HW2 Precision and Stability

Fei Ding

September 15, 2009

Problem1

The Code:

```
program precision
double precision:: eps, one  ! for single precision, change 'double precision' to 'real'
integer:: i, N
eps=1.
N=100
OPEN(unit=1,file='data1.txt')
do i=1, N
   eps=eps/2.
   one=1.+eps
   write(1,'(1X,I5,F28.23,ES30.6)') i,one,eps
end do
close(1)
end program precision
```

The double precision is 2.220446×10^{-16}, which is the last ϵ value before $(1+\epsilon)$ got to constant 1.00000000000000000000000;

The single precision is 1.192093×10^{-7}, which is the last ϵ value before $(1+\epsilon)$ got to constant 1.00000000000000000000000.

Problem2

The Code:

```
program limits
  IMPLICIT NONE
  real:: realUnder, realOver !parameters for single precision limit
  double precision:: doubleUnder,doubleOver !parameters for double precision limit
  integer:: N,loopNum
  OPEN(unit=2,file='data2.txt')
  OPEN(unit=3,file='data3.txt')
```

```
realUnder=1.0
realOver=1.0
doubleUnder=1.0
doubleOver=1.0
N=2000
do loopNum=1, N
   realUnder=realUnder/2.0
   realOver=realOver*2.0
   doubleUnder=doubleUnder/2.
   doubleOver=doubleOver*2.
   write (2,*), loopNum, realUnder, realOver
   write (3,*), loopNum, doubleUnder, doubleOver
end do
close(2)
close(3)
end program limits
```

The maximum single precision real number my computer supports is $1.70141183 \times 10^{38}$, which is the last realOver value before it got to +Infinity; the minimum single precision real number of my computer supports is $1.40129846 \times 10^{-45}$, which is the last realUnder value before it got to constant 0.0000000.

The maximum double precision real number my computer supports is $8.98846567431157954 \times 10^{307}$, which is the last doubleOver value before it got to +Infinity; the minimum double precision real number of my computer supports is $4.94065645841246544 \times 10^{-324}$, which is the last realUnder value before it got to constant 0.0000000000000000.

Problem3

The code:

```
program main
IMPLICIT NONE
integer:: n,dFact
integer:: i,M
double precision::x,dx,jRecurs,jAsympt,sphBessel
double precision::error,relativeError
open(unit=1,file='data4_1.txt')
open(unit=2,file='data4_2.txt')
print *,'Enter n for Spherical Bessel functions, jn(x):'
read *, n
x=0.0
M=500
dx=(1.0-0.0)/M
do i=1,M
```

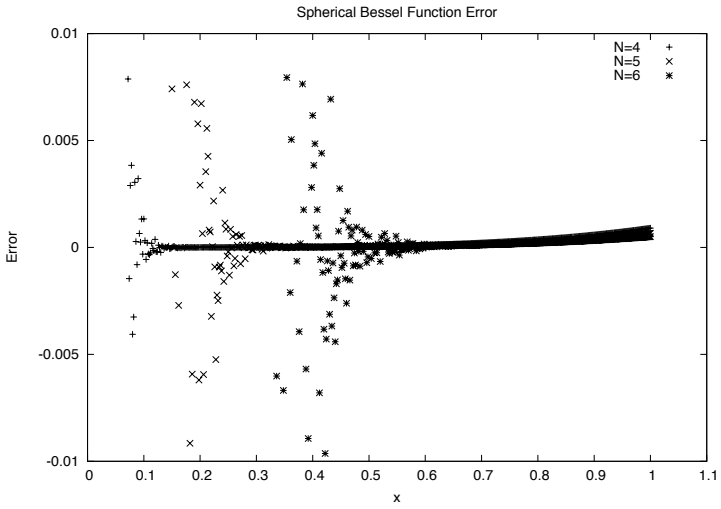
```
   x=x+dx
   jRecurs=sphBessel(n,x)
   jAsympt=x**n/dFact(2*n+1)*(1-x**2/(2*(2*n+3)))
   error=jRecurs-jAsympt
   relativeError=error/jAsympt
   write(1,*), x,jRecurs,jAsympt
   write(2,*), x,relativeError
end do
close(1)
close(2)
end program main

!Subroutine to calculate double factorial
recursive integer function dFact(n) result(Fvalue)
  IMPLICIT NONE
  integer:: n, Fvalue
  if (n==0) then
    Fvalue=1
  else if (n==1) then
    Fvalue=1
  else
    Fvalue=dFact(n-2)*n
  end if
  return
end function dFact

!Subroutine to calculate spherical Bessel function using recursing realation
recursive double precision function sphBessel(n,x) result(Bvalue)
  IMPLICIT NONE
  integer:: n
  double precision:: x,Bvalue
  if (n.eq.0) then
    Bvalue=dsin(x)/x
  else if (n.eq.1) then
    Bvalue=dsin(x)/(x**2)-dcos(x)/x
  else
    Bvalue=(2*(n-1)+1)/x*sphBessel(n-1,x)-sphBessel(n-2,x)
  end if
  return
end function sphBessel
```

