

**PHYSICS 115/242**  
**Homework 1, Solutions**

1. SOURCE CODE

```
print *, "Hello World"  
end
```

-----

OUTPUT

Hello World

2. (a) SOURCE CODE

```
i = 0  
ipower = 0  
  
do while (i >= 0)  
    i = ishft(i, 1) + 1      !i = 2 * i + 1  
    ipower = ipower + 1  
    print '(i12,4x,A,i2,A)', i, "2^",ipower," - 1"  
enddo
```

END

-----

OUTPUT

```
          1      2^ 1 - 1  
          3      2^ 2 - 1  
          .....  
1073741823    2^30 - 1  
2147483647    2^31 - 1  
          -1      2^32 - 1
```

This shows that the largest positive integer is  $2^{31} - 1$  and so  $32 - \text{bit}$  integers are used.

(b) First we do single precision.

SOURCE CODE

```
x = 1
```

```
do i = 1, 40  
    x = 10 * x  
    print *, i, x  
enddo
```

END

---

OUTPUT

```

1  10.00000
2  100.0000
.....
36 1.000000E+36
37 1.000000E+37
38 1.000000E+38
39 ++++++
```

This shows that the largest floating point single precision number on my compiler is about  $10^{38}$ .

For double precision, we simply declare  $x$  to be double precision and the output is then

```

1  10.00000
2  100.0000
.....
307 1.0000000000000000D+307
308 1.0000000000000000D+308
309 ++++++
```

so the largest double precision number is about  $10^{308}$ .

- (c) The program below starts with  $\epsilon = 1$  and at each iteration it divides  $\epsilon$  by 10. It then calculates  $1 - (1 - \epsilon)$ , which should, of course equal exactly  $\epsilon$ . This is the single precision version:

```

eps = 1

do
  eps = eps / 10
  y = 1 - eps
  print *, eps, 1 - y
  if (1 - y == 0) exit
enddo
```

END

The output below shows that the  $\epsilon < 10^{-7}$ , the computer cannot distinguish between  $1 - \epsilon$  and 1. Hence there are about  $7$  digits of precision in single precision.

```

0.1000000      0.1000000
1.000000E-02   9.999990E-03
.....
1.000000E-07   1.192093E-07
1.000000E-08   0.000000E+00
```

For double precision, I simply declare  $y$  and  $\epsilon$  to be double precision and the output is then

```

0.100000000000000000      1.0000000000000000D-001
1.0000000000000000D-002  1.0000000000000000D-002
.....
1.0000000000000000D-016  1.11022302462516D-016
1.0000000000000000D-017  0.0000000000000000D+000

```

which shows that there are about 16 bits of precision using double precision.

### 3. SOURCE CODE

```

IMPLICIT NONE
REAL*8 sumupdp, sumdowndp, nupdp, ndowndp
REAL    nup, ndown, sumup, sumdown
INTEGER n, i, idec

n = 1
do idec = 1, 7

    n = 10 * n
    sumup = 0; sumdown = 0; sumupdp = 0; sumdowndp = 0
    do i = 1, n

        nup = i
        nupdp = i
        ndown = n + 1 - i
        ndowndp = n + 1 - i
        sumup = sumup + 1/nup
        sumdown = sumdown + 1/ndown
        sumupdp = sumupdp + 1/nupdp
        sumdowndp = sumdowndp + 1/ndowndp

    end do
    print '(i8, 1x, 4f9.5, 3x, e11.5, 1x, e11.5)', n, sumdown, sumup,      &
        sumupdp, sumdowndp, (sumup - sumdowndp)/sumdowndp,          &
        (sumdown - sumdowndp)/sumdowndp

end do

END

```

-----  
 OUTPUT

| n        | down(sp) | up(sp)   | down(dp) | up(dp)   | error up    | error down  |
|----------|----------|----------|----------|----------|-------------|-------------|
| 10       | 2.92897  | 2.92897  | 2.92897  | 2.92897  | 0.59952E-07 | 0.59952E-07 |
| 100      | 5.18738  | 5.18738  | 5.18738  | 5.18738  | 0.79433E-07 | -.10441E-06 |
| 1000     | 7.48547  | 7.48548  | 7.48547  | 7.48547  | 0.10074E-05 | 0.11555E-06 |
| 10000    | 9.78760  | 9.78761  | 9.78761  | 9.78761  | 0.70283E-06 | -.17411E-06 |
| 100000   | 12.09015 | 12.09085 | 12.09015 | 12.09015 | 0.58287E-04 | 0.46790E-06 |
| 1000000  | 14.39265 | 14.35736 | 14.39273 | 14.39273 | -.24574E-02 | -.52224E-05 |
| 10000000 | 16.68603 | 15.40368 | 16.69531 | 16.69531 | -.77365E-01 | -.55585E-03 |

In the table or results, the columns for the error are *fractional errors* obtained by taking the difference between a single precision result, and the double precision(down) result.

