nb-l08-linsolve

November 21, 2019

Scientific Computing, TU Berlin, WS 2019/2020, Lecture 08

Jürgen Fuhrmann, WIAS Berlin

1 Homework 01

1.1 Homework issues (for next time)

- Please zip files such that they unpack into subdirecotry
- Maybe remove ___MacOSX folders before
- Always put labels onto plots
- Please no function calls outside of module.
- Please have proper file extensions for julia: `.jl`
- I don't like spaces in file names
- I's not a good idea to Pkg.add in some modules.

1.2 Homework Task 1

- Write a Julia program which calculates $\sum_{n=1}^{K}\frac{1}{n^2}$ for $K = 10, 100, 1000, 10000, 100000$ and report the values for `Float16`,`Float32`, `Float64`. Compare the results to the value of $\sum_{n=1}^{\infty}\frac{1}{n^2}$
- We have $\sum_{n=1}^{\infty}\frac{1}{n^2} = \frac{\pi^2}{6}$.

[1]:
```
function baselsum(T::Type, n)
    s=zero(T)
    eins=one(T)
    for k=1:n
        x=T(k)
        s+=eins/(x*x)
    end
    s
end

baselerror(T::Type,n)= abs(baselsum(T,n)-pi^2/6)
```

[1]:
```
baselerror (generic function with 1 method)
```

[2]:
```
@show baselerror(Float16,10000)
@show baselerror(Float32,10000)
```

```
@show baselerror(Float64,10000)
@show baselerror(BigFloat,10000)
```

```
baselerror(Float16, 10000) = 0.017980941848226406
baselerror(Float32, 10000) = 0.0002087441248377342
baselerror(Float64, 10000) = 9.999500016122376e-5
baselerror(BigFloat, 10000) = 9.999500016663625960982935095467830823966312034574
690979758265123362934550706695e-05
```

[2]:
```
9.999500016663625960982935095467830823966312034574690979758265123362934550706695
e-05
```

- Timings

[3]:
```
using BenchmarkTools
```

[4]:
```
@btime baselerror(Float16,10000)
@btime baselerror(Float32,10000)
@btime baselerror(Float64,10000)
@btime baselerror(BigFloat,10000)
```

```
  5.736 ms (49489 allocations: 773.27 KiB)
  8.915  s (0 allocations: 0 bytes)
  8.718  s (0 allocations: 0 bytes)
  8.377 ms (89497 allocations: 4.42 MiB)
```

[4]:
```
9.999500016663625960982935095467830823966312034574690979758265123362934550706695
e-05
```

1.2.1 How to improve ?

- Reverse summation
- Sum up smalles contributions first, so they won't be cancelled by larger ones

[5]:
```
function rbaselsum(T::Type, n)
    s=zero(T)
    eins=one(T)
    for k=n:-1:1
        x=T(k)
        s+=eins/(x*x)
    end
    s
end

rbaselerror(T::Type,n)= abs(rbaselsum(T,n)-pi^2/6)
```

[5]:
```
rbaselerror (generic function with 1 method)
```

[6]:
```
@show rbaselerror(Float16,10000)
@show rbaselerror(Float32,10000)
@show rbaselerror(Float64,10000)
@show rbaselerror(BigFloat,10000)
```

```
rbaselerror(Float16, 10000) = 0.004309066848226406
rbaselerror(Float32, 10000) = 0.00010002525276742169
rbaselerror(Float64, 10000) = 9.999500016677487e-5
rbaselerror(BigFloat, 10000) = 9.999500016663625960982935095467830823966312034574690979758265123362934569706266e-05
```

[6]:
```
9.999500016663625960982935095467830823966312034574690979758265123362934569706266e-05
```

- Kahan summation

[7]:
```
function kbaselsum(T::Type, n)
    sum=zero(T)
    error_compensation=zero(T)
    eins=one(T)
    for k=1:n
        x=T(k)
        increment=eins/(x*x)
        corrected_increment=increment-error_compensation;
        good_sum=sum+corrected_increment;
        error_compensation= (good_sum-sum)-corrected_increment;
        sum=good_sum
    end
    sum
end
kbaselerror(T::Type,n)= abs(kbaselsum(T,n)-pi^2/6)
```