The error for $n = 4, n = 5$ and $n = 6$ are plotted in the same figure as following,

From the figure we can see,

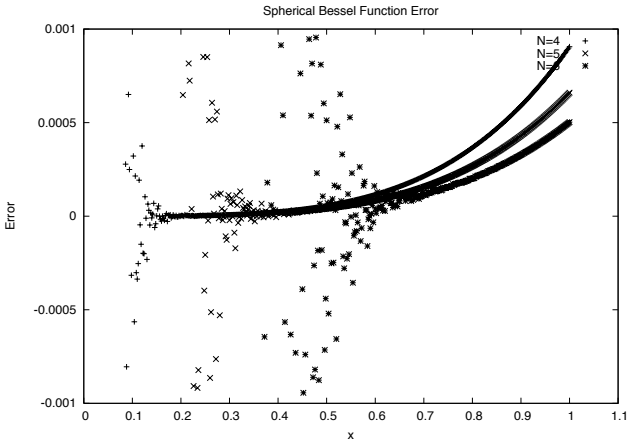


The error for different n are pretty close in the 10^{-2}scale.

For all three n, the errors for small x are large and abnormal. That is because x=0 is a pole for $j_n(0)$and $j_n(1)$. Also we can see that larger n requires longer time for the error to behave normally, e.g. $n = 4$, the normal part starts around $x = 0.13$; $n = 5$, around $x = 0.3$; $n = 6$, around $x = 0.6$. I think this is because of larger n uses more recursion steps, which would screw up the data more than smaller n.

For the normal part, as x becomes larger, the errors for all n increase. This makes sense because usually when polynomial expansion is used, the value for relative large x turn out to be more off.

Change the Y-aixs scale to 10^{-3}, get the figure as following,



The figure shows clearly that for larger n, the error is smaller. This is kind of

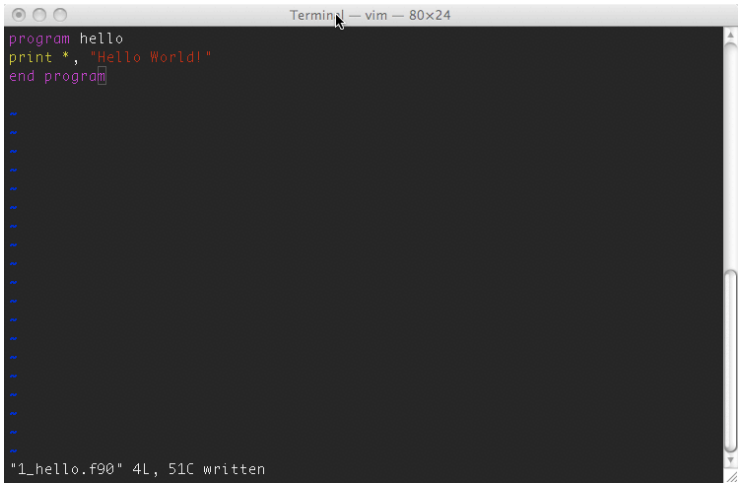
wired to me. Because I think as the recursion step gets more, the errors should accumulate, leading the final error to be larger. But anyway, this is the result I got.

Problem 4 I checked the wikipedia and discussed with my batch mates. I think by the Lanczos algorithm we should get the highest excited state first instead of getting ground state. So the algorithm should be stable for highest ground state.

However, for any other excited state with lower eigenvalue and the ground state, the round-off error which came from making the initial vector orthogonal to the highest excited state would introduce slight components of the more significant eigenvectors back into the computation. That is when we apply repeated applications of H, the error would bring the computation back to the first step, so we would still get the highest excited state instead of what we expected.

Make up for Problem 1 at HW1 I had a problem with the fortran compiler in HW1, so I would show the screen cap part for HW1 here.

The code



The compiler command, the application invocation and the result,