PHYSICS 115/242

Homework 1, Solutions

1. (a) SOURCE CODE

```
#include <stdio.h>

main()
{
    printf (" Hello World \n");
}
-----------
OUTPUT
 Hello World
```

   (b) i. SOURCE CODE

```
#include <math.h>
#include <stdio.h>

main()
{
    double x;
    x = 1.9e10;
    printf (" %11.1e \n", x);
}
-----------
OUTPUT
  1.9e+10
```

   ii. SOURCE CODE

```
#include <math.h>
#include <stdio.h>

main()
{
    double gm;
    gm = (sqrt((double)5) - 1)/2;
    printf (" The Golden Mean is %11.8f \n", gm);
}
-----------
OUTPUT
 The Golden Mean is  0.61803399
```

2. (a) SOURCE CODE

```
#include <math.h>
#include <stdio.h>

main()
{
    int i, ipower;

    i = 0;
    ipower = 0;

    while (i >= 0)
    {
        ipower += 1;
        i = 2*i + 1;
        printf (" %12d     2^%2d - 1\n", i, ipower);
    }
}
--------
OUTPUT
            1     2^ 1 - 1
            3     2^ 2 - 1
            7     2^ 3 - 1
            ..............
   1073741823     2^30 - 1
   2147483647     2^31 - 1
           -1     2^32 - 1
```

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

(b) First we do single precision.

```
SOURCE CODE

#include <math.h>
#include <stdio.h>

main()
{
    float x;
    int i;

    x = 1;

    for (i = 1; i <= 40; i++)
```

```
    {
        x *= 10;
        printf (" %2d     %16.4e \n", i, x);
    }
}
---
OUTPUT
  1           1.0000e+01
  2           1.0000e+02
  .......................
 36           1.0000e+36
 37           1.0000e+37
 38           1.0000e+38
 39                  inf
 40                  inf
```

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

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

```
   1           1.0000e+01
   2           1.0000e+02
   .......................
 307          1.0000e+307
 308          1.0000e+308
 309                  inf
 310                  inf
```

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

(c) First we do single precision:

```
CODE
#include <math.h>
#include <stdio.h>

main()
{
    float x;
    int i;

    x = 1;
    for (i = 1; i <= 40; i++)
    {
        x /= 10;
        printf (" %2d     %16.4e \n", i, x);
    }
```

```
}

OUTPUT
  k             1/10^k
-------------------------
  1           1.0000e-01
  2           1.0000e-02
  3           1.0000e-03
 ......................
 35           1.0000e-35
 36           1.0000e-36
 37           1.0000e-37
 38           0.0000e+00
 39           0.0000e+00
```

This shows that with my compiler, icc, the smallest single precision number different from zero is between 10^{-37} and 10^{-38}.

Next we do double precision by simply declaring `x` to be double and increasing the range of the loop:

```
OUTPUT
  k           10^{-k}
---------------------------
  1           1.0000e-01
  2           1.0000e-02
  3           1.0000e-03
.........................
 306          1.0000e-306
 307          1.0000e-307
 308          0.0000e+00
 309          0.0000e+00
```

This shows that the smallest double precision number different from zero using the icc compiler is between 10^{-307} and 10^{-308}.

Note: Some compilers, such as gcc, can represent slightly smaller numbers by not assuming a hidden "1" at the beginning of the mantissa if the exponent is zero. In this way the smallest number different from zero has a 1 in the least significant bit in the mantissa (and exponent zero) and so is about ϵ_m times smaller than the above numbers, where ϵ_m is the machine precision (related to the number of bits in the mantissa, see below). This gives about 10^{-45} for single precision and about 10^{-323} for double precision.

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

```
#include <math.h>
#include <stdio.h>

main()
{
    float x, y, eps;
    int i;

    eps = 1;
    y = 0;

    while (1 - y != 0)
    {
        eps /= 10;
        y = 1 - eps;
        printf (" %12.5e   %12.5e \n", eps, 1-y);
    }
}
```

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

```
  1.00000e-01    1.00000e-01
  1.00000e-02    9.99999e-03
  ..........................
  1.00000e-07    1.19209e-07
  1.00000e-08    0.00000e+00
```

For double precision, I simply declare y and ϵ to be double and the output is then

```
  1.00000e-01    1.00000e-01
  1.00000e-02    9.99999e-03
  ..........................
  1.00000e-16    1.11022e-16
  1.00000e-17    0.00000e+00
```

which shows that there are about $\boxed{\text{16 bits of precision}}$ using double precision.

