3, xspan, y0, iternums, true, false, title, text) # show, dense
```



$$\text{Problem 3.1 : } \frac{dy}{dx} = -20(y - x^2) + 2x$$



$$\text{Problem 3.1 : } \frac{dy}{dx} = -20(y - x^2) + 2x$$



Iternum: 5

x	True y	Pred y
0.20000000	0.04610521	1.76000000
0.40000000	0.16011182	8.81333333
0.60000000	0.36000205	43.68000000
0.80000000	0.64000004	217.29333333
1.00000000	1.00000000	1084.32000000

Iternum: 10

x	True y	Pred y
0.10000000	0.05511176	0.12277778
0.20000000	0.04610521	0.07925926
0.30000000	0.09082625	0.10475309
0.40000000	0.16011182	0.16658436
0.50000000	0.25001513	0.25386145
0.60000000	0.36000205	0.36295382
0.70000000	0.49000028	0.49265127
0.80000000	0.64000004	0.64255042
0.90000000	0.81000001	0.81251681
1.00000000	1.00000000	1.00250560

Iternum: 20

x	True y	Pred y
0.05000000	0.12512648	0.12755208
0.10000000	0.05511176	0.05694661
0.15000000	0.03909569	0.04015706
0.20000000	0.04610521	0.04667348
0.25000000	0.06474598	0.06505464
0.30000000	0.09082625	0.09101007
0.35000000	0.12280396	0.12293086
0.40000000	0.16011182	0.16021366
0.45000000	0.20254114	0.20263220
0.50000000	0.25001513	0.25010166
0.55000000	0.30250557	0.30259021

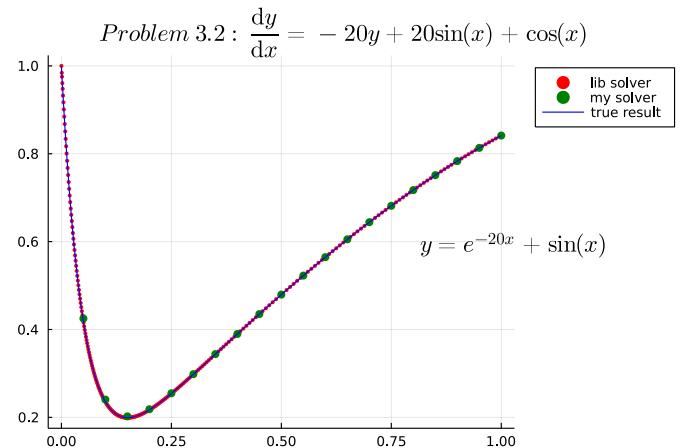
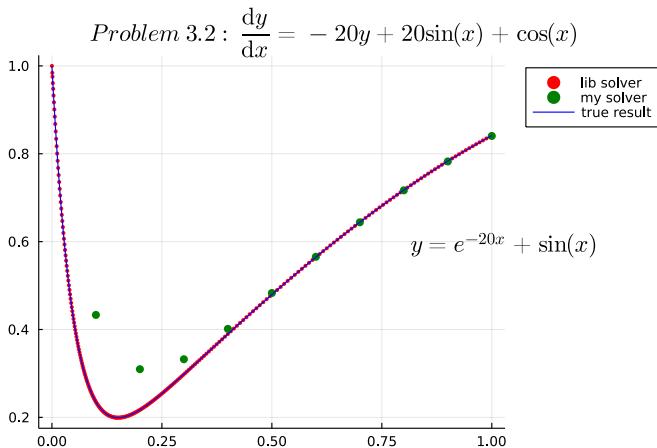
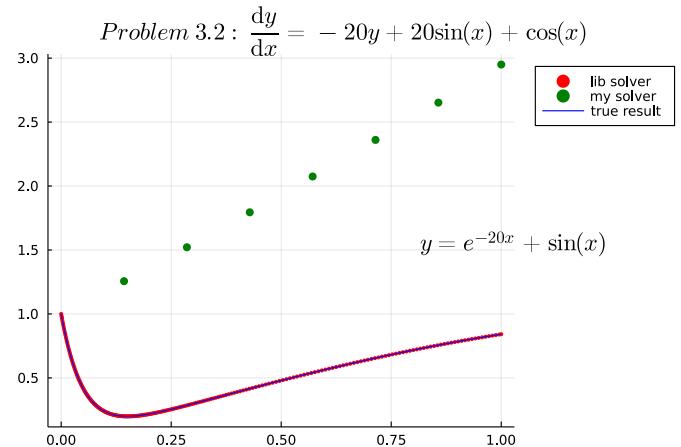
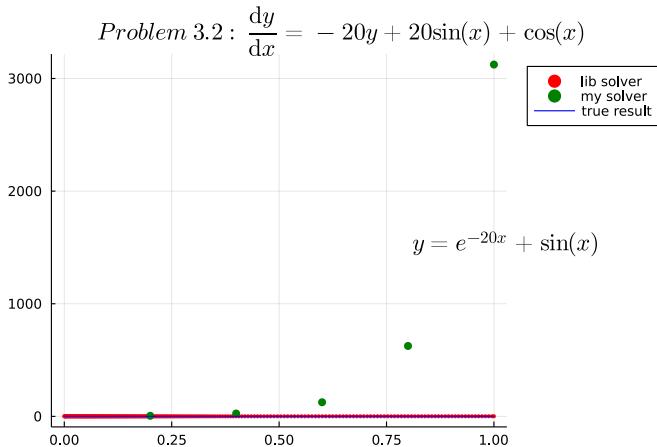
0.60000000	0.36000205	0.36008591
0.65000000	0.42250075	0.42258430
0.70000000	0.49000028	0.49008370
0.75000000	0.56250010	0.56258347
0.80000000	0.64000004	0.64008338
0.85000000	0.72250001	0.72258335
0.90000000	0.81000001	0.81008334
0.95000000	0.90250000	0.90258334
1.00000000	1.00000000	1.00008333

3.2

Problem 3.2 $\frac{dy}{dx} = -20y + 20\sin(x) + \cos(x)$

```
iternums = [5, 7, 10, 20] # 为观察方便, 添加了n=7的作图, 表格数据仍为所求[5, 10, 20]
```

```
f1(y, p, x) = -20y + 20sin(x) + cos(x)
xspan = (0.0, 1.0)
y0 = 1.0
f2(x, y) = -20y + 20sin(x) + cos(x)
f3(x) = exp(-20x) + sin(x)
title = L"Problem\ 3.2: \frac{\mathrm{d} y}{\mathrm{d} x}=-20y+20\sin(x)+\cos(x)"
text = L"y=e^{-20x}+\sin(x)"
show_result(f1, f2, f3, xspan, y0, iternums, true, false, title, text) # show, dense
```



Iternum: 5

x	True y	Pred y
0.20000000	0.21698497	5.19733811
0.40000000	0.38975380	25.37617070
0.60000000	0.56464862	125.48681526
0.80000000	0.71735620	625.31209552
1.00000000	0.84147099	3123.79515095

Iternum: 10

x	True y	Pred y
0.10000000	0.23516870	0.43313900
0.20000000	0.21698497	0.30966047
0.30000000	0.29799896	0.33232467
0.40000000	0.38975380	0.40141397
0.50000000	0.47947094	0.48307434
0.60000000	0.56464862	0.56543528
0.70000000	0.64421852	0.64398900
0.80000000	0.71735620	0.71672235
0.90000000	0.78332692	0.78249915
1.00000000	0.84147099	0.84052572

Iternum: 20

x	True y	Pred y
0.05000000	0.41785861	0.42497852
0.10000000	0.23516870	0.24045622
0.15000000	0.19922520	0.20216844
0.20000000	0.21698497	0.21843866
0.25000000	0.25414191	0.25481165
0.30000000	0.29799896	0.29829102
0.35000000	0.34380969	0.34392855
0.40000000	0.38975380	0.38979534
0.45000000	0.43508894	0.43509617
0.50000000	0.47947094	0.47946262
0.55000000	0.52270393	0.52268809
0.60000000	0.56464862	0.56462864
0.65000000	0.60518867	0.60516599
0.70000000	0.64421852	0.64419376
0.75000000	0.68163907	0.68161253
0.80000000	0.71735620	0.71732804
0.85000000	0.75128045	0.75125076
0.90000000	0.78332692	0.78329581
0.95000000	0.81341551	0.81338305
1.00000000	0.84147099	0.84143727

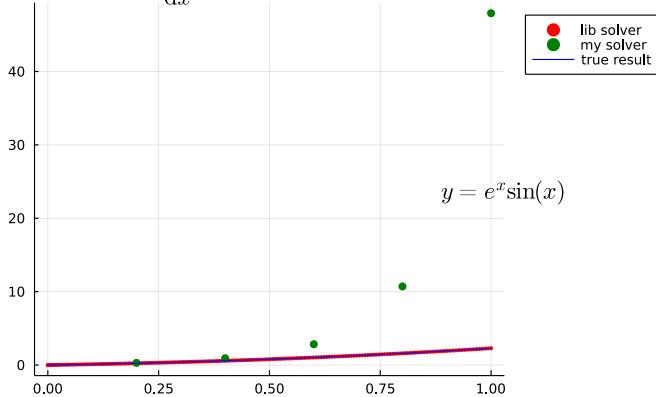
3.3

Problem 3.3 $\frac{dy}{dx} = -20(y - e^x \sin(x)) + e^x(\sin(x) + \cos(x))$

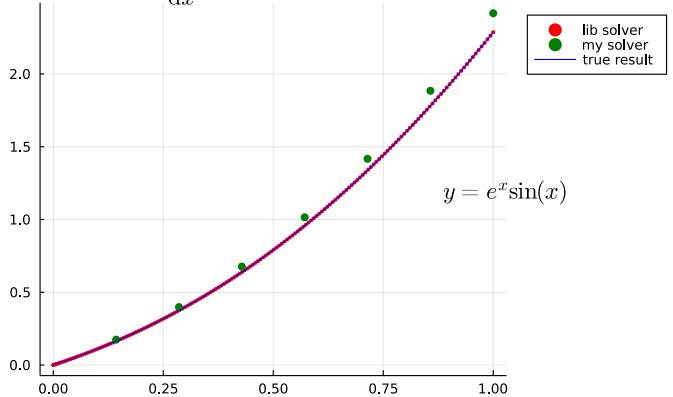
```
iternums = [5, 7, 10, 20] # 为观察方便, 添加了n=7的作图, 表格数据仍为所求[5, 10, 20]
```

