Homework 1, Problem 1 (solutions)

Torsten Clay

Introduction

Part 1 in text:

a. sign=1, e = 00001110, f= 1010 0000 0000 0000 0000 0000

b. $e=(2^1+2^2+2^3)=14$, $p=e-127=-113$

c. $f=2^0+2^{-1}+2^{-3}=1.625$

d. $1.625 * 2^{-113}=1.565*10^{-24}$

We want to determine the minimum and maximum numbers that can be represented in each type: single precision and double precision.

To determine the maximum number that can be represented (floating point numbers), either multiply or divide by a number until under or overflow is reached. This will give us approximately the largest/smallest floating point numbers that can be used.

Instead of executing this process a fixed number of iterations, we can check when it is done as follows: the minimum (underflow) quantity should test greater than zero, and the maximum (overflow) quantity should not give zero when its reciprocal is taken.

Code

overunder.f90

Results

Here is the output on a Linux system using the gfortran compiler:

```
single precision
underflow =   1.40129846E-45
overflow=  3.40279548E+38

double precision
underflow =   4.94065645841246544E-324
overflow=  1.79768990553502587E+308
```

```
Integer imin= -2147483648
Integer imax=  2147483647

Single precision: Fortran 90 tiny(x)=  1.17549435E-38
                          huge(x)=  3.40282347E+38
Double precision: Fortran 90 tiny(x)=  2.22507385850720138E-308
                          huge(x)=  1.79769313486231571E+308
Integer: Fortran 90 huge(x)  =  2147483647
Integer: Fortran 90 huge(x)+1= -2147483648
```

Discussion

When finding the underflow, it didn't matter whether the number was divided by 2 or by a smaller number, because for the smallest numbers (denormals) only the mantissa is significant, and it is stored in powers of two. Therefore one should be able to reach the smallest floating point number by successively dividing by 2. However, for the overflow, this did make a difference; a slightly larger overflow could be found by changing the 1.001 in the code to 1.0001, etc.

With integers, if one subtracts -1 from the smallest integer, the result typically "wraps around" to the largest integer, and vice-versa. However, it is not good practice to count on this behavior in a program.

There are some built-in functions that give these values in Fortran 90. tiny(x) and huge(x) give the minimum and maximum valus of the variable x. Other similar functions are digits(x), epsilon(x), maxexponent(x), minexponent(x), precision(x), radix(x), range(x). Note that tiny(x) seems to give the smallest normalized real, so the number you determined is actually smaller since it is a "denormal".

huge(x) works with integer types but tiny(x) does not. We know the smallest integer anyway!

```
! R.T. Clay PH4433/6433 8/2015
!
! program to find underflow and overflow limits
!
Program underover
  implicit none

  real (kind=4) :: fover,funder
  real (kind=4) :: fmin,fmax,ftemp
  real (kind=8) :: dover,dunder
  real (kind=8) ::  dmin,dmax,dtemp
  integer :: i,imin,imax

  ! single precision

  fmin=1.0
  funder=1.0
  do while (funder>0.0)
     fmin=funder        ! fmin saves last value before underflow to 0
     funder=funder*0.5
  enddo

  fmax=0.0
  fover=1.0
  do while (1.0/fover>0.0)
     fmax=fover         ! saves last value before overflow
     fover=fover*1.00001
  enddo

  write(*,*) "single precision"
  write(*,*) "underflow = ",fmin
  write(*,*) "overflow=",fmax


  ! double precision

  dmax=0.0_8
  dmin=1.0_8
  dunder=1.0_8
  dover=1.0_8

  do while (dunder>0.0_8)
     dmin=dunder
     dunder=dunder*0.5_8
  enddo

  do while (1.0/dover>0.0_8)
     dmax=dover
     dover=dover*1.00001_8
  enddo

  write(*,*)
  write(*,*) "double precision"
  write(*,*) "underflow = ",dmin
  write(*,*) "overflow=",dmax

  ! integers
  ! this may not work with all compilers! With gfortran it does
  ! not work correctly with all levels of optimization; with -O2
  ! the program never terminates
  i=0
  do
     i=i-1
     ! check to see if result is no longer negative
     if (i>0) then
        imax=i
        imin=i+1
        exit
     endif
  enddo
  write(*,*)
  write(*,*) 'Integer imin=',imin
  write(*,*) 'Integer imax=',imax
  ! using built-in functions
  write(*,*)
  write(*,*) 'Single precision: Fortran 90 tiny(x)=',tiny(fover)
  write(*,*) '                  huge(x)=',huge(fover)
  write(*,*) 'Double precision: Fortran 90 tiny(x)=',tiny(dover)
  write(*,*) '                  huge(x)=',huge(dover)
  write(*,*) 'Integer: Fortran 90 huge(x) =',huge(i)
  imin=huge(i)
  imin=imin+1
  write(*,*) 'Integer: Fortran 90 huge(x)+1=',imin
end program underover
```