[7]:
```
kbaselerror (generic function with 1 method)
```

[8]:
```
@show kbaselerror(Float16,10000)
@show kbaselerror(Float32,10000)
@show kbaselerror(Float64,10000)
@show kbaselerror(BigFloat,10000)
```

```
kbaselerror(Float16, 10000) = 0.004309066848226406
kbaselerror(Float32, 10000) = 0.00010002525276742169
kbaselerror(Float64, 10000) = 9.999500016655283e-5
kbaselerror(BigFloat, 10000) = 9.999500016663625960982935095467830823966312034574690979758265123362934569706266e-05
```

[8]:
```
9.999500016663625960982935095467830823966312034574690979758265123362934569706266e-05
```

Plotting…

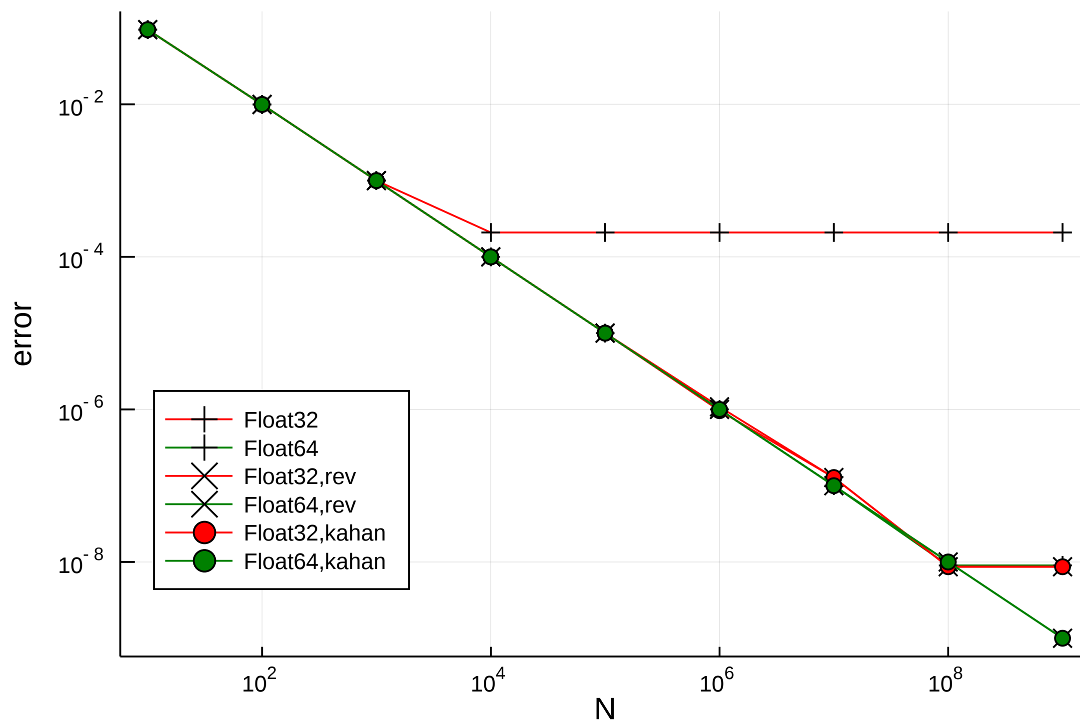
[9]:
```
using Plots

Ns=[10^i for i=1:9]
p=plot(xlabel="N",ylabel="error",xaxis=:log, yaxis=:log,legend=:bottomleft)
plot!(p,Ns,baselerror.(Float32,Ns),label="Float32",markershape=:+,color=:red)
plot!(p,Ns,baselerror.(Float64,Ns),label="Float64",markershape=:+,color=:green)

plot!(p,Ns,rbaselerror.(Float32,Ns),label="Float32,rev",markershape=:x,color=:
↪red)
plot!(p,Ns,rbaselerror.(Float64,Ns),label="Float64,rev",markershape=:x,color=:
↪green)

plot!(p,Ns,kbaselerror.(Float32,Ns),label="Float32,kahan",markershape=:
↪circle,color=:red)
plot!(p,Ns,kbaselerror.(Float64,Ns),label="Float64,kahan",markershape=:
↪circle,color=:green)
```