```
f1(y, p, x) = -20(y - exp(x)*sin(x)) + exp(x) * (sin(x) + cos(x))
xspan = (0.0, 1.0)
y0 = 0.0
f2(x, y) = -20(y - exp(x)*sin(x)) + exp(x) * (sin(x) + cos(x))
f3(x) = exp(x) * sin(x)
title = L"Problem\ 3.3: \frac{\mathrm{d} y}{\mathrm{d} x}=-20(y-e^x \ \sin(x))+e^x (\sin(x) + \cos(x))"
text = L"y=e^x \sin(x)"
show_result(f1, f2, f3, xspan, y0, iternums, true, false, title, text) # show, dense
```

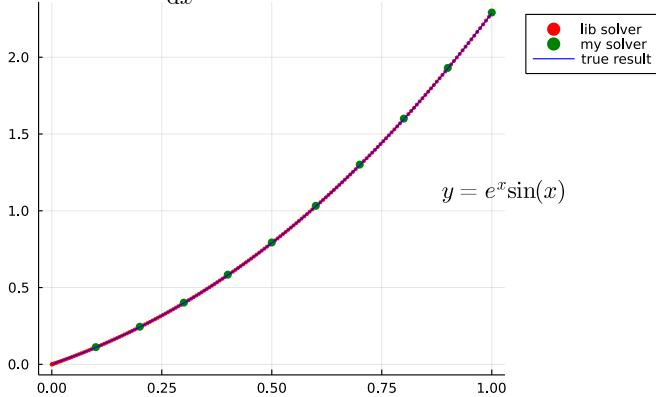
Problem 3.3: $\frac{dy}{dx} = -20(y - e^x \sin(x)) + e^x(\sin(x) + \cos(x))$



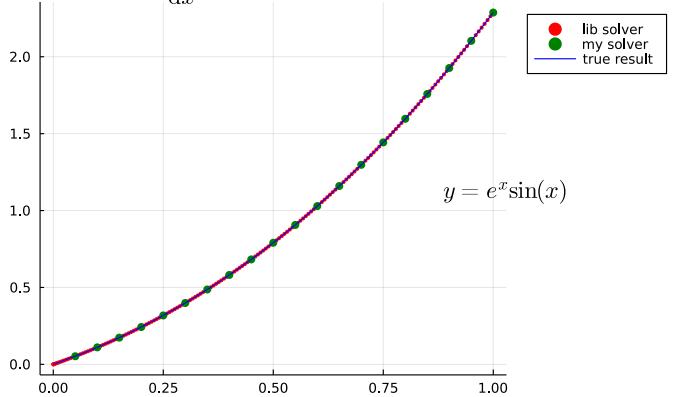
Problem 3.3: $\frac{dy}{dx} = -20(y - e^x \sin(x)) + e^x(\sin(x) + \cos(x))$



Problem 3.3: $\frac{dy}{dx} = -20(y - e^x \sin(x)) + e^x(\sin(x) + \cos(x))$



Problem 3.3: $\frac{dy}{dx} = -20(y - e^x \sin(x)) + e^x(\sin(x) + \cos(x))$



Iternum: 5

x	True y	Pred y
0.20000000	0.24265527	0.29864621
0.40000000	0.58094390	0.92721987
0.60000000	1.02884567	2.83547734
0.80000000	1.59650534	10.71088533
1.00000000	2.28735529	47.94144638

Iternum: 10

x	True y	Pred y
0.10000000	0.11033299	0.11205511

0.20000000	0.24265527	0.24511651
0.30000000	0.39891055	0.40177810
0.40000000	0.58094390	0.58409696
0.50000000	0.79043908	0.79382205
0.60000000	1.02884567	1.03241831
0.70000000	1.29729511	1.30101499
0.80000000	1.59650534	1.60032101
0.90000000	1.92667330	1.93052103
1.00000000	2.28735529	2.29115692

Iternum: 20

x	True y	Pred y
0.05000000	0.05254166	0.05259504
0.10000000	0.11033299	0.11040899
0.15000000	0.17362234	0.17370939
0.20000000	0.24265527	0.24274900
0.25000000	0.31767297	0.31777169
0.30000000	0.39891055	0.39901355
0.35000000	0.48659515	0.48670207
0.40000000	0.58094390	0.58105449
0.45000000	0.68216175	0.68227577
0.50000000	0.79043908	0.79055629
0.55000000	0.90594922	0.90606933
0.60000000	1.02884567	1.02896834
0.65000000	1.15925927	1.15938414
0.70000000	1.29729511	1.29742175
0.75000000	1.44302927	1.44315720
0.80000000	1.59650534	1.59663402
0.85000000	1.75773083	1.75785967
0.90000000	1.92667330	1.92680163
0.95000000	2.10325633	2.10338342
1.00000000	2.28735529	2.28748035

思考题

- 对实验 1，数值解和解析解相同吗？为什么？试加以说明。

对于问题1.1，数值解和解析解是相同的，因为本题的解是线性函数，能够通过所得数值解的两个点确定直线的方程，即等价于得到了解析解。

本例中，待求解微分方程为 $\frac{dy}{dx} = x + y$ ，解为 $y = -x - 1$ ，而 `rungekutta()` 函数求解的任意两点（如 `(0.2, -1.2), (1.0, -2.0)`）所决定的直线方程即为 $y = -x - 1$ 。

而对于问题1.2，虽然数值解和解析解之间差异已经极小（绝对误差在 $1e-7 \sim 1e-5$ 数量级，仅仅对比相同x所在的y取值，如下表所示），但对于非线性函数 $y = \frac{1}{1+x}$ ，在未知函数解析式类型的情况下，是几乎不可能仅仅通过数值解所求得的点来推断准确的函数解析式的，此时不能认为所求得的数值解就是解析解。

Test x	True y	5-Iter Pred y	10-Iter Pred y	20-Iter Pred y
0.20000000	0.83333333	0.83333904	0.83333373	0.83333336
0.40000000	0.71428571	0.71429213	0.71428615	0.71428574
0.60000000	0.62500000	0.62500589	0.62500040	0.62500003
0.80000000	0.55555556	0.55556069	0.55555590	0.55555558
1.00000000	0.50000000	0.50000441	0.50000030	0.50000002

2. 对实验 2, N 越大越精确吗? 试加以说明。

虽然确实N越大越精确, 但从此例实验的结果来看, 因为当n=5的时候已经获得足够精确的数值解了, 再增大n的值只是增加了计算量, 却不能再明显提高结果的精度, 此时我们不能一味的增大N, 而要根据所需要达到的精度要求及时终止计算。

本例中, $y = x^2(e^x - e)$, 在迭代次数从5增加到20的时候, 数值上的精度只增加了2位, 继续增大n对于所求数值解精度改变很小, 很难继续使用Runge-Kutta方法继续进行求解, 并且这样的计算资源成本是不可忽略的。

Test x	True y	5-Iter Pred y	10-Iter Pred y	20-Iter Pred y
1.40000000	2.62035955	2.61394279	2.61974052	2.62031131
1.80000000	10.79362466	10.77631317	10.79201760	10.79350178
2.20000000	30.52458129	30.49165420	30.52159814	30.52435589
2.60000000	72.63928396	72.58559861	72.63450354	72.63892578
3.00000000	156.30529585	156.22519828	156.29825744	156.30477188

3. 对实验 3, N 较小会出现什么现象? 试加以说明

当n较小的时候所得数值解和正确结果相差较大, 结果失真, 说明在一定条件下确实需要更大的n来更好的获得数值解。而具体这个n的大小如何选取则取决于待求解微分方程性质, 这里应该涉及到更深入的课程或者研究。

对本例而言, 从下表以及所绘制的图像都很容易能看到, 当n较小的时候会导致求得数值解偏差极大, 甚至于几乎就完全是错误的(大约与正确结果相差1e3的量级), 所以选择充分大的n, 并设置结果收敛的措施, 才能确保最终可以得到精度合适的数值解的同时不会造成太大的计算资源浪费。

下表为了便于对齐, 略去了多余的x数据, 方程的解析解为 $y = e^{-20x} + \sin(x)$, 数值解如下所示:

x	True y	5-Iter Pred y	10-Iter Pred y	20-Iter Pred y
0.20000000	0.04610521	1.76000000	0.07925926	0.04667348
0.40000000	0.16011182	8.81333333	0.16658436	0.16021366
0.60000000	0.36000205	43.68000000	0.36295382	0.36008591
0.80000000	0.64000004	217.29333333	0.64255042	0.64008338
1.00000000	1.00000000	1084.32000000	1.00250560	1.00008333

以下为方程 $\frac{dy}{dx} = -20(y - e^x \sin(x)) + e^x(\sin(x) + \cos(x))$ 的部分数值解表格, 为便于集中观察而总结如下, 解析解为 $y = e^x \sin(x)$,

x	True y	5-Iter Pred y	10-Iter Pred y	20-Iter Pred y
0.20000000	0.24265527	0.29864621	0.24511651	0.24274900
0.40000000	0.58094390	0.92721987	0.58409696	0.58105449
0.60000000	1.02884567	2.83547734	1.03241831	1.02896834
0.80000000	1.59650534	10.71088533	1.60032101	1.59663402
1.00000000	2.28735529	47.94144638	2.29115692	2.28748035

参考资料

1. julia ordinary differential equations tutorial https://diffeq.sciml.ai/stable/tutorials/ode_example/
2. intro to solving differential equations in julia <https://www.youtube.com/watch?v=KPEqYtEd-zY>
3. julia ode solver type: Runge-Kutta https://diffeq.sciml.ai/stable/solvers/ode_solve/#Explicit-Runge-Kutta-Methods
4. julia ode problem type https://diffeq.sciml.ai/stable/types/ode_types/#ode_prob
5. julia ode speed up perf https://diffeq.sciml.ai/stable/features/performance_overloads/#performance_overload
6. julia ode common solver option https://diffeq.sciml.ai/stable/basics/common_solver_opts/#solver_options
7. 《计算方法实验指导》实验题目 3 四阶龙格—库塔(Runge—Kutta)方法

附录

执行代码

本部分代码用于将需要呈现的结果封装在一个 `show_result()` 函数中，作图时调用重载的三个作图函数 `show_plot()`，分别绘制出 `lib solver`, `my solver` 和 `true result` 的图像，用于观察结果。在运行的循环中，打印出每次执行时的数据，以表格方式呈现。

在本部分之后，是各个问题的逐一求解过程，因题目本身不带更多条件，为标准的常微分方程初值问题求解，故仅按部就班完成了代码的编写和求解，以及结果展示。

为便于区分题目，所绘制的图像中给出了题目的微分方程和标准解的解析式，可供参考。考虑到图片整洁性的原因，略去对于x范围和初值的呈现，前者可直接从x轴范围看出，后者可从标准解的y坐标大致读出。