3. SOURCE CODE

```
#include <math.h>
#include <stdio.h>
main()
{
    double sumupdp, sumdowndp, nupdp, ndowndp;
    float  nup, ndown, sumup, sumdown;
    int n, i, idec;
```

```
    n = 1;
    printf ("       n    down(sp)  up(sp)  down(dp)  up(dp)     error up     error down

    for (idec = 0; idec < 7; idec++)
    {
        n *= 10;
        sumup = 0; sumdown = 0; sumupdp = 0; sumdowndp = 0;

        for (i = 1; i <= n; i++)
        {
            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;
        }

        printf (" %8d %9.4f%9.4f%9.4f%9.4f   %12.4e %12.4e\n",
                n, sumdown, sumup, sumupdp, sumdowndp,
                (sumup - sumdowndp)/sumdowndp, (sumdown - sumdowndp)/sumdowndp);
    }
}
--------
OUTPUT
       n    down(sp)  up(sp)  down(dp)  up(dp)     error up     error down
      10    2.9290   2.9290   2.9290   2.9290     5.9952e-08   5.9952e-08
     100    5.1874   5.1874   5.1874   5.1874     7.9433e-08  -1.0441e-07
    1000    7.4855   7.4855   7.4855   7.4855     1.0074e-06   1.1555e-07
   10000    9.7876   9.7876   9.7876   9.7876     7.0283e-07  -1.7411e-07
  100000   12.0902  12.0909  12.0901  12.0901     5.8287e-05   5.4678e-07
 1000000   14.3927  14.3574  14.3927  14.3927    -2.4574e-03  -5.2224e-06
10000000   16.6860  15.4037  16.6953  16.6953    -7.7365e-02  -5.5585e-04
```

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:

(a) 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 at least the four decimal places printed.

(b) In single precision the results obtained by summing up and summing down are significantly different for large n.

(c) By comparing with the double precision results, we see that the single precision down sum is more accurate than the up sum.

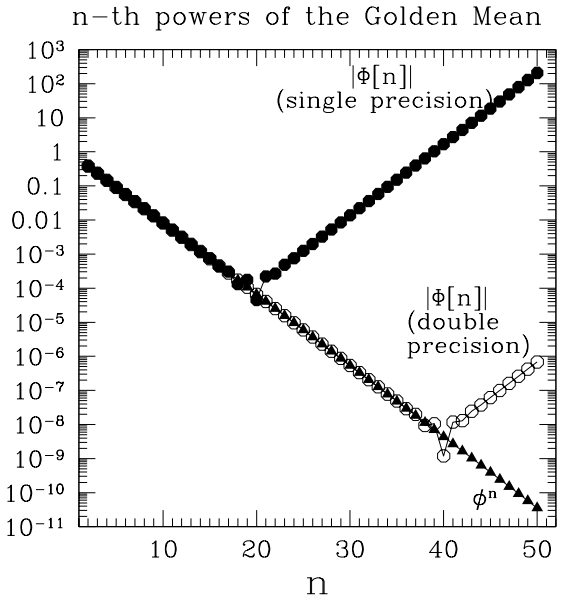
(d) The errors come from adding small numbers to a large total, because, in this situation, the computer cannot represent accurately the difference between the totals before and after the addition. $S^{(up)}$ is less accurate than $S^{(down)}$ because, for $S^{(up)}$, the small numbers (of which there 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

```
#include <math.h>
#include <stdio.h>

#define NMAX   50
main()
{
    double golden, phidp[NMAX], phimult;
    float  phi[NMAX];
    int n;

    printf ("      n   recurrence(SP)  recurence (DP)  exact\n");
    golden = (sqrt((double) 5) - 1) / 2 ;
    phi[0] = 1;
    phi[1] = golden;
    phidp[0] = 1;
    phidp[1] = golden;
    phimult = golden;

    for (n = 2; n <= NMAX; n++)
    {
        phi[n] = phi[n-2] - phi[n-1];
        phidp[n] = phidp[n-2] - phidp[n-1];
        phimult = golden * phimult;
        printf (" %6d%15.5e%15.5e%15.5e\n", n, phi[n], phidp[n], phimult);
    }

}
-----------
OUTPUT
      n   recurrence(SP)  recurence (DP)  exact
      2    3.81966e-01    3.81966e-01    3.81966e-01
      3    2.36068e-01    2.36068e-01    2.36068e-01
      4    1.45898e-01    1.45898e-01    1.45898e-01
```

```
      5    9.01700e-02    9.01699e-02    9.01699e-02
      6    5.57280e-02    5.57281e-02    5.57281e-02
..................
     16    4.36902e-04    4.53104e-04    4.53104e-04
     17    3.06249e-04    2.80034e-04    2.80034e-04
     18    1.30653e-04    1.73070e-04    1.73070e-04
     19    1.75595e-04    1.06963e-04    1.06963e-04
     20   -4.49419e-05    6.61070e-05    6.61070e-05
..................
     35    1.51471e-01    4.89667e-08    4.84655e-08
     36   -2.45085e-01    2.91423e-08    2.99533e-08
     37    3.96556e-01    1.98244e-08    1.85122e-08
     38   -6.41640e-01    9.31784e-09    1.14411e-08
     39    1.03820e+00    1.05066e-08    7.07102e-09
     40   -1.67984e+00   -1.18878e-09    4.37013e-09
```



See also the figure.