[9]:



1.3 Homework Task 2

$$
\begin{aligned}
-u'' &= 1 \quad \texttt{in}\ \Omega \\
-u'(0) + \alpha(u(0) - v_L) &= 0 \\
u'(1) + \alpha(u(1) - v_R) &= 0
\end{aligned}
$$

- Assume $f = 1, v_L = 0, v_R = 0$

- Interior:

$$
\begin{aligned}
-u' &= x + C \\
u(x) &= -\frac{1}{2}x^2 - Cx + D
\end{aligned}
$$

- Left boundary condition:

$$
\begin{aligned}
-u'(0) + \alpha u(0) &= 0 \\
C + \alpha D &= 0 \\
C &= -\alpha D
\end{aligned}
$$

- Right boundary condition:

$$
\begin{aligned}
u'(1) + \alpha u(1) &= 0 \\
-1 - C + \alpha\left(-\frac{1}{2} - C + D\right) &= 0 \\
-1 + \alpha D + \alpha\left(-\frac{1}{2} + \alpha D + D\right) &= 0 \\
D(2\alpha + \alpha^2) &= \frac{1}{2}\alpha + 1 \\
\alpha D(2 + \alpha) &= \frac{\alpha + 2}{2} \\
D &= \frac{1}{2\alpha} \\
C &= -\frac{1}{2}
\end{aligned}
$$

- Solution:

$$
u(x) = -\frac{1}{2}x^2 + \frac{1}{2}x + \frac{1}{2\alpha}
$$

[10]:
```
u(x,alpha)=0.5*(-x*x +x + 1/alpha)
u_exact(N,alpha)=u.(collect(0:1/(N-1):1),alpha)
```

[10]: u_exact (generic function with 1 method)

1.4 Discrete problem from finite volume approximation

- gives a better idea how to handle boundary conditions…

$$\begin{pmatrix} \alpha+\frac{1}{h} & -\frac{1}{h} & & & & & \\ -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & & & \\ & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & & \\ & \ddots & \ddots & \ddots & \ddots & & \\ & & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & \\ & & & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & \\ & & & & -\frac{1}{h} & \frac{1}{h}+\alpha \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-2} \\ u_{N-1} \\ u_N \end{pmatrix} = \begin{pmatrix} \frac{h}{2} \\ h \\ h \\ \vdots \\ h \\ h \\ \frac{h}{2} \end{pmatrix}$$

1.5 Problem setup

[11]:
```
function setup(N,alpha)
    h=1.0/(N-1)
    a=[-1/h for i=1:N-1]
    b=[2/h for i=1:N]
    c=[-1/h for i=1:N-1]
    b[1]=alpha+1/h
    b[N]=alpha+1/h
    f=[h for i=1:N]
    f[1]=h/2
    f[N]=h/2
    return a,b,c,f
end
```

[11]: setup (generic function with 1 method)

Correctness check

[12]:
```
check(N,alpha,solver)=norm(solver(setup(N,alpha)...)-u_exact(N,alpha))
```

[12]: check (generic function with 1 method)

Setup tools

[13]:
```
using LinearAlgebra
using SparseArrays
```

1.6 Solvers

Progonka (c) Daniel Kind, Alon Cohn

Returns the solution u of the tridiagonal system defined by a,b,c,f where - a is the lower diagonal of size N-1, - b is the main diagonal of size N, - c is the upper diagonal of size N-1 - f is the right hand side vector of size N