```
function show_plot(p, f::Function, tspan, u0::Float64, reltol, abstol, dense::Bool)
    prob = ODEProblem(f, u0, tspan)
    alg = RK4()
    sol = solve(prob, alg, reltol=1e-8, abstol=1e-8)
    if dense
        p = plot!(sol, seriestype=:scatter, markersize=1, msw=0, color=:red, label="lib
solver")
    else
        p = plot!(sol.t, sol.u, seriestype=:scatter, markersize=2, msw=0, color=:red,
label="lib solver")
    end
    p, sol
end

function show_plot(p, f::Function, xspan, y0::Float64, iternum::Integer)
    xs, ys = rungekutta(f, xspan, y0, iternum)
    p = plot!(xs, ys, seriestype=:scatter, markersize=4, msw=0, color=:green, label="my
solver")
    p, xs, ys
end

function show_plot(p, f::Function, xs, show::Bool, text)
    x = xlims(p)[2]
    y = mean(ylims(p))
    annotate!(x, y, text, :black)
    if show
        p = plot!(f, color=:blue, label="true result")
    else
        p = plot!(f, color=:blue, label="true result")
    end
    p, xs, f.(xs)
end

function show_result(f1::Function, f2::Function, f3::Function, xspan, y0, iternums, show::Bool,
dense::Bool, title, text)
    println("\n\n" * title)
```

```

for iternum in iternums
    print("\nIternum: $iternum\n")
    p = plot(legend=:outertopright, title=L"~~~~~ * title)
    p, sol = show_plot(p, f1, xspan, y0, 1e-8, 1e-8, dense)
    p, xs, ys = show_plot(p, f2, xspan, y0, iternum)
    p, xt, yt = show_plot(p, f3, xs, show, text)
    data = [xt yt ys]
    header = ("x", "True y", "Pred y")
    pretty_table(
        data;
        alignment=[:c, :c, :c],
        header=header,
        header_crayon=crayon"bold",
        # tf = tf_markdown,
        formatters=ft_printf("%14.8f"))
    display(p)
end
end

```

测试代码

这是一段从教材上选取的测试代码。

待求微分方程为 $\frac{dy}{dx} = y - \frac{2x}{y}$, 解析解为抛物线 $y^2 = 2x + 1$, 编写的 `rungekutta()` 函数进行数值求解时只求解了 $y > 0$ 的情形。

除此以外, 调用 `DifferentialEquations.jl` 库中经 `ODEProblem()` 返回类型重载了的 `solve()` 方法获得了更精确的数值解。

因本部分仅做测试用, 运行过程未经过封装, 略显零乱, 但考虑到与本实验问题求解并无直接关联, 故未作更多修改。

Test 1 - Simple

```

f(y, p, x) = y - 2x / y
xspan = (0.0, 1.0)
y0 = 1.0
prob = ODEProblem(f, y0, xspan)
alg = RK4()
sol = solve(prob, alg, reltol=1e-8, abstol=1e-8)
plot(title=L"~~~~~ Problem: \frac{\mathit{d} y}{\mathit{d} x}=y-\frac{2x}{y}")
plot!(sol.t, sol.u, seriestype=:scatter, markersize=3, msw=0, color=:red, label="lib solver")

f(x, y) = y - 2x / y
println("My Runge-Kutta Solver:")
num = convert(Integer, 1.0 / 0.2)
xs, ys = rungekutta(f, xspan, y0, 5)
yt = .√(2 .* xs .+1)
data = [xs yt ys]
header = ("x", "True y", "Pred y")
pretty_table(
    data;
    alignment=[:c, :c, :c],
    header=header,
    header_crayon=crayon"bold",
    formatters=ft_printf("%14.8f"))
p = plot!(xs, ys, seriestype=:scatter, markersize=5, msw=0, color=:green, label="my solver")

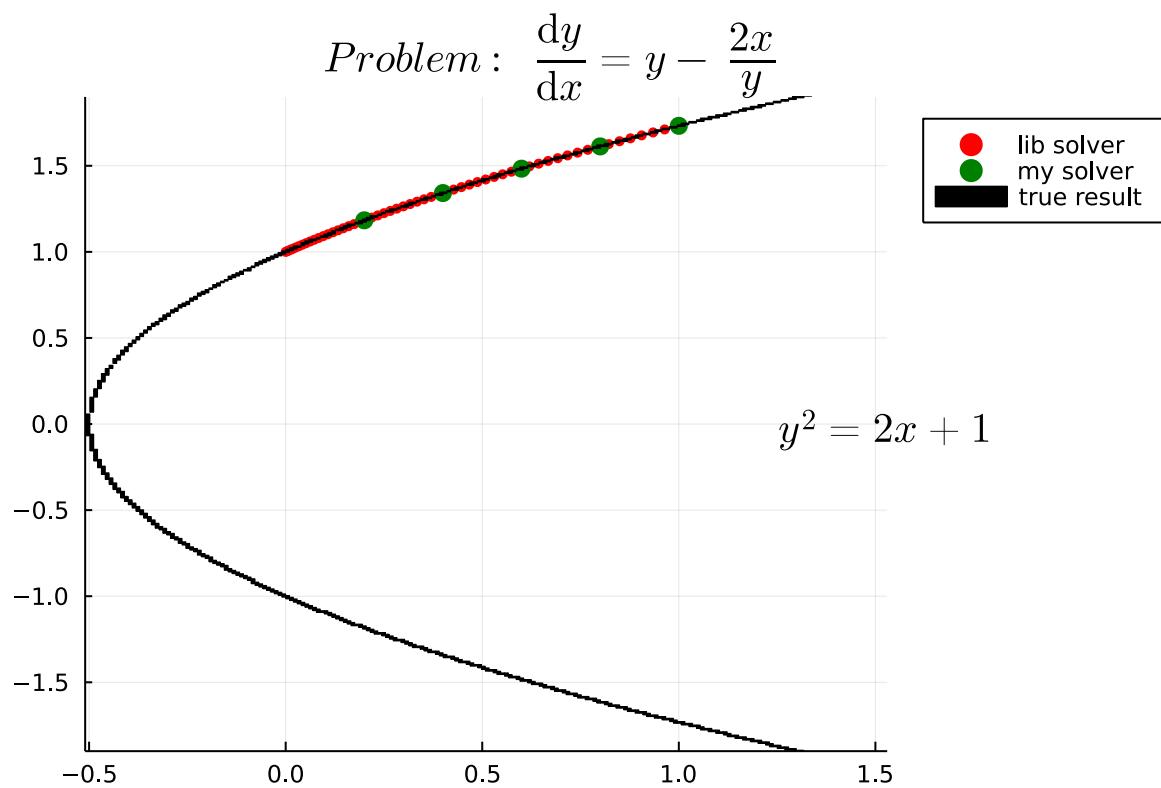
f(x, y) = y^2 - 2x - 1
p = plot!(f == 0.0, color=:green, linewidth=0.1, label="true result") # \Equal[Tab]
p = plot!(legend=:outertopright, xlim=(-0.51, 1.53), ylim=(-1.9, 1.9))

```

