《数值分析》

实 验 指 导 书

指导老师：李宛珊

数学实验中心

目录

实验 1 线性方程组求解

【实验目的】

（1）熟悉求解线性方程组的有关理论和方法；

（2）能编程实现列主元消去法、LU 分解法、追赶法、Jacobi 迭代法、Gauss-Seidel 迭代法等；

（3）通过测试，进一步了解各种算法的适应条件和优缺点；

（4）根据不同类型的方程组，选择合适的数值方法。

【实验内容】

用 LU 分解法求解给定线性方程组 Ax=b。

输入：系数矩阵 A,

输出：解向量 x。

【实验步骤】

1. LU 分解法—算法公式：

（1）A 的 LU 分解

对于$k=1,2,3,\cdots,n$

$$\begin{cases} U\text{第}k\text{行计算：}u_{k,j}=a_{k,j}-\sum_{r=1}^{k-1} l_{k,r}u_{r,j}\,, & j=k,k+1,\cdots,n, \\ L\text{第}k\text{列计算：}l_{i,k}=\dfrac{1}{u_{k,k}}\left(a_{i,k}-\sum_{r=1}^{k-1} l_{i,k}u_{r,k}\right), & i=k,k+1,\cdots,n. \end{cases}$$

（2）求解 Ly=b（前代法，从第 1 项开始往后算）

$$y_i=b_i-\sum_{k=1}^{i-1} l_{i,k}y_k\,,\qquad i=1,2,\cdots,n$$

（3）求解 Ux=y（回代法，从第 n 项开始往前算）

$$x_i=\frac{1}{u_{i,i}}\left(y_i-\sum_{k=i+1}^{n} u_{i,k}x_k\right),\qquad i=n,n-1,\cdots,1.$$

2. 算法流程：

根据算法特点，计算过程中，将分解后的 L 和 U 的元素仍存储在 A 的相应位置即可。

输入参数：矩阵 A，右端向量 b，输出：解向量 x

（1）对于$k=1,2,3,\cdots,n$

$$a_{k,j}=a_{k,j}-\sum_{r=1}^{k-1}a_{k,r}a_{r,j}\,,\qquad j=k,k+1,\cdots,n,$$

$$a_{i,k}=\frac{1}{a_{k,k}}\left(a_{i,k}-\sum_{r=1}^{k-1}a_{i,k}u_{r,k}\right),\qquad i=k,k+1,\cdots,n.$$

（2）求解 Ly=b（前代法，从第 1 项开始往后算）

$$y_i=b_i-\sum_{k=1}^{i-1}a_{i,k}y_k\,,\qquad i=1,2,\cdots,n$$

（3）求解 Ux=y（回代法，从第 n 项开始往前算）

$$x_i=\frac{1}{u_{i,i}}\left(y_i-\sum_{k=i+1}^{n}a_{i,k}x_k\right),\qquad i=n,n-1,\cdots,1.$$

3. Mworks 代码

```
function Doolittle_m(A::Matrix, b::Vector)
    #Doolittle分解，A=LU，分解后L和U存储在A相应位置
    n=size(A,1) #返回A的行数，即x的维数
    x=zeros(n,1) #初始化x

    #计算L第1列
    for i in 2:n
        A[i,1]=A[i,1]/A[1,1]
    end

    #对于2:n，计算U第k行和L第k列
    for k in 2:n
        #计算U第k行
        for j in k:n
            A[k,j]=A[k,j]-A[k,1:k-1]'*A[1:k-1,j]
        end
        #计算L第k列
        for i in k+1:n
            A[i,k]=(A[i,k]-A[i,1:k-1]'*A[1:k-1,k])/A[k,k]
        end
    end

```

```julia
    #前代法解Ly=b
    y=zeros(n,1)
    y[1]=b[1]
    for k in 2:n
        y[k]=b[k]-A[k,1:k-1]'*y[1:k-1]
    end

    #回代法解Ux=y
    x[n]=y[n]/A[n,n]
    for k in n-1:-1:1
        x[k]=(y[k]-A[k,k+1:n]'*x[k+1:n])/A[k,k]
    end

    return x
end
```