Homework 1, Problem 2 (solutions)

Torsten Clay

Introduction

We want to determine the minimum numbers that can be added to 1.0 and give a result different from 1.0. This is called the machine precision, e_m.

Code

machprec.f90

Note in checking when $1+e_m=1$, you should avoid comparing two reals with '==' or '/='. Instead use '>' or '<'. This is a very convenient place to use a while loop rather than a do loop.

Results

Here are the results I got (64-bit Linux system)

```
Single precision   1.19209290E-07
Double precision   2.22044604925031308E-016

Fortran 90 function epsilon(x)
single=  1.19209290E-07
double=  2.22044604925031308E-016
```

Discussion

The definition of what "machine precision" is varies slightly. Your textbook says it is the largest number that can be added to 1 and not change the value of 1; other sources say it is the largest number that when added to 1 gives a value different from 1. Here I am taking the second definition which I think is somewhat more common.

Another way to think of machine precision is that it is the largest (fractional) error in representing a number using floating point.

These are typical values for machine precision in single and double precision. Note that always dividing the small number by 2 works here because floating point values are stored in base 2.

It is bad practice to rely on comparing two floating point values. If you have to test for equality of two fp values, you should allow them to be different by some small amount, of order machine precision or in practice several times machine

precision.

There is a built-in function to return the machine precision, epsilon(x). It returns the same value.

```
! R.T. Clay PH4433/6433 8/2015
!
! determine floating point machine precision
!
Program machprec
  implicit none

  real (kind=4) :: feps,fmachep
  real (kind=8) :: deps,dmachep

  ! single precision

  feps=1.0
  do while ((1.0+feps)>1.0)
     fmachep=feps
     feps=feps*0.5
  enddo
  write(*,*)'Single precision ',fmachep

  ! double precision

  deps=1.0_8
  do while ((1.0_8+deps)>1.0_8)
     dmachep=deps
     deps=deps*0.5_8
  enddo

  write(*,*)'Double precision ',dmachep

  write(*,*)
  write(*,*) 'Fortran 90 function epsilon(x)'
  write(*,*) '   single=',epsilon(fmachep)
  write(*,*) '   double=',epsilon(dmachep)

end program machprec
```

Homework 1, Problem 3 (solutions)

Torsten Clay

Introduction

We want to sum the series $1/(2n+1)^2$ from n=0 to infinity. The exact sum is $pi^2/8 = 1.23370055$. The exercise will compare the result of summing the series in the increasing direction and in the decreasing direction.

Code

The code uses single precision to make the roundoff error more easily visible.