```

x = xlims(p)[2]
y = mean(ylims(p))
ymax = ylims(p)[2]
annotate!(x, y, L"y^2=2x+1", :black)
display(p)

```



My Runge-Kutta Solver:

x	True y	Pred y
0.20000000	1.18321596	1.18322929
0.40000000	1.34164079	1.34166693
0.60000000	1.48323970	1.48328146
0.80000000	1.61245155	1.61251404
1.00000000	1.73205081	1.73214188

实验题目4 牛顿(Newton)迭代法

实验简介

本实验为牛顿迭代法求解非线性方程的根，需要编写牛顿迭代法代码，并在给定初值的情况下对实验题目所提供方程进行数值的精确求解。

本次实验过程中，主要为熟悉牛顿迭代法的代码实现，同时对比了重根修正的牛顿迭代法的特点，将牛顿迭代法改写为重根的特殊形式，简化代码。除此之外，本实验过程中对比了 Roots.jl 及 NLsolve.jl 与手写的牛顿法的差异，意外的发现 NLsolve.jl 库中指定 nsolve() 参数为 method="newton" 时求解题目所给方程的效率低于直接手写的牛顿法，不过一定程度上可以理解为 NLsolve.jl 本身为求解高维线性系统的通用工具，对于求解单个非线性方程时有冗余的开销。当然，以 Roots.jl 库的 fzeros() 函数求解结果为精确值进行对比，手写的牛顿法效率和准确度都稍低于经过高度优化的库函数。

实验的目的为使用牛顿迭代法，在给定初值的条件下数值求解非线性方程的根。

该实验报告主要分为6个部分，大纲罗列如下：

- **实验简介**: 即本部分的所有内容
- **数学原理**: 即牛顿迭代法的数学推导和重根修正的牛顿迭代法的数学原理
- **代码实现**: 使用 Julia 语言，根据数学原理，编写实验代码
- **实验题目**: 实验指导书中所要求的进行求解的给定初值的非线性方程，给定求解精度和最大迭代次数后使用牛顿迭代法进行数值求解。各题目输出均打印了 Roots.jl、NLsolve.jl 和本人手写的牛顿法的结果，对比了所求的根的精度、求解耗时、迭代次数等，便于进一步分析。

注：详细执行过程请参考附录：执行代码部分

- **问题1**: 本题为对简单的非线性方程进行求解的问题，直接调用 show_result() 进行调用即可。
- **问题2**: 探究直接采用牛顿法求解带有重根问题的缺陷，通过对比本题两小问求解的结果，加强对理论知识的认识，引出重根修正的牛顿迭代法的使用。同时因本人对比了标准的非线性系统求解库所得结果与手写牛顿法的结果，很容易确定所编写算法的正确性。
- **思考题**: 本部分为实验指导书中所要求完成的思考题解答
- **参考资料**: 本部分为实验过程中查阅的参考资料
- **附录**: 本实验的非主体部分
 - **执行代码**: 本部分为对实验代码进行封装的部分，get_func_diff() 是符号求导的预处理，show_plot() 是根据函数作图的部分，collect_data() 是汇总表格打印所需数据的部分，最后 show_result() 为外界直接调用的部分。

数学原理

牛顿迭代法

设 $y = f(x)$ 有反函数 $x = g(y)$ ，则在 $f(x) = 0$ 的根邻域内， $g(y)$ 关于点 $y_i = f(x_i)$ 的 Taylor 展式为

$$x = g(y) = \sum_{j=0}^{m+1} \frac{(y - y_i)^j}{j!} g^{(j)}(y_i) + \frac{(y - y_i)^{m+2}}{(m+2)!} g^{(m+2)}(\eta_i),$$

其中， η_i 介于 y 与 y_i 之间。因为 $\alpha = g(0)$ ，所以得

$$\begin{aligned} \alpha &= x_i + \sum_{j=0}^{m+1} \frac{(-1)^j}{j!} [f(x_i)]^j g^{(j)}(y_i) + \frac{(-1)^{m+2}}{(m+2)!} [f(x_i)]^{m+2} g^{(m+2)}(\eta_i) \\ &\approx x_i - \frac{f(x_i)}{f'(x_i)} + \sum_{j=2}^{m+1} \frac{(-1)^j}{j!} [f(x_i)]^j g^{(j)}(y_i) \end{aligned}$$

所以可以构造迭代公式

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} + \sum_{j=2}^{m+1} \frac{(-1)^j}{j!} [f(x_i)]^j g^{(j)}(y_i)$$

可以证明

$$\lim_{i \rightarrow \infty} \frac{|\varepsilon_{i+1}|}{|\varepsilon_i|^{m+2}} = \frac{1}{(m+2)!} |f'(\alpha)|^{m+2} |g^{(m+2)}(0)| \neq 0,$$

即迭代公式的收敛阶为 $p = m + 2$, 取 $m = 0$ 时迭代公式为牛顿迭代公式, 即

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}, i = 0, 1, 2, \dots,$$

收敛阶为 $p = 2$, 即

$$\lim_{i \rightarrow \infty} \frac{|\varepsilon_{i+1}|}{|\varepsilon_i|^2} = \frac{|f''(\alpha)|}{2|f'(\alpha)|}.$$

应该指出, *Newton*迭代法是局部收敛的方法, 因此, 他是否收敛, 与初值的选取有关. 当初值 x_0 的选取充分接近根 α 时, 一般可保证迭代收敛.

重根修正的牛顿迭代法

上述讨论是基于 α 为单根的, 若根的重数 $r > 1$, 即 $f(\alpha) = f'(\alpha) = \dots = f^{(r-1)}(\alpha) = 0$, 则所有建立在反函数基础上的推导均无效.

虽然如此, 可以证明*Newton*迭代在重根邻域是线性收敛的.

考虑牛顿迭代法的迭代函数如下

$$\begin{aligned}\varphi(x) &= x - \frac{f(x)}{f'(x)} \\ &= x - \frac{1}{r} \frac{f^{(r)}(\zeta_1)}{f^{(r)}(\zeta_2)} (x - \alpha), \zeta_1, \zeta_2 \text{ 在 } x \text{ 与 } \alpha \text{ 之间.}\end{aligned}$$

考虑到 $\varphi(\alpha) = \alpha$, 于是有

$$\varphi'(\alpha) = \lim_{x \rightarrow \alpha} \frac{\varphi(x) - \varphi(\alpha)}{x - \alpha} = 1 - \frac{1}{r} \neq 0, r \neq 1,$$

由 $r > 1$, $\varphi(\alpha) = 1 - \frac{1}{r} < 1$, 故可知*Newton*迭代法对于 r 重根 α 是一阶收敛的.

接下来, 对*Newton*迭代法进行修正, 当 r 重数已知时, 修正如下

$$x_{i+1} = x_i - r \frac{f(x_i)}{f'(x_i)}, i = 0, 1, 2, \dots,$$

可以证明, 此式对 r 重根 α 是二阶收敛的.

代码实现

packages

首先导入所需要的包。

`NLsolve.jl` 是指定求解非线性系统求解方式的数值求解库, 本例中将指定采用 `newton` 方法对非线性方程进行求解;

`Sympy.jl` 是从 `Python` 移植来的符号运算库, 完全支持 `Python` 中的所有功能, 本例中用于对函数进行符号求导;

`Roots.jl` 是官方提供的求根工具库, 内置封装且进行了优化的在指定区间数值求根的工具 `fzeros()`, 本例中是实验效果最好的求解工具, 精度最高, 耗时最短。

```
using Printf
using Plots
using NLsolve
using SymPy
using Roots
using PrettyTables
using LaTeXStrings
```

newton

本部分为牛顿法的代码实现, 使用重根的迭代法作为外层代码, 牛顿法实现为重根修正的特殊形式, 即 $\lambda = 1$

```
# multi-root newton method
function newton(f::Function, df::Function, ε1, ε2, N, x0, λ)
    n = 1
    while n <= N
        F = f(x0)
        DF = df(x0)
```

```

if abs(F) < ε1
    return n - 1, x0
end
if abs(DF) < ε2
    @printf("Reach a critical point!\n")
    return
end
x1 = x0 - λ * F / DF
Tol = abs(x1 - x0)
if Tol < ε1
    return n - 1, x1
end
n = n + 1
x0 = x1
end
@printf("Fail to converge within %d iterations!\n", n)
end
# newton method
function newton(f::Function, df::Function, ε1, ε2, N, x0)
    newton(f, df, ε1, ε2, N, x0, 1.0)
end

```

实验题目

实验指导书中所要求的进行求解的给定初值的非线性方程，给定求解精度和最大迭代次数后使用牛顿迭代法进行数值求解。

各题目输出均打印了 `Roots.jl`、`NLSolve.jl` 和本人手写的牛顿法的结果，对比了所求的根的精度、求解耗时、迭代次数等，便于进一步分析。

问题 1

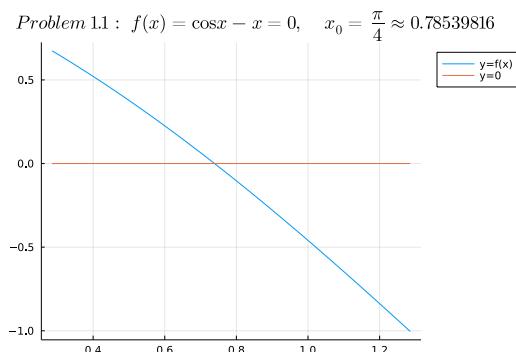
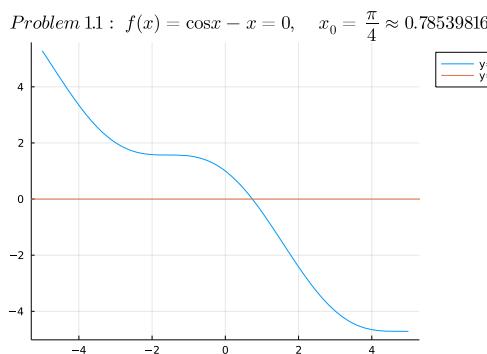
1.1

$$f(x) = \cos x - x = 0, \quad x_0 = \frac{\pi}{4} \approx 0.78539816$$

```

f(x) = cos(x) - x
ε1 = 1e-6
ε2 = 1e-4
N = 10
x0 = pi / 4
title = "Problem 1.1: f(x)=\cos x -x=0,~~~~~x_0=\frac{\pi}{4}\approx 0.78539816"
show_result(f, x0, ε1, ε2, N, title)

```

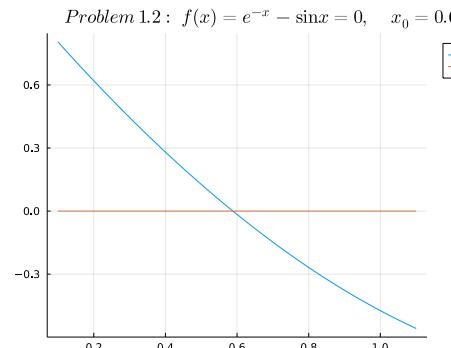
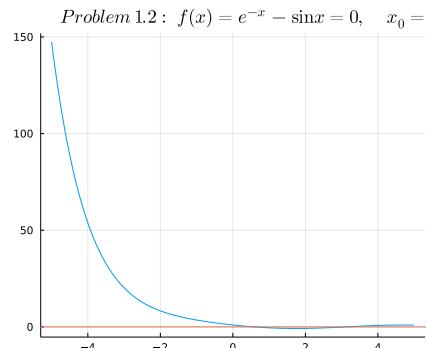


Method	x	f(x)	Time Cost (s)	Iter
lib Roots	0.739085	0.0	0.0047204	NaN
lib NLSolve	0.739085	-7.51299e-8	0.0956659	2.0
my newton	0.739085	-7.51299e-8	0.0195131	2.0

1.2

$$f(x) = e^{-x} - \sin x = 0, \quad x_0 = 0.6$$

```
f(x) = exp(-x) - sin(x)
ε1 = 1e-6
ε2 = 1e-4
N = 10
x0 = 0.6
title = "Problem 1.2:\ f(x)=e^{-x}-\sin x=0,~~~x_0=0.6"
show_result(f, x0, ε1, ε2, N, title)
```



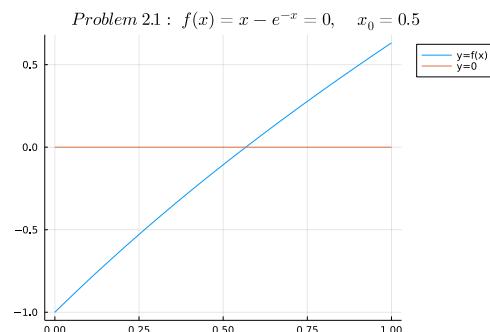
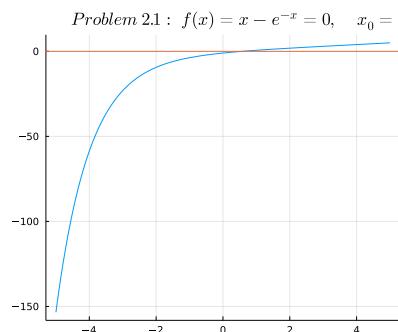
Method	x	f(x)	Time Cost (s)	Iter
lib Roots	0.588533	0.0	0.0041733	NaN
lib NLsolve	0.588533	1.57258e-9	0.0842298	2.0
my newton	0.588533	1.57258e-9	0.0200605	2.0

问题 2

2.1

$$f(x) = x - e^{-x} = 0, \quad x_0 = 0.5$$