4. 代码运行命令及结果

```
julia> A=rand(3,3)
3×3 Matrix{Float64}:
 0.134556  0.89608   0.929994
 0.506702  0.539744  0.400308
 0.478146  0.706914  0.464959

julia> b=A*ones(3,1)
3×1 Matrix{Float64}:
 1.9606296158047418
 1.4467542036830017
 1.6500191645617779

julia> Doolittle_m(A,b)
3×1 Matrix{Float64}:
 0.9999999999999976
 1.0000000000000029
 0.9999999999999974
```

【实验题目】

Hilbert矩阵$H_n = [h_{ij}] \in R^{n\times n}$，其元素$h_{ij} = \frac{1}{i+j-1}, n = 2,3,4,\cdots$:

(1) 计算$\mathrm{cond}(H_n)_\infty$，分析条件数作为$n$的函数如何变化，

(2) 令$x = (1,1,\cdots,1)^T \in R^n$，计算$b_n = H_n x$，然后用直接法或迭代法解线性方程组$H_n x = b_n$，解为$\tilde{x}$，计算剩余向量$r_n = b_n - H_n\tilde{x}$和误差向量$\tilde{x} - x$，

(3) 对每个n，观察$\tilde{x}$分量有效数字的变化，分析原因，当n为多大时绝对误差达到100%（即$\tilde{x}$连一位有效数字也没有了）。

实验 2 非线性方程求根

【实验目的】

（1）熟悉非线性方程求根的简单迭代法、牛顿迭代法及牛顿下山法；

（2）能编程实现简单迭代法、牛顿迭代法及牛顿下山法

（3）认识选择迭代格式的重要性；

（4）对迭代速度建立感性认识，分析实验结果体会初值对迭代的影响。

【实验内容】

用牛顿下山法解方程$x^3 - x - 4 = 0$（初值在[0,2]中随机生成）。

输入：初值、误差限、最大迭代次数、下山最大次数，

输出：近似根各步下山因子。

【实验步骤】

1. 牛顿下山法—算法公式：

$$x_{k+1} = x_k - \lambda \frac{f(x_k)}{f'(x_k)}, \mathrm{k} = 0,1, \cdots\cdots$$

其中λ为下山因子，一般选取方法，每步迭代计算中，初值为 1，之后每次减半，直至下降条件$f(x_{k+1}) < |f(x_k)|$满足。

2. 算法流程：

输入参数：迭代误差限ε，下山因子取值下限ε_λ，初值x_0（可在指定区间内随机生成），输出：迭代结果

（1）$f_0 = f(x_0)$,

（2）$f_1 = f'(x_0), \lambda = 1$,

（3）$x_1 = x_0 - \lambda \frac{f_0}{f_1}, f_2 = f(x_1)$,

（4）若$|f_2| \geq |f_0|$，且$\lambda > \varepsilon_\lambda, \lambda = \alpha\lambda$，转(3)，

（5）若$|f_2| \geq |f_0|$，且$\lambda < \varepsilon_\lambda$，下山失败，

（6）若$|f_2| < \varepsilon$，或$|x_1 - x_0| < \varepsilon$，输出$x^* = x_1$，迭代终止，

（7）若$|f_2| < |f_0|$，$x_0 = x_1, f_0 = f_2$，转(2).

3. Mworks 代码

```julia
function Newton_downhill(n::Int,a::Float64,b::Float64,e1::Float64,e2::Float64)
    #Newton下山法
    #n为最大允许迭代次数
    #a,b为求解区间
    #e1为|f(x)|<e1迭代终止条件
    #e2为下山因子取值下限

    #定义函数和导数
    function f(x::Float64)
        return x^3-x-4.0
    end
    function df(x::Float64)
        return 3x^2-1.0
    end

    #初始化
    x=zeros(n+1,1)
    x[1]=a+(b-a)*rand()
    lambda=1.0
    i=1
    #迭代
    while true
        f1=f(x[i])
        df_val=df(x[i])
        x[i+1]=x[i]-lambda*f1/df_val;
        f2=f(x[i+1])

        #判断f值是否足够小，是否满足迭代终止条件
        if abs(f2)<e1
            result=x[i+1]
            break;
        end

        #判断下山条件是否满足
        if abs(f2)<abs(f1)
            lambda=1.0;
        elseif abs(f2)>=abs(f1) && lambda>e2
            lambda=lambda/2.0
        elseif abs(f2)>=abs(f1) && lambda<e2
            println("下山失败")
            break
        end

        #判断是否达到最大迭代次数
        if i>=n
            println("已达最大迭代次数")
            break
        end

        i=i+1
    end
    return x
end
```