series.f90

Results

The four columns are i) the number of terms in the summation, ii) the result of summing up, iii) the relative error in the up sum, iv) the result of summing down, and v) the relative error in the down sum. Below I removed a lot of output from the middle of the run.

```
exact =    1.23370063
   0  1.000000000000 -0.18943058  1.000000000000 -0.18943058
 100  1.231225371361 -0.00200637  1.231225371361 -0.00200637
 200  1.232456564903 -0.00100840  1.232456803322 -0.00100821
 300  1.232869863510 -0.00067340  1.232869982719 -0.00067330
 400  1.233076930046 -0.00050555  1.233077049255 -0.00050546
 500  1.233201622963 -0.00040448  1.233201503754 -0.00040458
 600  1.233284473419 -0.00033733  1.233284592628 -0.00033723
 700  1.233343839645 -0.00028921  1.233343958855 -0.00028911
 800  1.233388304710 -0.00025316  1.233388423920 -0.00025307
 900  1.233424067497 -0.00022418  1.233423113823 -0.00022495
1000  1.233449697495 -0.00020340  1.233450770378 -0.00020253
1100  1.233473539352 -0.00018408  1.233473539352 -0.00018408
1200  1.233495116234 -0.00016659  1.233492374420 -0.00016881
1300  1.233507037163 -0.00015692  1.233508348465 -0.00015586
1400  1.233518958092 -0.00014726  1.233522176743 -0.00014465
1500  1.233530879021 -0.00013760  1.233533978462 -0.00013509
1600  1.233542799950 -0.00012793  1.233544349670 -0.00012668
1700  1.233554720879 -0.00011827  1.233553647995 -0.00011914
1800  1.233566641808 -0.00010861  1.233561754227 -0.00011257
1900  1.233578562737 -0.00009895  1.233569025993 -0.00010668
2000  1.233590483665 -0.00008928  1.233575582504 -0.00010136
2100  1.233596086502 -0.00008474  1.233581542969 -0.00009653
2200  1.233596086502 -0.00008474  1.233587026596 -0.00009209
2300  1.233596086502 -0.00008474  1.233591914177 -0.00008812
2400  1.233596086502 -0.00008474  1.233596444130 -0.00008445
2500  1.233596086502 -0.00008474  1.233600616455 -0.00008107
```

```
2600  1.233596086502 -0.00008474  1.233604431152 -0.00007798
...
1896200  1.233596086502 -0.00008474  1.233700394630 -0.00000019
1896300  1.233596086502 -0.00008474  1.233700394630 -0.00000019
1896400  1.233596086502 -0.00008474  1.233700394630 -0.00000019
1896500  1.233596086502 -0.00008474  1.233700394630 -0.00000019
```

Discussion

Notice that at around n=2100, the error in the upwards sum reaches a constant for all higher n. At this point, the terms being added are small enough that there are not enough mantissa bits to accurately add the sum and the next term. The upward sum then gets "stuck" at this value for all higher n. Notice that the "best" value from the upward sum is incorrect by much more than machine precision.

The downwards sum continues to get closer to the exact result for much larger n than shown here, eventually getting about 6 digits correct; this is about as good as can be achieved with single precision. Summing a fixed number of terms is however very inefficient; best might be to stop the sum once it has converged (stopped changing) to some desired tolerance.

```
! RT Clay PH4433/6433 08/2015
program series
  implicit none

  real(kind=4),parameter :: exact=(atan(-1.0)*4.0)**2/8
  integer :: n, nmax
  real(kind=4) :: sumup,sumdn

  print *,'exact=',exact
  do nmax=0,10000000,100
     sumup=0.0
     do n=0,nmax
        sumup=sumup+1.0/(2*n+1.0)**2
     enddo
     sumdn=0.0
     do n=nmax,0,-1
        sumdn=sumdn+1.0/(2*n+1.0)**2
     enddo

     print "(I8,F16.12,F12.8,F16.12,F12.8)",nmax,sumup,(sumup-exact)/exact, &
          sumdn,(sumdn-exact)/exact
  enddo

end program series
```

Homework 1, Problem 4 (solutions)

Introduction

The Taylor series for exp(-x) has terms $x^n/n!$. Evaluating the function by calculating x^n and n! directly does not work well because the factorial function will quickly overflow. A better way is to sum a series of terms s_n, where $s_n=-s_{n-1}*(x/n)$. The point of this problem is to show that this approach also has numerical problems: when x is negative, terms alternate in sign and subtractive roundoff is significant.

Code

1. Part A
2. Part B

Results

From the first program.

```
-x                      "exact" exp(-x)           series                      n
0.0000000000000000       1.0000000000000000        1.0000000000000000           1
10.000000000000000      4.5399929762484854E-005  4.5399929623571721E-005      51
20.000000000000000      2.0611536224385579E-009  5.6218848072715590E-009      81
30.000000000000000      9.3576229688401748E-014 -3.0668123563010421E-005     109
40.000000000000000      4.2483542552915888E-018  -3.1657318940631241         137
50.000000000000000      1.9287498479639178E-022   11072.933382891970         165
60.000000000000000      8.7565107626965200E-027  -335168107.25821894         192
70.000000000000000      3.9754497359086468E-031  -32979604674602.469         220
80.000000000000000      1.8048513878454153E-035   91805682166301088.         247
90.000000000000000      8.1940126239905147E-040 -5.0516253513522463E+021     274
100.00000000000000      3.7200759760208361E-044 -2.9137556468915326E+025     302
```