```
f(x) = x - exp(-x)
ε1 = 1e-6
ε2 = 1e-4
N = 10
x0 = 0.5
title = "Problem 2.1:\ f(x)=x-e^{-x}=0,~~~x_0=0.5"
show_result(f, x0, ε1, ε2, N, title)
```

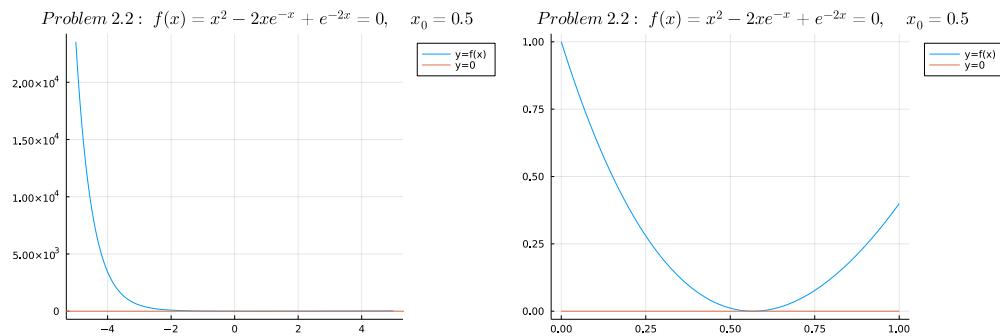


Method	x	f(x)	Time Cost (s)	Iter
lib Roots	0.567143	0.0	0.0038645	NaN
lib NLsolve	0.567143	-1.9648e-7	0.0644682	2.0
my newton	0.567143	-1.9648e-7	0.0200168	2.0

2.2

$$f(x) = x^2 - 2xe^{-x} + e^{-2x} = 0, \quad x_0 = 0.5$$

```
f(x) = x^2 - 2x * exp(-x) + exp(-2x)
ε1 = 1e-6
ε2 = 1e-4
N = 20
x0 = 0.5
λ = 2
title = "Problem 2.2:\ f(x)=x^2 -2x e^{-x} + e^{-2x}=0,~~~~~ x_0=0.5"
show_result(f, x0, λ, ε1, ε2, N, title)
```



Method	x	f(x)	Time Cost (s)	Iter
lib Roots	0.567143	0.0	0.0050046	NaN
lib NLsolve	0.566606	7.09902e-7	0.0675075	7.0
my newton	0.566606	7.09902e-7	0.0201382	7.0
my newton (λ=2)	0.567143	3.86358e-14	0.0156408	2.0

思考题

1. 对问题 1 确定初值的原则是什么？实际计算中应如何解决？

选择一个有根区间，本例中容易得到

$$f(x) = \cos(x) - x, \text{ 有 } f(0) = 1 > 0, f\left(\frac{\pi}{2}\right) = -\frac{\pi}{2} < 0, \text{ 取区间中点即 } x = \frac{\pi}{4} \text{ 为初值}$$

$$f(x) = e^{-x} - \sin(x), \text{ 有 } f(0) = 1 > 0, f(1.2) \approx -0.631 < 0, \text{ 同样取区间中点即 } x = 0.6 \text{ 为初值}$$

实际计算中，根据其他算法求出多个精度较粗的有根区间，然后使用牛顿法逼近获得较为精确的数值解。

通常，对于我们而言，可能在给定区间直接作出函数图像是最简单最可行的方式^{8,9}。

不过也存在一些数值的算法，可以计算给定区间函数根的情况，如 Julia 的 Roots.jl 库中的 zeros() 函数，是本实验中尝试的所有方法中效率最高的，精度最高和计算耗时最短。

除此以外，在我所查找的资料中提及，对于更一般的情形，试图通过程序自动化来计算函数的根的话，情况会变得更加复杂，涉及到多个领域的研究^{10,11}。

2. 对问题 2 如何解释在计算中出现的现象？试加以说明

本例中，

$$(1) f_1(x) = x - e^{-x} = 0$$

Method	x	f(x)	Time Cost (s)	Iter
lib Roots	0.567143	0.0	0.0038645	NaN
lib NLsolve	0.567143	-1.9648e-7	0.0644682	2.0
my newton	0.567143	-1.9648e-7	0.0200168	2.0

$$(2) f_2(x) = x^2 - 2xe^{-x} + e^{-2x} = (x - e^{-x})^2 = f_1^2(x) = 0$$

Method	x	f(x)	Time Cost (s)	Iter
lib Roots	0.567143	0.0	0.0050046	NaN
lib NLsolve	0.566606	7.09902e-7	0.0675075	7.0
my newton	0.566606	7.09902e-7	0.0201382	7.0
my newton ($\lambda=2$)	0.567143	3.86358e-14	0.0156408	2.0

显然，方程(2)在方程(1)的根位置有重根，可以看到直接应用牛顿迭代法计算轮数为7轮，耗时通常情况下稍微增加，但由于当前实例计算简单、精度要求低，耗时变化不明显，不过能看到在给定精度要求情况下所得精度低于无重根牛顿迭代法。

由理论课知识可知，当存在重根时牛顿迭代法的收敛速度为线性收敛。在后续使用修正的牛顿法可以使收敛速度重新达到平方收敛，耗时几乎与无重根时一致，迭代次数和精度也相同。

除此以外，在本实验中，直接手写的牛顿法 `my newton` 运行效率意外的高于库 `NLsolve.jl` 指定 `nlsolve()` 参数 `method=:newton` 后的求解效率，耗时更短，这是让人十分意外的。当然，事实上在同一数量级时间的差异并不大，而 `NLsolve.jl` 库通用用于求解非线性系统，仍然是指定了使用牛顿法来求解实际问题时的优选。而对于一般求解数值根，在给定区间时，使用求解方式得到优化的 `Roots.jl` 库中的 `fzeros()` 函数应当为效率最高的选择。

3. 略

参考资料

1. julia covert sym to func <https://stackoverflow.com/questions/27357579/julia-how-do-i-convert-a-symbolic-expression-to-a-function>
2. julia SymPy Tutorial https://docs.juliahub.com/Sympy/Kzewl/1.0.28/Tutorial/basic_operations/#lambdify-1
3. julia SymPy Tutorial <https://docs.juliahub.com/Sympy/Kzewl/1.0.28/Tutorial/solvers/>
4. julia time elapse <https://discourse.julialang.org/t/difference-between-tic-toc-time-or-elapse-in-julia/1177/2>
5. julia NLsolve github <https://github.com/JuliaNLsolvers/NLsolve.jl>
6. 《数值分析原理》吴勃英 18-19,27-29
7. 《计算方法实验指导》实验题目 4 牛顿(Newton)迭代法
8. how to find initial guess <https://computingskillset.com/solving-equations/how-to-find-the-initial-guess-in-newtons-method/>
9. how to choose starting point <https://math.stackexchange.com/questions/743373/how-to-choose-the-starting-point-in-newtons-method>
10. wikipedia newton fractal https://en.wikipedia.org/wiki/Newton_fractal
11. How to find all roots of complex polynomials by Newton's method <https://link.springer.com/article/10.1007/s002220100149>

附录

执行代码

本部分为对实验代码进行封装的部分，`get_func_diff()` 是符号求导的预处理，`show_plot()` 是根据函数作图的部分，`collect_data()` 是汇总表格打印所需数据的部分，最后 `show_result()` 为外界直接调用的部分。

preprocessing

```
function get_func_diff(symf::Sym)
    @syms x
    symdf = diff(symf)
    f = lambdify(symf)
    df = lambdify(symdf)
    f, df
end
function get_func_diff(f::Function)
    @syms x
    symf = f(x)
    symdf = diff(symf)
    df = lambdify(symdf)
    f, df
end
function redefine_func(f::Function, df::Function)
    function f!(r, x)
        r .= f.(x)
        df.(x)
    end
end
```

```

end
function j!(J, x)
    (s1, s2) = size(J)
    J .= zeros(s1, s1)
    for i in 1:s1
        J[i, i] = df(x[i])
    end
end
f!, j!
end
function show_plots(f::Function, x0, title)
    x = range(start=x0 - 0.5, stop=x0 + 0.5, length=100)
    y = [f.(x), 0 .* x]
    label = ["y=f(x)" "y=0"]
    p = plot(f, label="y=f(x)", title=title, legend=:outertopright)
    p = plot!([0], seriestype=:hline, label="y=0")
    display(p)
    p = plot(x, y, label=label, title=title, legend=:outertopright)
    display(p)
    p
end
function collect_data(f::Function, df::Function, f!::Function, j!::Function,
                      x0, ε1, ε2, N, λ)
    t1 = @elapsed rs = fzeros(f, x0 - 0.5, x0 + 0.5)
    r1 = rs[1]
    i1 = NaN

    t2 = @elapsed sol = nlsolve(f!, j!, [x0]; method=:newton, ftol=ε1, iterations=N)
    r2 = sol.zero[1]
    i2 = sol.iterations[1]

    t3 = @elapsed i3, r3 = newton(f, df, ε1, ε2, N, x0)
    if λ > 1
        t4 = @elapsed i4, r4 = newton(f, df, ε1, ε2, N, x0, λ)
        xs = [r1, r2, r3, r4]
        ts = [t1, t2, t3, t4]
        is = [i1, i2, i3, i4]
        method_name = ["lib Roots", "lib NLsolve", "my newton", "my newton (λ=$λ)"]
        data = [method_name xs f.(xs) ts is]
        return data
    else
        xs = [r1, r2, r3]
        ts = [t1, t2, t3]
        is = [i1, i2, i3]
        method_name = ["lib Roots", "lib NLsolve", "my newton"]
        data = [method_name xs f.(xs) ts is]
        return data
    end
end

```

show_result