[14]:
```
function progonka(a,b,c,f)
    N = size(f,1)
    u=Vector{eltype(a)}(undef,N)
    Alpha=Vector{eltype(a)}(undef,N)
    Beta=Vector{eltype(a)}(undef,N)
    Alpha[2] = -c[1]/b[1]
    Beta[2] = f[1]/b[1]
    for i in 2:N-1 #Forward Sweep
        Alpha[i+1]=-c[i]/(a[i-1]*Alpha[i]+b[i])
        Beta[i+1]=(f[i]-a[i-1]*Beta[i])/(a[i-1]*Alpha[i]+b[i])
    end
    u[N]=(f[N]-a[N-1]*Beta[N])/(a[N-1]*Alpha[N]+b[N])
    for i in N-1:-1:1 #Backward Sweep
        u[i]=Alpha[i+1]*u[i+1]+Beta[i+1]
    end
    return u
end
```

[14]:
```
progonka (generic function with 1 method)
```

Setup data

[15]:
```
alpha=1
N=1000
a,b,c,f=setup(N,alpha)
```

[15]:
```
([-999.0, -999.0, -999.0, -999.0, -999.0, -999.0, -999.0, -999.0, -999.0, -999.0
…  -999.0, -999.0, -999.0, -999.0, -999.0, -999.0, -999.0, -999.0, -999.0,
-999.0], [1000.0, 1998.0, 1998.0, 1998.0, 1998.0, 1998.0, 1998.0, 1998.0,
1998.0, 1998.0  …  1998.0, 1998.0, 1998.0, 1998.0, 1998.0, 1998.0, 1998.0,
1998.0, 1998.0, 1000.0], [-999.0, -999.0, -999.0, -999.0, -999.0, -999.0,
-999.0, -999.0, -999.0, -999.0  …  -999.0, -999.0, -999.0, -999.0, -999.0,
-999.0, -999.0, -999.0, -999.0, -999.0], [0.0005005005005005005,
0.001001001001001001, 0.001001001001001001, 0.001001001001001001,
0.001001001001001001, 0.001001001001001001, 0.001001001001001001,
0.001001001001001001, 0.001001001001001001, 0.001001001001001001  …
0.001001001001001001, 0.001001001001001001, 0.001001001001001001,
0.001001001001001001, 0.001001001001001001, 0.001001001001001001,
0.001001001001001001, 0.001001001001001001, 0.001001001001001001,
0.0005005005005005005])
```

Check correctness of solution

[16]:
```
@show check(N,alpha,progonka)
```

check(N, alpha, progonka) = 4.207319001047909e-12

[16]: 4.207319001047909e-12

Benchmark

[17]:
```
@btime progonka(a,b,c,f);
```

```
  9.119  s (3 allocations: 23.81 KiB)
```

Progonka in C

- Create file progonka.c

[18]:
```
open("progonka.c", "w") do io
    write(io, """
void progonka(int N,double* u,double* a,double* b,double* c,double* f,double*␣
 ↪Alpha,double* Beta)
{
  int i;
  /* Adjust indexing:
    This is C pointer arithmetic. Shifting the start addresses by 1
    allows to keep the indexing from 1.
 */
  u--;
  a--;
  b--;
  c--;
  f--;
  Alpha--;
  Beta--;
  Alpha[2] = -c[1]/b[1];
  Beta[2] = f[1]/b[1];
  for(i=2;i<=N-1;i++)
  {
    Alpha[i+1]=-c[i]/(a[i-1]*Alpha[i]+b[i]);
    Beta[i+1]=(f[i]-a[i-1]*Beta[i])/(a[i-1]*Alpha[i]+b[i]);
  }
  u[N]=(f[N]-a[N-1]*Beta[N])/(a[N-1]*Alpha[N]+b[N]);
  for(i=N-1;i>=1;-i--)
  {
    u[i]=Alpha[i+1]*u[i+1]+Beta[i+1];
  }
}
""")
end
```

[18]: 606

- Compile file progonka.c with highest optimization level

[19]:
```
run(`gcc -Ofast --shared  progonka.c -o progonka.so`)
```

[19]:
```
Process(`gcc -Ofast --shared progonka.c
-o progonka.so`, ProcessExited(0))
```