From the second program.

```
-x                      "exact" exp(-x)           series                      n
0.0000000000000000       1.0000000000000000        1.0000000000000000           1
10.000000000000000      4.5399929762484854E-005  4.5399929762484861E-005      51
20.000000000000000      2.0611536224385579E-009  2.0611536224385583E-009      81
30.000000000000000      9.3576229688401748E-014  9.3576229688401710E-014     109
40.000000000000000      4.2483542552915888E-018  4.2483542552915896E-018     137
50.000000000000000      1.9287498479639178E-022  1.9287498479639169E-022     165
60.000000000000000      8.7565107626965200E-027  8.7565107626965171E-027     192
70.000000000000000      3.9754497359086468E-031  3.9754497359086495E-031     220
80.000000000000000      1.8048513878454153E-035  1.8048513878454164E-035     247
90.000000000000000      8.1940126239905147E-040  8.1940126239905163E-040     274
100.00000000000000      3.7200759760208361E-044  3.7200759760208386E-044     302
```

Discussion

Another way to see what is happening is for a fixed x to print out each term in the

sum and the total sum:

For x=30, exp(-30)=9.35762296884017482E-014:

i	s_i	sum
1	-30.000000000000000	1.0000000000000000
2	450.00000000000000	-29.000000000000000
3	-4500.0000000000000	421.00000000000000
4	33750.000000000000	-4079.0000000000000
5	-202500.00000000000	29671.000000000000
6	1012500.0000000000	-172829.00000000000
7	-4339285.7142857146	839671.00000000000
8	16272321.428571429	-3499614.7142857146
9	-54241071.428571433	12772706.714285715
10	162723214.28571430	-41468364.714285716
11	-443790584.41558450	121254849.57142858
12	1109476461.0389612	-322535734.84415591
13	-2560330294.7052951	786940726.19480526
14	5486422060.0827761	-1773389568.5104899
15	-10972844120.165552	3713032491.5722861
16	20574082725.310410	-7259811628.5932655
17	-36307204809.371307	13314271096.717144
18	60512008015.618843	-22992933712.654163
19	-95545275814.135010	37519074302.964676
20	143317913721.20251	-58026201511.170334
21	-204739876744.57501	85291712210.032181
22	279190741015.32953	-119448164534.54283
23	-364161836106.95154	159742576480.78668
24	455202295133.68945	-204419259626.16486
25	-546242754160.42737	250783035507.52460
26	630280100954.33923	-295459718652.90277
27	-700311223282.59912	334820382301.43646
28	750333453517.07043	-365490840981.16266
29	-776207020879.72803	384842612535.90778
30	776207020879.72803	-391364408343.82025
31	-751168084722.31738	384842612535.90778
32	704220079427.17261	-366325472186.40961
33	-640200072206.52063	337894607240.76300
34	564882416652.81226	-302305464965.75763
35	-484184928559.55334	262576951687.05463
36	403487440466.29443	-221607976872.49872
37	-327151978756.45490	181879463593.79572
38	258277877965.62228	-145272515162.65918
39	-198675290742.78638	113005362802.96310
40	149006468057.08978	-85669927939.823273
41	-109029122968.60228	63336540117.266510
42	77877944977.573059	-45692582851.335770
43	-54333449984.353302	32185362126.237289
44	37045534080.240891	-22148087858.116013
45	-24697022720.160595	14897446222.124878
46	16106753947.930822	-9799576498.0357170
47	-10280906775.274994	6307177449.8951054
48	6425566734.5468712	-3973729325.3798885
49	-3934020449.7225742	2451837409.1669827
50	2360412269.8335447	-1482183040.5555916
51	-1388477805.7844381	878229229.27795315
52	801044887.95256042	-510248576.50648499
53	-453421634.69012856	290796311.44607544
54	251900908.16118255	-162625323.24405313
55	-137400495.36064503	89275584.917129427
56	73607408.228916973	-48124910.443515599
57	-38740741.173114203	25482497.785401374
58	20038314.399886657	-13258243.387712829
59	-10188973.423671180	6780071.0121738277

```
 60   5094486.7118355902       -3408902.4114973526
 61  -2505485.2681158641        1685584.3003382375
 62   1212331.5813463859       -819900.96777762659
 63  -577300.75302208855        392430.61356875929
 64   270609.72797910403       -184870.13945332926
 65  -124896.79752881725        85739.588525774772
 66   56771.271604007838       -39157.209003042473
 67  -25419.972360003510        17614.062600965364
 68   11214.693688236843       -7805.9097590381461
 69  -4875.9537774942792        3408.7839291986966
 70   2089.6944760689771       -1467.1698482955826
 71  -882.96949693055376        622.52462777339451
 72   367.90395705439744       -260.44486915715925
 73  -151.19340700865649        107.45908789723819
 74   61.294624462968855       -43.734319111418301
 75  -24.517849785187543        17.560305351550554
 76   9.6780985994161366       -6.9575444336369898
 77  -3.7706877660062870        2.7205541657791468
 78   1.4502645253870334       -1.0501336002271402
 79 -0.55073336407102536       0.40013092515989324
 80  0.20652501152663452      -0.15060243891113212
 81 -7.64907450098646358E-002  5.59225726155024028E-002
 82  2.79844189060480392E-002 -2.05681723943622330E-002
 83 -1.01148502070053155E-002  7.41624651168580618E-003
 84  3.61244650250189866E-003 -2.69860369531950937E-003
 85 -1.27498111853008192E-003  9.13842807182389284E-004
 86  4.44760855301191353E-004 -3.61138311347692633E-004
 87 -1.53365812172824605E-004  8.36225439534987194E-005
 88  5.22837996043720276E-005 -6.97432682193258860E-005
 89 -1.76237526756310202E-005 -1.74594686149538584E-005
 90  5.87458422521034062E-006 -3.50832212905848785E-005
 91 -1.93667611820121098E-006 -2.92086370653745362E-005
 92  6.31524821152568790E-007 -3.11453131835757468E-005
 93 -2.03717684242764133E-007 -3.05137883624231784E-005
 94  6.50162822051374784E-008 -3.07175060466659402E-005
 95 -2.05314575384644698E-008 -3.06524897644608026E-005
 96  6.41608048077014620E-009 -3.06730212219992650E-005
 97 -1.98435478786705535E-009 -3.06666051415184929E-005
 98  6.07455547306241479E-010 -3.06685894963063614E-005
 99 -1.84077438577648921E-010 -3.06679820407590518E-005
100  5.52232315732946699E-011 -3.06681661181976262E-005
101 -1.64029400712756431E-011 -3.06681108949660532E-005
102  4.82439413861048328E-012 -3.06681272979061217E-005
103 -1.40516334134285934E-012 -3.06681224735119823E-005
104  4.05335579233517128E-013 -3.06681238786753215E-005
105 -1.15810165495290601E-013 -3.06681234733397392E-005
106  3.27764619326294167E-014 -3.06681235891499056E-005
107 -9.18966222410170665E-015 -3.06681235563734439E-005
108  2.55268395113936283E-015 -3.06681235655631077E-005
```