```

function show_result(f::Function, x0, ε1, ε2, N, title)
    f, df = get_func_diff(f)
    f!, j! = redefine_func(f, df)
    show_plots(f, x0, L"~~~~~" * title)

    header = ("Method", "x", "f(x)", "Time Cost (s)", "Iter")
    data = collect_data(f, df, f!, j!, x0, ε1, ε2, N, 1)
    pretty_table(
        data;
        alignment=[:c, :c, :c, :c, :c],
        header=header,
        header_crayon=crayon"bold"
    )
end

```

```
function show_result(f::Function, x0, λ, ε1, ε2, N, title)
    f, df = get_func_diff(f)
    f!, j! = redefine_func(f, df)
    show_plots(f, x0, L"~~~~~" * title)

    header = (["Method", "x", "f(x)", "Time Cost (s)", "Iter"])
    data = collect_data(f, df, f!, j!, x0, ε1, ε2, N, λ)
    pretty_table(
        data;
        alignment=[:c, :c, :c, :c, :c],
        header=header,
        header_crayon=crayon"bold"
    )
end
```

实验题目5 高斯(Gauss)列主元消去法

实验简介

本实验为高斯列主元消去法，需要完成使用高斯消元法代码的编写，并对实验题目进行求解。

本次实验过程中，主要为学习高斯消元法的代码写法，同时充分利用矩阵的初等行变换的特点，分别实现了高斯消元、显式消元和隐式消元的 Julia 代码，同时绘制了各写法求解时的时间消耗和求解矩阵阶数的关系图，注意所绘制图像横坐标为样本数而非阶数，每相邻两个样本的阶数之差为5。

实验的目的为使用高斯列主元消元法求解线性方程组。

该实验报告主要分为7个部分，大纲罗列如下：

- **实验简介**: 即本部分的所有内容
- **数学原理**: 即高斯列主元消元法的数学问题定义和求解公式
- **代码实现**: 使用 Julia 语言，根据数学原理，编写实验代码
- **实验题目**: 实验指导书中所要求的求解的线性方程组，以矩阵形式给出，各题目均将输入矩阵打印到控制台以便于观察，这之后同时提供 lib solver 和 my gauss solver 的解，二者结果精确到小数点后6位完全一致，说明求解正确。

注：详细执行过程请参考附录：执行代码部分

- **问题1**: 此题结果实验指导书中已给出，均为 [1, 1, 1, 1]，用于对算法正确性进行检查。
- **问题2**: 此题为线性方程组求解的问题，直接调用写好的求解函数 gauss() 即可。
- **总结**: 无思考题，此处对于本实验代码进行简单的总结，当前问题和后续的优化方向。
- **参考资料**: 本次实验过程中查阅的参考资料
- **附录**: 本实验的非主体部分
 - **执行代码**: 本部分是实验代码进行运行时封装的部分，将函数的调用细节隐藏在 show_result() 函数内部，便于直接从外部使用特定参数对函数进行调用。
 - **测试代码**: 对于程序基本的正确性、性能的测试代码

Test 1 - Correctness: 使用随机生成的高阶矩阵对程序的正确性进行基本的检验，使用 lib solver 与编写的高斯消元结果向量之差的范数来衡量结果的精确程度。

Test 2 - Performance: 首先是各算法求解高阶线性方程组的时间消耗进行统计，然后绘制出方程组求解时间随矩阵阶数变化的折线图，注意横坐标并非阶数而是样本序号，样本从1阶矩阵到4000阶矩阵，每两个连续样本阶数之差为5。

数学原理

给定方程组

$$Ax = b,$$

其中, $A = (a_{ij})_{n \times n}$ 是非奇异阵, $x = (x_1, x_2, \dots, x_n)^T$, $b = (b_1, b_2, \dots, b_n)^T$. 写成分量形式如下

$$\sum_{j=1}^n a_{ij}x_j = b_i, i = 1, 2, \dots, n.$$

为方便讨论，我们将方程组写成如下形式

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = a_{1,n+1} \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = a_{2,n+1} \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = a_{n,n+1} \end{cases}$$

其中，记 $a_{i,n+1} = b_i, i = 1, 2, \dots, n$.

第k次消元为先选择当前第k列最大元素所在的行，然后与第k行进行交换。即先选列主元(*pivoting*)，在代码中由封装的 `pivoting!()` 函数实现该功能，然后交换两行，代码中由 `swaprows!()` 函数实现。

消元时，每次计算当前行首元素与接下来 $n - k + 1$ 行该列元素的比值的倒数 m_{ik} ，然后将此 $n - k + 1$ 行减去当前行（第k行）乘上所求比值，完成当次消元，即

$$\begin{aligned} m_{ik} &= a_{ik}^{(k-1)} / a_{kk}^{(k-1)}, & i &= k+1, k+2, \dots, n. \\ a_{ij} &= a_{ij} - m_{ik} \cdot a_{kj}^{(k-1)}, & i &= k+1, k+2, \dots, n; j = k+1, k+2, \dots, n, n+1 \end{aligned}$$

当消元完成之后，求得上三角矩阵，此后是回代过程，如下所示

$$\begin{cases} x_n = a_{n,n+1}^{(n-1)} / a_{nn}^{(n-1)}, \\ x_i = \left[a_{i,n+1}^{(i-1)} - \sum_{j=i+1}^n a_{ij}^{(i-1)} x_j \right] / a_{ii}^{(i-1)}, i = 1, n-1, n-2, \dots, 2, 1. \end{cases}$$

然后完成代码编写即可。

代码实现

首先导入所需要的包，`LinearAlgebra.jl` 是使用线性代数工具的标准库，包含矩阵特征值求解，矩阵求逆，解线性方程组等内置函数，本例中用运算符 `\` 求解线性方程组。

```
using Printf
using Plots
using LinearAlgebra
using PrettyTables
using Statistics
```

swaprows

```
function swaprows!(X::AbstractMatrix, i::Integer, j::Integer)
    @inbounds for k = 1:size(X, 2)
        X[i, k], X[j, k] = X[j, k], X[i, k]
    end
end
```

pivoting

```
function pivoting!(A::Matrix{Float64}, k::Integer, n::Integer)
    val, idx = findmax(A[k:n, k])
    idx += k - 1 # index must add length omitted by slice
    return val, idx
end

function pivoting!(A::Matrix{Float64}, b::Vector{Float64}, k::Integer, n::Integer,
implicit::Bool)
    s = [maximum(A[i, k:n]) for i in k:n]
    if 0 in s
        println("Cannot solve a singular matrix!")
        return
    end
```

```

if implicit
    val, idx = findmax(A[k:n, k] ./ s[1:n-k+1])
else
    A[k:n, k:n] = A[k:n, k:n] ./ s
    b[k:n] = b[k:n] ./ s
    val, idx = findmax(A[k:n, k])
end
idx += k - 1 # index must add length omitted by slice
return val, idx
end

```

gauss

```

# Gauss列主元消去法
function gauss(n, A::Matrix{Float64}, b::Vector{Float64})
    for k = 1:n-1
        # select pivot in columns
        val, idx = pivoting!(A, k, n)
        if val == 0
            println("Cannot solve a singular matrix!")
            return
        end
        # swap rows
        if idx != k
            swaprows!(A, idx, k)
            b[idx], b[k] = b[k], b[idx]
        end
        # elimination
        for i = k+1:n
            m = A[i, k] / A[k, k]
            A[i, :] -= A[k, :] * m
            b[i] -= b[k] * m
        end
    end
    if A[n, n] == 0
        println("Cannot solve a singular matrix!")
        return
    end
    x = similar(b, Float64)
    x[n] = b[n] / A[n, n]
    for k = n-1:-1:1 # the usage of reverse sequence
        x[k] = (b[k] - dot(A[k, k+1:n], x[k+1:n])) / A[k, k] # something really annoying
    end
    x
end

```

explicit/implicit

```

# Gauss列主元消去法
function gauss(n, A::Matrix{Float64}, b::Vector{Float64}, implicit::Bool)
    for k = 1:n-1
        # select pivot in columns
        val, idx = pivoting!(A, b, k, n, implicit)
        if val == 0
            println("Cannot solve a singular matrix!")
            return
        end
        # swap rows
        if idx != k

```

```

    swaprows!(A, idx, k)
    b[idx], b[k] = b[k], b[idx]
end
# elimination
for i = k+1:n
    m = A[i, k] / A[k, k]
    A[i, :] -= A[k, :] * m
    b[i] -= b[k] * m
end
if A[n, n] == 0
    println("Cannot solve a singular matrix!")
    return
end
x = similar(b, Float64)
x[n] = b[n] / A[n, n]
for k = n-1:-1:1 # the usage of reverse sequence
    x[k] = (b[k] - dot(A[k, k+1:n], x[k+1:n])) / A[k, k] # something really annoying
end
x
end

```

实验题目

实验指导书中所要求的求解的线性方程组，均以矩阵形式给出，各题目均将输入矩阵打印到控制台以便于观察，这之后同时提供 `lib solver` 和 `my gauss solver` 的解，二者结果精确到小数点后6位完全一致，说明求解正确。

本部分在阶数较小的时候，意外的看到手写的 `my gauss solver` 效率高于 `lib solver`，尽管如此，从上图曲线变化很明显可以注意到 `lib solver` 求解效率的稳定性。

问题 1

1.1

```

A = [0.4096 0.1234 0.3678 0.2943
      0.2246 0.3872 0.4015 0.1129
      0.3645 0.1920 0.3781 0.0643
      0.1784 0.4002 0.2786 0.3927]
b = [1.1951; 1.1262; 0.9989; 1.2499]
show_result(A, b)

```

```

input matrix: [A | b]

| 0.40960000  0.12340000  0.36780000  0.29430000 | 1.19510000 |
| 0.22460000  0.38720000  0.40150000  0.11290000 | 1.12620000 |
| 0.36450000  0.19200000  0.37810000  0.06430000 | 0.99890000 |
| 0.17840000  0.40020000  0.27860000  0.39270000 | 1.24990000 |
L

lib solver result:
0.000011 seconds (3 allocations: 384 bytes)

| 1.00000000 |
| 1.00000000 |
| 1.00000000 |
| 1.00000000 |
L

my gauss solver result:
0.000006 seconds (34 allocations: 3.031 KiB)
L

```

```

| 1.00000000 |
| 1.00000000 |
| 1.00000000 |
| 1.00000000 |
└         ┘

```

1.2