4. 代码运行命令及结果

```
julia> x=Newton_downhill(20,0.0,2.0,1e-8,1e-4)
21×1 Matrix{Float64}:
 0.7677096026341106
 6.385528698970248
 5.355302450226379
 3.6592232480676214
 2.6038815576715546
 2.032495176795856
 1.8250174212941255
 1.7968206438891399
 ⋮
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
```

【实验题目】

公元1225年，比萨数学家Fibonacci研究了方程

$$x^3+2x^2+10x-20=0,$$

得到一个根$x^*\approx 1.368808107$，没有人知道他是如何得到这个结果的。对于这个方程，分别用简单迭代法、迭代加速方法、Newton迭代法和Newton下山法求解：

（1） 根据计算结果，分析不同迭代方法的收敛速度，

（2） 分析初值的选取对迭代敛散性的影响。

实验 3 数值积分

【实验目的】

（1）熟悉复化梯形法、复化 Simpson 法、龙贝格算法；

（2）能编程实现复化梯形法、复化 Simpson 法、龙贝格算法等；

（3）理解并掌握自适应算法和收敛加速算法的基本思想；

（4）分析实验结果体会各种方法的精确度，建立计算机求解定积分问题的感性认识。

【实验内容】

用龙贝格算法计算积分$\int_a^b \frac{\sin x}{x}\mathrm{d}x$。

输入：积分区间，误差限，

输出：序列T_n, S_n, C_n, R_n及积分结果。

【实验步骤】

1. 龙贝格算法—算法公式：

用$T_1^{(k)}$表示区间[a,b]等距剖分2^k时用复化梯形公式计算数值积分的结果，$T_{m+1}^{(k)}(m=1,2,\cdots)$表示在复化梯形公式计算结果的基础上，m 次外推得到的数值积分结果：

$$
\begin{cases}
T_1^{(0)} = \dfrac{b-a}{2}[f(a)+f(b)], \\
T_1^{(k)} = \dfrac{T_1^{(k-1)}}{2} + \dfrac{(b-a)}{2^k}\sum\limits_{i=1}^{2^{k-1}}\left[f(a+(2i-1)\dfrac{(b-a)}{2^k})\right], \\
T_{m+1}^{(k)} = \dfrac{4^m T_m^{(k+1)} - T_m^{(k)}}{4^m-1}.
\end{cases}
$$

其中$k=0,1,2,\cdots,m=1,2,\cdots$.

根据上面公式可逐行构造下列二维数组表：

k	2^k	$T_1^{(k)}$	$T_2^{(k)}(S_{2^{k-1}})$	$T_3^{(k)}(C_{2^{k-2}})$	$T_4^{(k)}(R_{2^{k-3}})$
0	1	$T_1^{(0)}$			
1	2	$T_1^{(1)}$	$T_2^{(0)}$		
2	4	$T_1^{(2)}$	$T_2^{(1)}$	$T_3^{(0)}$	
3	8	$T_1^{(3)}$	$T_2^{(2)}$	$T_3^{(1)}$	$T_4^{(0)}$
⋮	⋮	⋮	⋮	⋮	⋮

2. 算法流程：

输入：积分区间[a,b]，被积函数 f(x)，精度要求ε

输出：$T_m^{(k)}$二维数组表

（1）置$h = b - a, T_1^{(0)} = \frac{h}{2}[f(a) + f(b)], k = 1,$

（2）计算$T_1^{(k)} = \frac{1}{2}[T_1^{(k-1)} + h\sum_{i=1}^{2^{k-1}} f(a + \left(i - \frac{1}{2}\right)h)]$,

对 $j = 1, \cdots, k$，$T_{j+1}^{(k-j)} = \frac{4^j T_j^{(k-j+1)} - T_j^{(k-j)}}{4^j - 1}$,

（3） 若$|T_k^{(0)}| - |T_{k-1}^{(0)}| < \varepsilon$，停止计算，

否则 $\frac{h}{2} \Rightarrow h, k + 1 \Rightarrow k$，返回 2。

3. Mworks 代码

```
function Romberg_int(a::Float64,b::Float64,e::Float64)
    #k-1为外推次数
    k=0
    n=1 #区间划分个数
    h=b-a #步长
    T=Array{Any}(undef, 0)
    push!(T, [h/2*(f(a)+f(b))]) #梯形公式初始计算值
    err=b-a
    while err>=e
        k=k+1
        h=h/2.0
        tmp=0.0
        for i=1:n
            tmp=tmp+f(a+(2*i-1)*h)
        end
        push!(T, [T[k][1]/2+h*tmp]) #梯形公式的递推
        for j=1:k
            push!(T[k+1], T[k+1][j]+(T[k+1][j]-T[k][j])/(4^j-1)) #外推计算
        end
        n=n*2;
        err=abs(T[k+1][k+1]-T[k][k]) #取对角线相邻两元素差判定计算是否停止
    end

    return T

end

#计算函数f(x)的值
function f(x::Float64)
    return sin(x)/x
end
```