Note:

- We expect the double precision results to be accurate and, indeed this is confirmed by the fact that the results obtained by summing up and summing down are the same to 5 decimal places.
- In single precision the results obtained by summing up and summing down are significantly different for large  $n$ .
- By comparing with the double precision results, we see that the single precision down sum is more accurate than the up sum.
- The errors come from adding small numbers to a large total, because, in this situation, the computer cannot represent accurately the difference between the totals before and after the addition.  $S^{(up)}$  is less accurate than  $S^{(down)}$  because, for  $S^{(up)}$ , the small numbers (of which there are many) are added to an already large total, whereas for  $S^{(down)}$  the small numbers are added to a small total which can be done quite accurately.

The moral is: don't add a lot of small numbers to a big one.

#### 4. SOURCE CODE

```
PROGRAM golden_mean
```

```

PARAMETER (NMAX = 50)
REAL phi(0: NMAX)
REAL*8 phi8(0: NMAX), golden

print '(A)', "    n  recurrence(SP) recurence (DP)  exact "
golden = (sqrt(5.d0) - 1)/2.d0
phi(0) = 1
phi(1) = golden
phi8(0) = 1

```

```

phi8(1) = golden
phimult = golden

do n = 2, NMAX
  phi(n) = phi(n-2) - phi(n-1)
  phi8(n) = phi8(n-2) - phi8(n-1)
  phimult = golden * phimult
  print '(i6, 3e15.5)', n, phi(n), phi8(n), phimult
end do

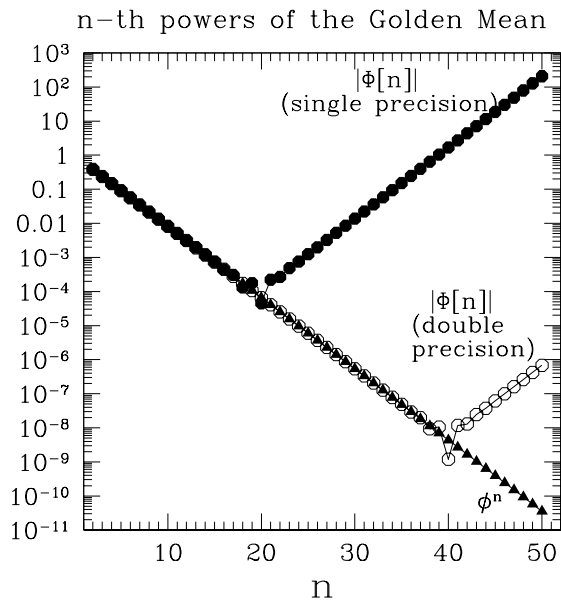
```

END PROGRAM golden\_mean

-----

OUTPUT

| n     | recurrence(SP) | recurrence (DP) | exact(i.e. from multiplication) |
|-------|----------------|-----------------|---------------------------------|
| 2     | 0.38197E+00    | 0.38197E+00     | 0.38197E+00                     |
| 3     | 0.23607E+00    | 0.23607E+00     | 0.23607E+00                     |
| 4     | 0.14590E+00    | 0.14590E+00     | 0.14590E+00                     |
| 5     | 0.90170E-01    | 0.90170E-01     | 0.90170E-01                     |
| ..... |                |                 |                                 |
| 47    | 0.48773E+02    | 0.16155E-06     | 0.15052E-09                     |
| 48    | -0.78917E+02   | -0.26106E-06    | 0.93024E-10                     |
| 49    | 0.12769E+03    | 0.42261E-06     | 0.57492E-10                     |
| 50    | -0.20661E+03   | -0.68367E-06    | 0.35532E-10                     |



See also the figure.

(a) The output and figure show the results for  $\phi^n$  obtained by directly multiplying.