- Julia wrapper for C code

[20]:
```
function cprogonka(a,b,c,f)
    u=Vector{eltype(a)}(undef,N)
    Alpha=Vector{eltype(a)}(undef,N)
    Beta=Vector{eltype(a)}(undef,N)
    ccall( (:progonka,"progonka"),␣
 ↪Cvoid,(Cint,Ptr{Cdouble},Ptr{Cdouble},Ptr{Cdouble},Ptr{Cdouble},Ptr{Cdouble},Ptr{Cdouble},P
           N,u,a,b,c,f,Alpha,Beta)
    return u
end
```

[20]:
```
cprogonka (generic function with 1 method)
```

[21]:
```
@show check(N,alpha,cprogonka)
@btime cprogonka(a,b,c,f);
```

```
check(N, alpha, cprogonka) = 4.207319001047909e-12
  12.465  s (8 allocations: 23.89 KiB)
```

Julia won…

Progonka in Fortran

- Create file fprogonka.f
- This is fixed format fortran 77, watch the line offset

[22]:
```
open("fprogonka.f", "w") do io
    write(io, """
      subroutine fprogonka(N,u,a,b,c,f,Alpha,Beta)
      integer*4 i
      integer*4 N
      real*8 u(N),a(N-1),b(N),c(N-1),f(N),Alpha(N),Beta(N)

      Alpha(2) = -c(1)/b(1)
      Beta(2) = f(1)/b(1)
      do i=2,N-1
         Alpha(i+1)=-c(i)/(a(i-1)*Alpha(i)+b(i));
         Beta(i+1)=(f(i)-a(i-1)*Beta(i))/(a(i-1)*Alpha(i)+b(i));
      enddo
      u(N)=(f(N)-a(N-1)*Beta(N))/(a(N-1)*Alpha(N)+b(N));
      do i=N-1,1,-1

         u(i)=Alpha(i+1)*u(i+1)+Beta(i+1);
```

```
    enddo
    end
""")
end
```

[22]: 488

- Compile file fprogonka.f with highest optimization level

```
[23]: run(`gfortran -Ofast --shared  fprogonka.f -o fprogonka.so`)
```

[23]: Process(`gfortran -Ofast --shared fprogonka.f -o fprogonka.so`, ProcessExited(0))

- Julia wrapper for Fortran code
- note the _ at the end of the function name

```
[24]: function fprogonka(a,b,c,f)
          u=Vector{eltype(a)}(undef,N)
          Alpha=Vector{eltype(a)}(undef,N)
          Beta=Vector{eltype(a)}(undef,N)
          ccall( (:fprogonka_,"fprogonka"),␣
      ↪Cvoid,(Ref{Int64},Ptr{Cdouble},Ptr{Cdouble},Ptr{Cdouble},Ptr{Cdouble},Ptr{Cdouble},Ptr{Cdou
                   Ref{Int64}(N),u,a,b,c,f,Alpha,Beta)
          u
      end
```

[24]: fprogonka (generic function with 1 method)

```
[25]: @show check(N,alpha,fprogonka)
      @btime fprogonka(a,b,c,f);
```

```
check(N, alpha, fprogonka) = 4.207319001047909e-12
  11.222  s (7 allocations: 23.88 KiB)
```

Julia won…

Julia tridiagonal solver

```
[26]: function julia_tri(a,b,c,f)
          u=zeros(N)
          A=Tridiagonal(a,b,c)
          Alu = lu(A,Val(false))
          Alu\f
      end
```

[26]: julia_tri (generic function with 1 method)

```
[27]: @show check(N,alpha,julia_tri)
      @btime julia_tri(a,b,c,f);
```

```
check(N, alpha, julia_tri) = 4.268791043778398e-12
  18.348  s (11 allocations: 51.86 KiB)
```