```

A = [136.01 90.860      0      0
      90.860 98.810 -67.590      0
            0 -67.590 132.01 46.260
            0      0 46.260 177.17]
b = [226.87; 122.08; 110.68; 223.43]
show_result(A, b)

```

```

input matrix: [A | b]
┌
| 136.01000000 90.86000000 0.00000000 0.00000000 | 226.87000000 |
| 90.86000000 98.81000000 -67.59000000 0.00000000 | 122.08000000 |
| 0.00000000 -67.59000000 132.01000000 46.26000000 | 110.68000000 |
| 0.00000000 0.00000000 46.26000000 177.17000000 | 223.43000000 |
└
lib solver result:
0.000010 seconds (3 allocations: 384 bytes)
┌
| 1.00000000 |
| 1.00000000 |
| 1.00000000 |
| 1.00000000 |
└         ┘
my gauss solver result:
0.000006 seconds (34 allocations: 3.031 KiB)
┌
| 1.00000000 |
| 1.00000000 |
| 1.00000000 |
| 1.00000000 |
└         ┘

```

1.3

```

A = [ 1 1/2 1/3 1/4
      1/2 1/3 1/4 1/5
      1/3 1/4 1/5 1/6
      1/4 1/5 1/6 1/7]
b = [25 / 12; 77 / 60; 57 / 60; 319 / 420]
show_result(A, b)

```

```

input matrix: [A | b]
┌
| 1.00000000 0.50000000 0.33333333 0.25000000 | 2.08333333 |
| 0.50000000 0.33333333 0.25000000 0.20000000 | 1.28333333 |
| 0.33333333 0.25000000 0.20000000 0.16666667 | 0.95000000 |
| 0.25000000 0.20000000 0.16666667 0.14285714 | 0.75952381 |
└
lib solver result:
0.000015 seconds (3 allocations: 384 bytes)
└         ┘

```

```

| 1.00000000 |
| 1.00000000 |
| 1.00000000 |
| 1.00000000 |
└         |

my gauss solver result:
0.000010 seconds (34 allocations: 3.031 KiB)

┌      ┐
| 1.00000000 |
| 1.00000000 |
| 1.00000000 |
| 1.00000000 |
└      ┘

```

1.4

```

A = [10  7  8  7
      7  5  6  5
      8  6 10  9
      7  5  9 10]
b = [32; 23; 33; 31]
show_result(A, b)

```

```

input matrix: [A | b]

┌      ┐
| 10.00000000  7.00000000  8.00000000  7.00000000 | 32.00000000 |
| 7.00000000  5.00000000  6.00000000  5.00000000 | 23.00000000 |
| 8.00000000  6.00000000 10.00000000  9.00000000 | 33.00000000 |
| 7.00000000  5.00000000  9.00000000 10.00000000 | 31.00000000 |
└      ┘

lib solver result:
0.000010 seconds (3 allocations: 384 bytes)

┌      ┐
| 1.00000000 |
| 1.00000000 |
| 1.00000000 |
| 1.00000000 |
└      ┘

my gauss solver result:
0.000006 seconds (34 allocations: 3.031 KiB)

┌      ┐
| 1.00000000 |
| 1.00000000 |
| 1.00000000 |
| 1.00000000 |
└      ┘

```

问题 2

2.1

```

A = [ 197   305  -206  -804
      46.8   71.3 -47.4  52.0
      88.6   76.4 -10.8  802
      1.45   5.90  6.13  36.5]
b = [136; 11.7; 25.1;  6.60]
show_result(A, b)

```

```

input matrix: [A | b]
[ 197.000000000 305.000000000 -206.000000000 -804.000000000 | 136.000000000
| 46.800000000 71.300000000 -47.400000000 52.000000000 | 11.700000000
| 88.600000000 76.400000000 -10.800000000 802.000000000 | 25.100000000
| 1.450000000 5.900000000 6.130000000 36.500000000 | 6.600000000
]

lib solver result:
0.000009 seconds (3 allocations: 384 bytes)
[ 0.95367911 |
| 0.32095685 |
| 1.07870808 |
| -0.09010851 |
]

my gauss solver result:
0.000009 seconds (34 allocations: 3.031 KiB)
[ 0.95367911 |
| 0.32095685 |
| 1.07870808 |
| -0.09010851 |
]

```

2.2

```

A = [0.5398  0.7161 -0.5554 -0.2982
      0.5257  0.6924  0.3565 -0.6255
      0.6465 -0.8187 -0.1872  0.1291
      0.5814  0.9400 -0.7779 -0.4042]
b = [0.2058; -0.0503; 0.1070; 0.1859]
show_result(A, b)

```

```

input matrix: [A | b]
[ 0.53980000  0.71610000 -0.55540000 -0.29820000 |  0.20580000
| 0.52570000  0.69240000  0.35650000 -0.62550000 | -0.05030000
| 0.64650000 -0.81870000 -0.18720000  0.12910000 |  0.10700000
| 0.58140000  0.94000000 -0.77790000 -0.40420000 |  0.18590000
]

lib solver result:
0.000053 seconds (3 allocations: 384 bytes)
[ 0.51617730 |
| 0.41521947 |
| 0.10996610 |
| 1.03653922 |
]

my gauss solver result:
0.000009 seconds (34 allocations: 3.031 KiB)
[ 0.51617730 |
| 0.41521947 |
| 0.10996610 |
| 1.03653922 |
]
```

2.3

```
A = [10  1  2
      1 10  2
      1  1  5]
b = [13; 13; 7]
show_result(A, b)
```

```
input matrix: [A | b]
[ 10.00000000  1.00000000  2.00000000 | 13.00000000 ]
[ 1.00000000 10.00000000  2.00000000 | 13.00000000 ]
[ 1.00000000  1.00000000  5.00000000 |  7.00000000 ]
]

lib solver result:
0.000009 seconds (3 allocations: 288 bytes)
[ 1.00000000 |
  1.00000000 |
  1.00000000 |
]

my gauss solver result:
0.000003 seconds (19 allocations: 1.453 KiB)
[ 1.00000000 |
  1.00000000 |
  1.00000000 |
```

2.4

```
A = [4 -2 -4
      -2 17 10
      -4 10  9]
b = [-2; 25; 15]
show_result(A, b)
```

```
input matrix: [A | b]
[ 4.00000000 -2.00000000 -4.00000000 | -2.00000000 ]
[ -2.00000000 17.00000000 10.00000000 | 25.00000000 ]
[ -4.00000000 10.00000000  9.00000000 | 15.00000000 ]
]

lib solver result:
0.000010 seconds (3 allocations: 288 bytes)
[ 1.00000000 |
  1.00000000 |
  1.00000000 |
]

my gauss solver result:
0.000004 seconds (19 allocations: 1.453 KiB)
[ 1.00000000 |
  1.00000000 |
  1.00000000 |
```

总结

本实验为高斯列主元消元法的实现，以及运用所写代码完成问题的求解，同时熟悉了 `Julia` 语言的一些内置函数，以提高代码运行效率的细节用法，例如：

- 使用 `findmax()` 同时获取向量最大元素及其下标，
- 使用 `@inbounds` 宏减少不必要的边界检查，以节约时间
- 使用 `similar()` 返回内容任意形状相同的矩阵或者向量
- 使用 `norm()` 和 `opnorm()` 分别计算向量和矩阵的范数
- 使用 `@elapsed` 宏获取对应代码运行的时间数值

本次实验代码随能正确完成任务，且在矩阵阶数较低时有高于库函数的运行效率，但较为零散和频繁的内存分配是的函数在处理高阶矩阵时的时间效率远远低于库函数解法。

主要的优化方向为分辨出代码中不必要的内存分配部分，但由于当前 `Julia` 几乎就是默认使用引用传递，不存在 `python` 语言中的内存消耗问题，优化可能还需要参考编译出的汇编代码，可在函数前加 `@code_native` 宏来查看。

这是本实验课的最后一个实验，虽然并非最后完成的实验，但在实验报告的整理过程中，通过之前实验完成过程中学习的 `PrettyTables.jl` 库重写了 `show_result()` 函数以获得更好的结果呈现方式。

通过这半学期的练习，对于 `Julia` 语言各个领域库的使用有了基本的了解，方便了日后的深入使用。

参考资料

1. julia swapcols fast <https://stackoverflow.com/questions/58667332/is-there-a-way-to-swap-columns-in-o1-in-julia>
2. julia _swapcol fast <https://discourse.julialang.org/t/swap-cols-rows-of-a-matrix/47904/9>
3. julia pivoting <https://stackoverflow.com/questions/45396685/what-does-an-exclamation-mark-mean-after-the-name-of-a-function>
4. julia pivoting <https://people.richland.edu/james/lecture/m116/matrices/pivot.html>
5. julia similar <https://stackoverflow.com/questions/62142717/julia-quick-way-to-initialise-an-empty-array-thats-the-same-size-as-another>
6. moving average pseudocode <https://stackoverflow.com/questions/28820904/how-to-efficiently-compute-average-on-the-fly-moving-average>
7. julia repeat method <https://www.geeksforgeeks.org/creating-array-with-repeated-elements-in-julia-repeat-method/>
8. julia repeat usage <http://www.jlhub.com/julia/manual/en/function/repeat>
9. moving average <https://stackoverflow.com/questions/12636613/how-to-calculate-moving-average-without-keeping-the-count-and-data-total>
10. 《数值分析原理》吴勃英 46-48
11. 《计算方法实验指导》实验题目 5 高斯(Gauss)列主元消去法

附录

执行代码

本部分是实验代码进行运行时封装的部分，将函数的调用细节隐藏在 `show_result()` 函数内部，便于直接从外部使用特定参数对函数进行调用。