(b) Starting with

$$\phi^{n+1} = \phi^{n-1} - \phi^n. \quad (1)$$

we divide by  $\phi^{n-1}$  to get the quadratic equation

$$\boxed{\phi^2 + \phi - 1 = 0}, \quad (2)$$

which has two solutions one of which is the golden mean  $\boxed{\phi = (-1 + \sqrt{5})/2 \simeq 0.618\dots}$

Hence, in principle, we can calculate  $\phi^n$  by constructing an array of numbers  $\Phi[0], \Phi[1], \dots$  from the following relation

$$\Phi[n+1] = \Phi[n-1] - \Phi[n], \quad (3)$$

with the “boundary conditions”  $\Phi[0] = 1, \Phi[1] = \phi$ . The value of  $\Phi[n]$  will then be  $\phi^n$ .

(c) The output and figure show that this algorithm is, however, unstable. Even if one uses double precision, eventually the error starts to grow exponentially.

(d) The problem is that there is another number which satisfies the quadratic equation, Eq. (2), namely  $\boxed{\tilde{\phi} = (-1 - \sqrt{5})/2 \simeq -1.618\dots}$ . Hence the *general* solution of the recurrence relation, Eq. (3) is

$$\boxed{\Phi[n] = A\phi^n + B\tilde{\phi}^n}, \quad (4)$$

where  $A$  and  $B$  are arbitrary constants. In principle, the boundary conditions we impose,  $\Phi[0] = 1, \Phi[1] = \phi$ , set  $A = 1, B = 0$ . However, because of roundoff errors, in practice  $B$  will not be identically zero but of order the machine precision  $\epsilon_m$ . Since  $|\tilde{\phi}| > |\phi|$ , for large enough  $n$  the  $B\tilde{\phi}^n$  term will dominate.

Hence the algorithm is *unstable*. The error grows *exponentially* with  $n$ .

One can estimate the value of  $n$  where the  $B\tilde{\phi}^n$  starts to dominate by setting

$$\boxed{\epsilon_m \left| \frac{\tilde{\phi}}{\phi} \right|^n = 1}. \quad (5)$$

With  $\epsilon_m = 10^{-7}$  (single precision) this gives  $n \simeq 17$  while for  $\epsilon_m = 10^{-16}$  (double precision) it gives  $n \simeq 38$ . These values correspond quite well to the points on the graph where the data from the recurrence relation deviates from the correct results obtained by multiplying.

5. (a) The Taylor series gives

$$\boxed{\frac{f(x+h) + f(x-h) - 2f(x)}{h^2} = f''(x) + \frac{h^2}{12}f^{(4)}(x) + \dots}. \quad (6)$$

which shows that the error in representing  $f''(x)$  by the finite difference expression on the LHS is of order  $h^2$ .

- (b) The program below takes  $f(x) = 1/x$  and evaluates the second derivative at  $x = 1$ . The answer is 2. The output shows that the results approach this with an error proportional to  $h^2$  since each factor of 10 decrease in  $h$  gives two more decimal places accuracy. Note the program uses double precision.

SOURCE CODE

```
REAL*8 f, x, h, df2dx2

f(x) = 1 / x

x = 1
h = 0.1d0

do i = 1, 4
    df2dx2 = ((f(x+h) - f(x)) - (f(x) - f(x-h)))/h**2
    print '(f10.5, 2x, f16.12)', h, df2dx2
    h = 0.1d0 * h
end do

END
```

-----  
OUTPUT

| h       | 2nd deriv.     |
|---------|----------------|
| 0.10000 | 2.020202020202 |
| 0.01000 | 2.000200020002 |
| 0.00100 | 2.000002000148 |
| 0.00010 | 2.000000017238 |

6. *For Physics 242 students only.*

SOURCE CODE (Fortran 90 with lots of comments)

```
!*****
!  
! Computes the spherical Bessel functions j_l(x) for a specified value of x
! for l=0, 1, ... 12.
!  
! Fortran 90 code.
!  
!*****
PROGRAM spherical_bessel

    INTEGER, PARAMETER :: NMAX = 51
```

```

      REAL(8), DIMENSION(0: NMAX) :: sph_bess
      REAL(8) :: x, scale
!
! Read in the value of x
!
      print '("x = ? ",$)'
      read *, x
      print '("x = ", f9.6/)', x
!
! Initialization; if NMAX is big enough, the answer is indepedent of what one
! puts here.
!
      sph_bess(NMAX) = 1; sph_bess(NMAX - 1) = 1
!
! Downward recursion; start with NMAX
!
      do i = NMAX - 1, 1, -1
          sph_bess(i-1) = ((2*i + 1)/x) * sph_bess(i) - sph_bess(i+1)
      end do
!
! Determine the scale factor by comparing with the known function j_0(x)
!
      scale = sph_bess(0) * x / sin(x)
!
! Rescale the l=0 to l=12 spherical Bessel functions by dividing by scale factor
!
      print '("  l  ", 7x, "j_l(x)")'
      do i = 0, 12
          sph_bess(i) = sph_bess(i) / scale
          print '(i4, 2x, e16.8)', i, sph_bess(i)
      end do

END PROGRAM spherical_bessel