... only half as fast as progonka

Julia dense matrix

```
[28]: function julia_dense(a,b,c,f)
          u=zeros(N)
          A=Matrix(Tridiagonal(a,b,c))
          Alu = lu(A)
          Alu\f
      end
```

```
[28]: julia_dense (generic function with 1 method)
```

```
[29]: @show check(N,alpha,julia_dense)
      @btime julia_dense(a,b,c,f);
```

```
check(N, alpha, julia_dense) = 1.6429367859842642e-11
  14.029 ms (9 allocations: 15.28 MiB)
```

This is what we expected: handling of the dense matrix is expensive

Julia dense matrix

```
[30]: function julia_inv(a,b,c,f)
          u=zeros(N)
          Ainv=inv(Tridiagonal(a,b,c))
          Ainv*f
      end
```

```
[30]: julia_inv (generic function with 1 method)
```

```
[31]: @show check(N,alpha,julia_inv)
      @btime julia_inv(a,b,c,f);
```

```
check(N, alpha, julia_inv) = 4.501529993246134e-12
  10.170 ms (13 allocations: 7.68 MiB)
```

- Almost by a factor of 1000 slower than tridiagonal solve
- This is what we expected: handling of the dense matrix is expensive
- The inverse matrix has no non-zero entries. This is always the case with 2nd order PDEs

Julia sparse matrix

[32]:
```
function julia_sparse(a,b,c,f)
    u=zeros(N)
    A=SparseMatrixCSC(Tridiagonal(a,b,c))
    Alu = lu(A)
    Alu\f
end
```

[32]: julia_sparse (generic function with 1 method)

[33]:
```
@show check(N,alpha,julia_sparse)
@btime julia_sparse(a,b,c,f);
```

```
check(N, alpha, julia_sparse) = 1.1718994017082038e-11
  2.040 ms (96 allocations: 1.19 MiB)
```

- Almost by a factor of 100 slower than tridiagonal solve
- This is *not* what we expected ...
- The benchmark included the sparse matrix setup time!

Julia sparse matrix solver with assembled sparse matrix

[34]:
```
function julia_sparse(A,f)
    u=zeros(N)
    Alu = lu(A)
    Alu\f
end
```

[34]: julia_sparse (generic function with 2 methods)

[35]:
```
A=SparseMatrixCSC(Tridiagonal(a,b,c))
@btime julia_sparse(A,f);
```

```
  468.926  s (69 allocations: 1.06 MiB)
```

- Still slower than tridiagonal solve
- Direct use of \ is faster than storing LU in between.

1.6.1 What about the sparse matrix build up?

- Simply build SparseMatrixCSC via loop

[36]:
```
function sparse_csc(M::Tridiagonal)
    N=size(M,1)
    A=spzeros(N,N)
    A[1,1]=M[1,1]
    A[2,1]=M[2,1]
    for i=2:N-1
        A[i-1,i]=M[i-1,i]
        A[i,i]=M[i,i]
        A[i+1,i]=M[i+1,i]
```

```
    end
    A[N-1,N]=M[N-1,N]
    A[N,N]=M[N,N]
    A
end
```

[36]: sparse_csc (generic function with 1 method)

- Use ExtendableSparse package which has an intermediate storage of matrix data
- Same algorithm as in pdelib of WIAS and the numcxx library from previous courses.
- I learned it long time ago and don't recall the source .... Any hints appreciated.

[37]:
```
using ExtendableSparse
function sparse_ext(M::Tridiagonal)
    N=size(M,1)
    A=ExtendableSparseMatrix{Float64,Int64}(N,N)
    A[1,1]=M[1,1]
    A[2,1]=M[2,1]
    for i=2:N-1
        A[i-1,i]=M[i-1,i]
        A[i,i]=M[i,i]
        A[i+1,i]=M[i+1,i]
    end
    A[N-1,N]=M[N-1,N]
    A[N,N]=M[N,N]
    flush!(A)
    A.cscmatrix
end
```

[37]: sparse_ext (generic function with 1 method)