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

(b) Starting with

$$\phi^{n+1} = \phi^{n-1} - \phi^n. \tag{1}$$

we divide by ϕ^{n-1} to get the quadratic equation

$$\boxed{\phi^2 + \phi - 1 = 0,} \tag{2}$$

which has two solutions one of which is the golden mean $\boxed{\phi = (-1 + \sqrt{5})/2 \simeq 0.618\ldots.}$ Hence, in principle, we can calculate ϕ^n by constructing an array of numbers $\Phi[0], \Phi[1], \cdots$ from the following relation

$$\Phi[n+1] = \Phi[n-1] - \Phi[n], \tag{3}$$

with the "boundary conditions" $\Phi[0] = 1, \Phi[1] = \phi$. The value of $\Phi[n]$ will then be ϕ^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 $\boxed{\tilde{\phi} = (-1 - \sqrt{5})/2 \simeq -1.618\ldots.}$ Hence the *general* solution of the recurrence relation, Eq. (3) is

$$\Phi[n] = A\phi^n + B\tilde{\phi}^n\,, \tag{4}$$

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 ϵ_m. In other words the numerical values are given approximately by

$$\boxed{\phi[n] = \phi^n + \epsilon_m\,\tilde{\phi}^n\,.} \tag{5}$$

Since $|\tilde{\phi}| > |\phi|$, for large enough n the $\tilde{\phi}^n$ term will dominate. Hence the algorithm is *unstable*, i.e. the error grows *exponentially* with n.

Equating the two terms on the RHS of Eq. (5), we see that the $\tilde{\phi}^n$ term dominates for $n > n^\star$, where

$$\boxed{\epsilon_m \left|\frac{\tilde{\phi}}{\phi}\right|^{n^\star} = 1\,.} \tag{6}$$

Since $|\tilde{\phi}/\phi| = 2.618\ldots$, if we take $\epsilon_m = 10^{-7}$ (single precision) then Eq. (6) gives $\boxed{n^\star \simeq 17}$ while if we take $\epsilon_m = 10^{-16}$ (double precision) then Eq. (6) gives $\boxed{n^\star \simeq 38}$. 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.

For $n < n^\star$, which is the stable region, the first term on the RHS of Eq. (5) dominates, while for $n > n^\star$, the unstable region, the second term on the RHS of Eq. (5) dominates. Since $|\tilde{\phi}| = 1/\phi$, the slope of the data in the figure in the unstable region is equal in magnitude (but opposite in sign) to that in the stable region. (Thanks to John Faulkner for pointing this out.)

5. (a) The Taylor series gives

$$\boxed{\frac{f(x+h) + f(x-h) - 2f(x)}{h^2} = f''(x) + \frac{h^2}{12} f^{(4)}(x) + \cdots\,.} \tag{7}$$

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 $\boxed{\text{error proportional to } h^2}$ since each factor of 10 decrease in h gives two more decimal places accuracy. The program uses double precision. Note that for $h = 0.0001$ the approximation error is dominated by roundoff errors.

source code

```
#include <math.h>
#include <stdio.h>

double f(double x)
{
        return 1 / x;                // the rhs of the equation
}

main()
{
    double f(double x), x, h, df2dx2;
    int i;

    x = 1;
    h = 0.1;
    printf ("      h           2nd deriv.\n");

    for (i = 0; i < 4; i++)
    {
        df2dx2 = ((f(x+h) - f(x)) - (f(x) - f(x-h)))/(h*h) ;
        printf (" %10.5f  %16.10f \n", h, df2dx2);
        h /= 10;
    }

}
```

OUTPUT

```
      h           2nd deriv.
    0.10000      2.0202020202
    0.01000      2.0002000200
    0.00100      2.0000020002
    0.00010      2.0000000100
```

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

   l         j_l(x)
   3    0.95185197E-05
   5    0.96163102E-09
   8    0.29012001E-15
  12    0.12646513E-24

x =  1.000000

   l         j_l(x)
   3    0.90065811E-02
   5    0.92561159E-04
   8    0.28264988E-07
  12    0.12416626E-12

x = 10.000000

   l         j_l(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

$$Y_{l-1} = \frac{2l+1}{x} Y_l - Y_{l+1}, \tag{8}$$

being used. These are

$$Y_l = j_l(x) \qquad \text{and} \quad Y_l = n_l(x)\,, \tag{9}$$

the spherical Bessel and spherical Neumann functions. Hence the *general* solution of Eq. (8) is

$$Y_l = A j_l(x) + B n_l(x), \tag{10}$$

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.

$$\begin{aligned} C_1 &= A j_{l_{\max}}(x) + B n_{l_{\max}}(x) \\ C_2 &= A j_{l_{\max}-1}(x) + B n_{l_{\max}-1}(x)\,. \end{aligned} \tag{11}$$

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. (11) (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. (8) 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. (8), for small l, 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

$$j_{l+1}(x) = \frac{2l+1}{x} j_l(x) - j_{l-1}(x), \tag{12}$$

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. (10), 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).