4. 代码运行命令及结果

调试控制台 终端

```
julia> a=3.0;b=4.0;e=1e-5;

julia> Romberg_int(a,b,e)
4-element Vector{Any}:
 [-0.07108031057017983]
 [-0.0856520449550356, -0.09050928974998752]
 [-0.08925281304983623, -0.09045306908143644, -0.09044932103686637]
 [-0.09015041681918348, -0.09044961807563257, -0.09044938800857898, -0.09044938907162203]
```

【实验题目】

1、对于一个“坏”的函数，例如定义在[0,1]上的$\sqrt{x}$，测试 Romberg 算法，并解释为什么它是“坏”函数。

2、用不同的方法计算定积分

$$\int_0^{\frac{\pi}{2}} \frac{1}{\sin^2 x + \frac{1}{4}\cos^2 x} \mathrm{d}x$$

使其误差≤ 10^{-7}（此积分精确值为π），分析和比较不同方法的计算结果。

附：自适应 Simpson 求积程序：

```
function adapint_simpson(f,a0,b0,tol0)
    #自适应求积算法
    int=0;n=1;a=[a0];b=[b0];tol=[tol0];app=[simpson0(f,a0,b0)];k=1;
    while n>0
        k=k+1
        c=(a[end]+b[end])/2.0
        oldapp=app[end]
        app[end]=simpson0(f,a[end],c)
        push!(app, simpson0(f,c,b[end]))

        if abs(oldapp-(app[end-1]+app[end]))<10*tol[end]
            #tol is 10*tol[end] for simpsonint
            int=int+app[end-1]+app[end]
            pop!(a)
            pop!(b)
            pop!(tol)
            n=n-1
        else
            b[end]=c
            push!(b,b[end])
            a[end]=c
            push!(a,a[end])
            tol[end]=tol[end]/2
            push!(tol, tol[end])
            n=n+1
            #plot(a[end],0,'bo',b[end],0,'ro')
            #hold on
        end
```

实验 4 常微分方程数值解法

【实验目的】

（1）熟悉常微分方程初值问题求解的 Euler 法、改进的 Euler 法、Runge-Kutta 法等；

（2）能编程实现 Euler 法、改进的 Euler 法、Runge-Kutta 法等；

（3）通过实验结果计算各算法的收敛阶，分析各个算法的优缺点；

【实验内容】

考虑常微分方程初值问题

$$\begin{cases} y' = 1 + \frac{y}{x} \\ y(1) = 2 \end{cases},$$

取$h = 0.1$，用显式 Euler 法、向后 Euler 法求其数值解并与精确解比较。

输入：求解区间，初值，

输出：数值解。

【实验步骤】

1. 显式 Euler 法—算法公式：

$y_{k+1} = y_k + hf(x_k, y_k)$，

2. 改进的 Euler 法—算法公式：

$$\begin{cases} y_{k+1} = y_k + \frac{1}{2}h(K_1 + K_2), \\ K_1 = f(x_k, y_k), \\ K_2 = f(x_{k+1}, K_1). \end{cases}$$

3. 经典四阶 Runge-Kutta 法—算法公式：

$$\begin{cases} y_{k+1} = y_k + \frac{1}{6}h(K_1 + 2K_2 + 2K_3 + K_4), \\ K_1 = f(x_k, y_k), \\ K_2 = f\left(x_k + \frac{1}{2}h, y_k + \frac{1}{2}hK_1\right), \\ K_3 = f\left(x_k + \frac{1}{2}h, y_k + \frac{1}{2}hK_2\right), \\ K_4 = f(x_k + h, y_k + hK_3). \end{cases}$$