This shows where the problem comes from: in summing the series, even though there are only about 100 terms, there is a range of numbers much larger than the machine precision. Because the signs alternate, cancellation of the larger magnitude terms must be done nearly perfectly.

On the other hand, for positive x, no cancellation happens in the sum and it can be computed accurately.

```
! R.T.Clay 8/2015
!
! calculate exp(-x) using sum and recursion relation
!
program expo
  implicit none

  integer :: i,j,n,istep
  real(kind=8) :: x,xstep,sum,fac,term
  real(kind=8),parameter :: eps=1.0e-15_8

  ! try -x=0,10,20,...,100
  xstep=10.0_8
  x=0.0_8
  do istep=0,10

     term = 1.0_8
     sum = term
     n = 0
     do
        n=n+1
        term = -term*x/n

        ! terminate when converged to within eps
        if (abs(term)<eps) exit
        sum=sum+term
     enddo
     write(*,*) x,exp(-x),sum,n
     x=x+xstep
  enddo

end program expo
```

```
! R.T.Clay 8/2015
!
! calculate exp(-x) using sum and recursion relation
!
program expo
  implicit none

  integer :: i,j,n,istep
  real(kind=8) :: x,xstep,sum,fac,term
  real(kind=8),parameter :: eps=1.0e-15_8

  ! try -x=0,10,20,...,100
  xstep=10.0_8
  x=0.0_8
  do istep=0,10

     term = 1.0_8
     sum = term
     n = 0
     do
        n=n+1
        term = term*x/n

        ! terminate when converged to within eps
        if (abs(term)<eps) exit
        sum=sum+term
     enddo
     write(*,*) x,exp(-x),1/sum,n
     x=x+xstep
  enddo

end program expo
```