```
function show_result(A, b)
    n = size(A, 1)
    A = Float64.(A)
    b = Float64.(b)

    println("input matrix: [A | b]")
    data=[A repeat([|], inner=(n, 1)) b]
```

```

pretty_table(
    data;
    header_crayon=crayon"bold",
    tf = tf_matrix,
    noheader=true,
    formatters=ft_printf("%11.8f"))

println("lib solver result:")
pretty_table(
    @time A \ b;
    header_crayon=crayon"bold",
    tf = tf_matrix,
    noheader=true,
    formatters=ft_printf("%11.8f"))

println("my gauss solver result:")
pretty_table(
    @time gauss(n, A, b);
    header_crayon=crayon"bold",
    tf = tf_matrix,
    noheader=true,
    formatters=ft_printf("%11.8f"))
# display(@time gauss(n, A, b, false)) # implicit=false
# display(@time gauss(n, A, b, true)) # implicit=true
end

```

测试代码

本部分为基本的测试代码，用于对代码的正确性和性能进行初步的检查

Test 1 - Correctness

使用随机生成的500阶矩阵对程序的正确性进行基本的检验，使用 `lib solver` 与三种 `gauss solver` 结果向量之差的范数来衡量结果的精确程度。

使用的范数为2范数，计算范数的库函数为 `norm()`，从计算结果来看，绝对误差大约在 $1e-10 \sim 1e-13$ 数量级，可以认为结果相当的精确。

```

# random test
for i in 1:5
    M = rand(500, 500)
    v = rand(500)
    A, b = copy(M), copy(v)
    try
        # default gauss
        @printf("[vector norm] absolute error: %10.6e\n", norm(A \ b - gauss(size(A, 1), A, b),
2))
        # implicit=false
        A, b = copy(M), copy(v)
        @printf("[vector norm] absolute error: %10.6e\n", norm(A \ b - gauss(size(A, 1), A, b,
false), 2))
        # implicit=true
        A, b = copy(M), copy(v)
        @printf("[vector norm] absolute error: %10.6e\n", norm(A \ b - gauss(size(A, 1), A, b,
true), 2))
    catch SingularException
        @printf("Cannot solve a singular matrix!\n")
    end
    println()
end

```

```

[vector norm] absolute error: 1.779781e-12
[vector norm] absolute error: 5.983915e-12
[vector norm] absolute error: 3.740373e-12

[vector norm] absolute error: 2.707323e-12
[vector norm] absolute error: 6.508607e-12
[vector norm] absolute error: 5.296513e-12

[vector norm] absolute error: 2.272272e-12
[vector norm] absolute error: 4.706284e-12
[vector norm] absolute error: 4.289481e-12

[vector norm] absolute error: 2.581938e-11
[vector norm] absolute error: 1.179110e-11
[vector norm] absolute error: 6.608919e-12

[vector norm] absolute error: 6.797016e-12
[vector norm] absolute error: 1.157560e-11
[vector norm] absolute error: 2.105902e-11

```

Test 2 - Performance

首先是对各算法求解500阶线性方程组的时间消耗进行统计，初步得出求解效率的比较结果。然后绘制出方程组求解时间随矩阵阶数变化的折线图，注意横坐标并非阶数而是样本序号，样本从1阶矩阵到4000阶矩阵，每两个连续样本阶数之差为5。

很明显可以看到，求解高阶矩阵的效率有：`lib solver` > `my gsolver` > `g implicit` > `g explicit`，显然内置库对于求解的优化远远超出了本人直接写下的高斯消元函数。

从内存分配上看到库函数仅有的4 `allocations`，明显大大的减少了内存分配所消耗的时间，而手写的高斯消元法中间存在较多的不连续且频繁的内存分配导致时间被大量的消耗。

```

# random test
for i in 1:5
    M = rand(500, 500)
    v = rand(500)
    A, b = copy(M), copy(v)
    try
        @printf("lib solver: ")
        @time A \ b
        @printf("my gsolver: ")
        A, b = copy(M), copy(v)
        @time gauss(size(A, 1), A, b)
        @printf("g explicit: ")
        A, b = copy(M), copy(v)
        @time gauss(size(A, 1), A, b, false) # implicit=false
        @printf("g implicit: ")
        A, b = copy(M), copy(v)
        @time gauss(size(A, 1), A, b, true) # implicit=true
    catch SingularException
        println("Cannot solve a singular matrix!")
    end
    println()
end

```

```

lib solver: 0.009829 seconds (4 allocations: 1.915 MiB)
my gsolver: 0.738466 seconds (500.50 k allocations: 1.936 GiB, 15.91% gc time)
g explicit: 1.210435 seconds (630.65 k allocations: 2.891 GiB, 13.71% gc time)
g implicit: 0.922749 seconds (628.74 k allocations: 2.268 GiB, 12.76% gc time)

```

```

lib solver: 0.010037 seconds (4 allocations: 1.915 MiB)
my gsolver: 0.703981 seconds (500.50 k allocations: 1.936 GiB, 14.03% gc time)
g explicit: 1.219443 seconds (630.65 k allocations: 2.891 GiB, 13.59% gc time)
g implicit: 0.921236 seconds (628.74 k allocations: 2.268 GiB, 12.88% gc time)

lib solver: 0.009741 seconds (4 allocations: 1.915 MiB)
my gsolver: 0.705588 seconds (500.50 k allocations: 1.936 GiB, 13.34% gc time)
g explicit: 1.201793 seconds (630.65 k allocations: 2.891 GiB, 12.86% gc time)
g implicit: 0.936341 seconds (628.74 k allocations: 2.268 GiB, 12.83% gc time)

lib solver: 0.009751 seconds (4 allocations: 1.915 MiB)
my gsolver: 0.709221 seconds (500.50 k allocations: 1.936 GiB, 13.65% gc time)
g explicit: 1.194078 seconds (630.65 k allocations: 2.891 GiB, 13.66% gc time)
g implicit: 0.924525 seconds (628.74 k allocations: 2.268 GiB, 11.92% gc time)

lib solver: 0.008812 seconds (4 allocations: 1.915 MiB)
my gsolver: 0.706725 seconds (500.50 k allocations: 1.936 GiB, 13.59% gc time)
g explicit: 1.175311 seconds (630.65 k allocations: 2.891 GiB, 12.84% gc time)
g implicit: 0.921059 seconds (628.74 k allocations: 2.268 GiB, 12.15% gc time)

```

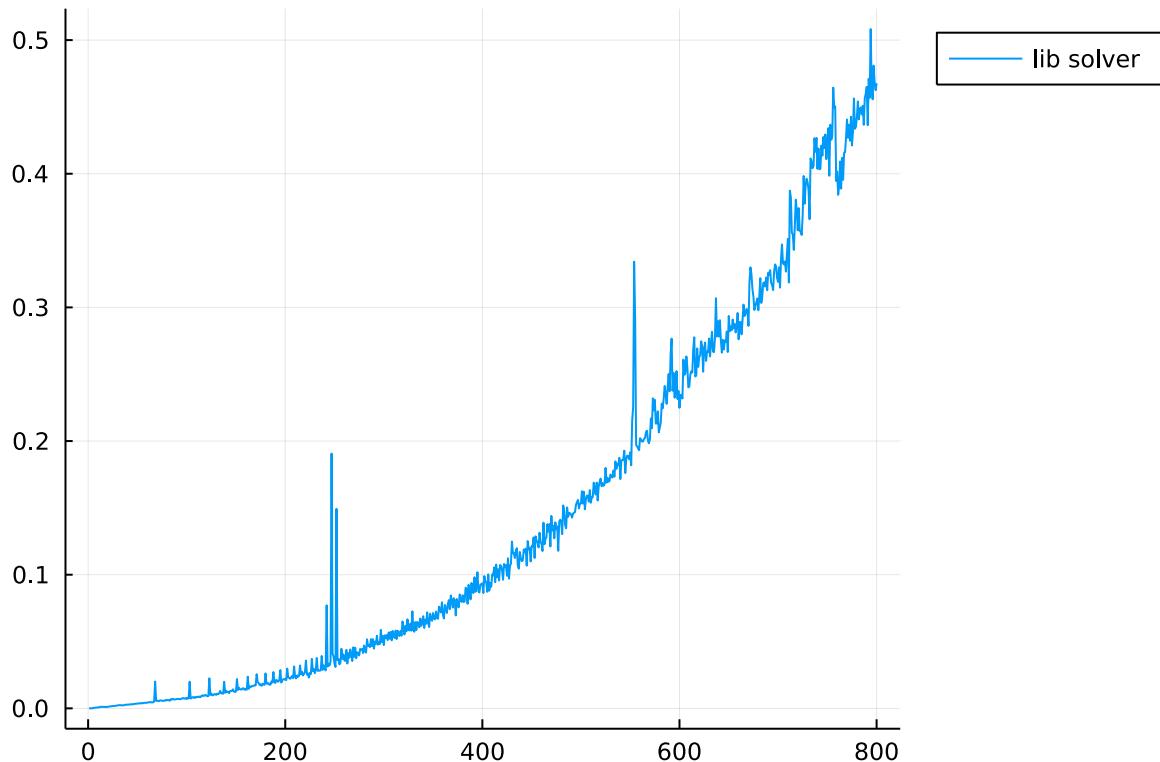
接下来是绘制求解时间随矩阵阶数变化的折线图，横坐标是样本序号，样本从1阶矩阵到4000阶矩阵，每两个连续样本阶数之差为5。因本人手写的高斯消元效率过低，求解阶数大于500的矩阵耗时过长，故绘制图像时有所选择。时间单位为秒，包含了生成矩阵所消耗的时间和求解时间，生成矩阵耗时比库函数求解低一个数量级，故可不多作考虑。

此处性能测试时间消耗过大，又因已经绘制了图像，故在 Jupyter Notebook 中均作注释处理。

```

iter = 1:5:4000
x = [@elapsed(rand(i,i)\rand(i)) for i = iter]
plot(x, label="lib solver", legend=:outertopright)

```



从下图中可以见得本人手写的高斯消元耗时远远超出了库函数解法，仅在阶数低于50（横坐标16附近）时大致能和 lib solver 耗时相当。

```

iter = 1:5:500
x = [@elapsed(rand(i,i)\rand(i)) for i = iter]
display(plot(x, label="lib solver", legend=:outertopright))
x = [@elapsed(gauss(i, rand(i,i), rand(i))) for i = iter]
plot!(x, label="my gsolver")
x = [@elapsed(gauss(i, rand(i,i), rand(i), false)) for i = iter]
plot!(x, label="g explicit")
x = [@elapsed(gauss(i, rand(i,i), rand(i), true)) for i = iter]
plot!(x, label="g implicit")

```