```

-----  
OUTPUT

(Other l values are were generated by the computer code but are not printed here.)

x = 0.100000

|   |                |
|---|----------------|
| 1 | j_l(x)         |
| 3 | 0.95185197E-05 |
| 5 | 0.96163102E-09 |
| 8 | 0.29012001E-15 |

12 0.12646513E-24

x = 1.000000

|    |                |
|----|----------------|
| 1  | j_1(x)         |
| 3  | 0.90065811E-02 |
| 5  | 0.92561159E-04 |
| 8  | 0.28264988E-07 |
| 12 | 0.12416626E-12 |

x = 10.000000

|    |                 |
|----|-----------------|
| 1  | j_1(x)          |
| 3  | -0.39495845E-01 |
| 5  | -0.55534512E-01 |
| 8  | 0.12557802E+00  |
| 12 | 0.17216000E-01  |

Why does this work? As noted in the comments in the question, two sets of quantities satisfy the recursion relation

$$Y_{l-1} = \frac{2l+1}{x}Y_l - Y_{l+1}, \quad (7)$$

being used. These are

$$Y_l = j_l(x) \quad \text{and} \quad Y_l = n_l(x), \quad (8)$$

the spherical Bessel and spherical Neumann functions. Hence the *general* solution of Eq. (7) is

$$Y_l = Aj_l(x) + Bn_l(x), \quad (9)$$

where  $A$  and  $B$  are constants determined by the “initial conditions”, i.e. the choice of  $Y_l$  for the two largest values of  $l$ . Let’s denote by  $l_{\max}$  the largest value of  $l$ , and call  $C_1$  and  $C_2$  the initial guesses for  $l = l_{\max}$  and  $l_{\max} - 1$ , i.e.

$$\begin{aligned} C_1 &= Aj_{l_{\max}}(x) + Bn_{l_{\max}}(x) \\ C_2 &= Aj_{l_{\max}-1}(x) + Bn_{l_{\max}-1}(x). \end{aligned} \quad (10)$$

Now, for small  $x$ ,  $j_l(x) \propto x^l$  and  $n_l(x) \propto x^{-l}$ . Hence, if  $x$  is not too large (in practice this turns out to mean that  $x \ll l_{\max}$ ), we have  $j_{l_{\max}}(x) \ll n_{l_{\max}}(x)$ . Hence, if we choose simple values for  $C_1$  and  $C_2$  (here we set them equal to 1), we must have  $A \gg B$  in order that the term involving  $j$  makes a significant contribution to the RHS of Eqs. (10) (which in general it must unless, by some remarkable coincidence, we happened to choose  $C_1/C_2$  to precisely equal  $n_{l_{\max}}(x)/n_{l_{\max}-1}(x)$ ).

Iterating Eq. (7) downwards to small  $l$ ,  $j_l(x)$  and  $n_l(x)$  then have similar magnitude, but since the coefficient of  $j_l(x)$  is very much greater than that of  $n_l(x)$  in Eq. (7),  $Y_l$  will be equal to  $Aj_l(x)$ , to a good approximation. We determine  $A$  by continuing down to  $l = 0$  and comparing  $Y_0$  with the known expression for  $j_0(x)$ .

One might have thought that a simpler approach would be to use the *upward* recursion

$$j_{l+1}(x) = \frac{2l+1}{x}j_l(x) - j_{l-1}(x), \quad (11)$$

to compute  $j_l(x)$  starting, say, from the known expressions for  $j_0(x)$  and  $j_1(x)$ . However, this is actually *unstable*, at least for small values of  $x$ , and so is not useful in practice. The reason is that the general solution is still given by Eq. (9), but now  $A$  is close to 1 and  $B$  is close to zero. However,  $B$  is not identically zero because of roundoff errors, and when one iterates to large  $l$  where  $x \ll l$ , we have  $n_l(x) \gg j_l(x)$ , so the contribution from the  $Bn_l(x)$  piece grows and eventually dominates. The situation is similar to that found in Qu. (4) where powers of the golden mean are obtained using Eq. (3).