4. Mworks 代码

显式 Euler 法：

```julia
function Euler_forward(a,b,N,ini)
    #显式Euler法
    #[a,b]为求解区域
    #N为剖分网格数
    #h为剖分步长
    f1(x,y)=1+y/x #右端项
    y_exact(x) =x*log(x)+2*x #精确解
    figure(1)
    hold(true)
    fplot(y_exact, [a b], "k-")
    h = (b - a) / N
    x1 = a:h:b
    y1 = similar(x1)
    y1[1] = ini  #初值
    for j in 2 : N+1
    #显式Euler格式
        y1[j]=y1[j-1] +h*f1(x1[j-1],y1[j-1])
    end
    #输出
     figure(1)
     plot(x1,y1,"bo")
     err1 = maximum(abs.(y1 - y_exact.(x1))) #最大模误差
     return err1
end
print(Euler_forward(1,2,10,2), '\n')
print(Euler_forward(1,2,20,2), '\n')
print(Euler_forward(1,2,40,2), '\n')
print(Euler_forward(1,2,80,2), '\n')
```

运行结果:

```
julia> 正在运行 Euler_forward.jl
0.048751554769035366
0.024687597534502004
0.012421881101610133
0.006230469131441652
```

改进的Euler法：

```
function Modi_Euler(a,b,N,ini)
    #改进的Euler法
    #[a,b]为求解区域
    #N为剖分网格数
    #h为剖分步长
    f1(x,y)=1+y/x #右端项
    y_exact(x) =x*log(x)+2*x #精确解
    figure(1)
    hold(true)
    fplot(y_exact, [a b], "k-")
    h = (b - a) / N
    x1 = a:h:b
    y1 = similar(x1)
    y1[1] = ini  #初值
    for j in 2 : N+1
    #改进的Euler法
        K1=f1(x1[j-1], y1[j-1])
        K2=f1(x1[j], y1[j-1]+h*K1)
        y1[j]=y1[j-1]+h/2*(K1+K2)
    end
    #输出
     figure(1)
     plot(x1,y1,"bo")
     err1 = maximum(abs.(y1 - y_exact.(x1))) #最大模误差
     return err1
end
print(Modi_Euler(1,2,10,2), '\n')
print(Modi_Euler(1,2,20,2), '\n')
print(Modi_Euler(1,2,40,2), '\n')
print(Modi_Euler(1,2,80,2), '\n')
```

运行结果:

```
julia> 正在运行 Modi_Euler.jl
0.0023560420225514633
0.0006068784405908819
0.00015397770981583372
3.877806056706845e-5
```

经典四阶Runge-Kutta法：

```julia
function RungeKutta_4(a,b,N,ini)
    #改进的Euler法
    #[a,b]为求解区域
    #N为剖分网格数
    #h为剖分步长
    f1(x,y)=1+y/x #右端项
    y_exact(x) =x*log(x)+2*x #精确解
    figure(1)
    hold(true)
    fplot(y_exact, [a b], "k-")
    h = (b - a) / N
    x1 = a:h:b
    y1 = similar(x1)
    y1[1] = ini  #初值
    for j in 2 : N+1
    #改进的Euler法
        K1=f1(x1[j-1], y1[j-1])
        K2=f1(x1[j-1]+h/2, y1[j-1]+h/2*K1)
        K3=f1(x1[j-1]+h/2, y1[j-1]+h/2*K2)
        K4=f1(x1[j-1]+h, y1[j-1]+h*K3)
        y1[j]=y1[j-1]+h/6*(K1+2*K2+2*K3+K4)
    end
     #输出
      figure(1)
      plot(x1,y1,"bo")
      err1 = maximum(abs.(y1 - y_exact.(x1))) #最大模误差
      return err1
end
print(RungeKutta_4(1,2,10,2), '\n')
print(RungeKutta_4(1,2,20,2), '\n')
print(RungeKutta_4(1,2,40,2), '\n')
print(RungeKutta_4(1,2,80,2), '\n')
```

运行结果:

```
julia> 正在运行 RungeKutta_4.jl
1.4747674308424052e-6
9.501746234263919e-8
6.022818688222742e-9
3.7897507354500704e-10
```

【实验题目】

用显式Euler法、隐式Euler法、梯形法、经典四阶Runge-Kutta法解常微分方程初值问题：

$$\begin{cases} y'(x) = -50y(x) + 49\sin x + 51\cos x, & 0 < x \le 10, \\ y(0) = 1. \end{cases}$$

取不同的步长h，对不同算法的稳定性和精度等进行比较分析。