Benchmark them

[38]:
```
@btime A=SparseMatrixCSC(Tridiagonal(a,b,c))
@btime A=sparse_csc(Tridiagonal(a,b,c))
@btime A=sparse_ext(Tridiagonal(a,b,c));
```

```
  1.569 ms (27 allocations: 137.13 KiB)
  174.675  s (27 allocations: 137.13 KiB)
  78.245  s (30 allocations: 252.31 KiB)
```

- It makes sense to have an intermediate structure

1.6.2 The case $\alpha \to \infty$

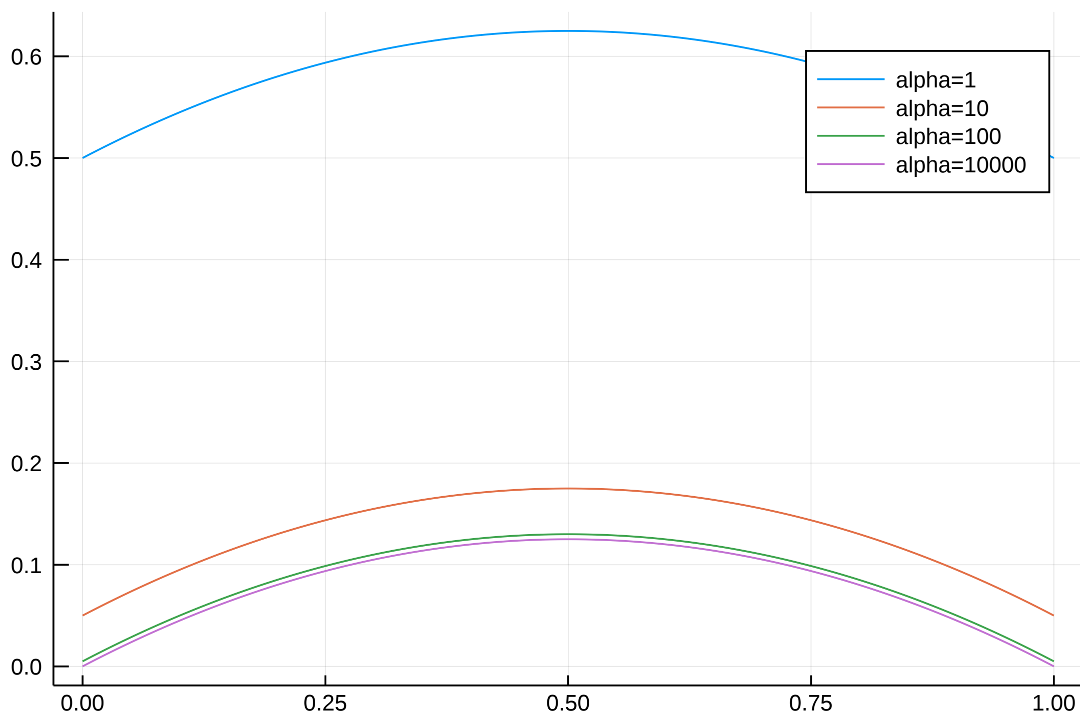
$$u(x) = -\frac{1}{2}x^2 + \frac{1}{2}x + \frac{1}{2\alpha}$$

- The problem with $\alpha > 0$ is called Robin boundary value problem or boundary value problem of the third kind
- We have $u(0) = u(1) = \frac{1}{2\alpha} \to 0 \quad \alpha \to \infty$
- i.e. for large α we approximate (homogeneous) Dirichlet boundary conditions.
- $\Rightarrow$ numerical trick for easy implementation of Dirichlet boundary conditions called penalty method: a large value of α penalizes the deviation from zero.

[39]:
```
using Plots

p=plot()
X=collect(0:1/(N-1):1)
p=plot()
plot!(p,X,u_exact(N,1),label="alpha=1")
plot!(p,X,u_exact(N,10),label="alpha=10")
plot!(p,X,u_exact(N,100),label="alpha=100")
plot!(p,X,u_exact(N,10000),label="alpha=10000")
```

[39]:



This notebook was generated using Literate.jl.