Okeefe Niemann
4/2/2014
1281465
PHYS 115

Assignment 1

1 Question 1

1.1 Part a

- Input:

```
int main()
{
printf ("\n Hello World!! \n\n");

return 0;
}
```

- Output:

```
Hello World!!
```

1.2 Part b

- Input:

```
#include <stdio.h>
#include <math.h>
main()
{
double i;
i = 19000000000;
printf ("\n This is a really big number: %.1e \n", i);

double golden_mean;
golden_mean = (sqrt(5)-1)/2;
printf("\n The value of the golden mean is %.8f. \n" , golden_mean);

return 0;
}
```

- Output:

```
This is a really big number: 1.9e+10

The value of the golden mean is 0.61803399.
```

2 Question 2

2.1 Part a

- Input:

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

int main() {
int j, value=1;

for(j=1;j<100; j++){
        if (value > 0){
                value *= 2;
                printf("%d %d\n", j, value);
        }
        else {
                break;
        }
}
value = value-1;

printf("The largest possible value is %d.\n", value);

return 0;
}
```

- output:

```
1 2
2 4
3 8
4 16
5 32
...
29 536870912
30 1073741824
31 -2147483648
The largest possible value is 2147483647.
```

2.2 Part b

Part i (Floats):

- Input:

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

int main() {
float j;
float value=1;

for(j=0; j<127; j++){
        if (value < INFINITY){
                value *= 2;
                printf("%f %.5e\n", j, value);
        }
        else {
                break;
        }
}
value=value*1.9999999;   /*Outputs roughly the largest value before the float overflows*/
printf("The largest value of a float is %e.5. \n", value);

return 0;
}
```

- Output:

```
...
121.000000 5.31691e+36
122.000000 1.06338e+37
123.000000 2.12676e+37
124.000000 4.25353e+37
125.000000 8.50706e+37
126.000000 1.70141e+38
The largest value of a float is 3.402823e+38.5.
```

Part ii (Doubles):

- Input:

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

int main() {

double i;
double value=1;

for(i=0;i<1023; i++){
        if (value < INFINITY){
                value *= 2;
                printf("%f %.5e\n", i, value);
        }
        else {
                break;
        }

}
value = value*1.999999999999999;  /*Outputs roughly the largest value before the double overflows*/
printf("The largest value of a double is roughly %e.5. \n", value);

return 0;
}
```

- Output:

```
...
1015.000000 7.02224e+305
1016.000000 1.40445e+306
1017.000000 2.80890e+306
1018.000000 5.61779e+306
1019.000000 1.12356e+307
1020.000000 2.24712e+307
1021.000000 4.49423e+307
1022.000000 8.98847e+307
The largest value of a double is roughly 1.797693e+308.5.
```

2.3 Part C

Note: Only the last few terms have been selected in the output sections due to the extremely large amount of numbers outputted, leading to the inability to view the first large cluster of terms.

Part i (Floats):

- Input:

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

int main() {
float j;
float value=1;

for(j=0;j<200; j++){
        if (value > 0){
                value *= .5;
                printf("%f %.5e\n", j, value);
        }
        else {
                break;
        }
}

printf("The smallest value of a float is roughly 1.40e-45.\n");
return 0;
}
```

- Output

```
...
144.000000 2.24208e-44
145.000000 1.12104e-44
146.000000 5.60519e-45
147.000000 2.80260e-45
148.000000 1.40130e-45
149.000000 0.00000e+00
The smallest value of a float is roughly 1.40e-45.
```

Part ii (Doubles):

- Input:

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

int main(){
double i;
double value=1;

for(i=0;i<1076; i++){
        if (value > 0){
                value *= .5;
                printf("%f %.5e\n", i, value);
        }
        else {
                break;
        }
}
printf("The smallest value of a double is roughly 4.94e-324.\n");

return 0;
}
```

- Output:

```
1070.000000 3.95253e-323
1071.000000 1.97626e-323
1072.000000 9.88131e-324
1073.000000 4.94066e-324
1074.000000 0.00000e+00
The smallest value of a double is roughly 4.94e-324.
```

2.4 Part D

Part i (Floats):

- Input:

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

int main(){

float i;
float epsilon = 1;

for (i=0; i<100; i++) {
        float x = 1 - epsilon, y = 1 - x;
        if (y > 0) {
                epsilon *= .1;
                printf("%.5e, %.5e, %.5e\n", epsilon, x, y);
        }
        else {
                break;
        }
}
printf("\nFloats have a precision of about 1.19209e-07.\n");
}
```

- Output:

```
1.00000e-01, 0.00000e+00, 1.00000e+00
1.00000e-02, 9.00000e-01, 1.00000e-01
1.00000e-03, 9.90000e-01, 9.99999e-03
1.00000e-04, 9.99000e-01, 9.99987e-04
1.00000e-05, 9.99900e-01, 1.00017e-04
1.00000e-06, 9.99990e-01, 1.00136e-05
1.00000e-07, 9.99999e-01, 1.01328e-06
1.00000e-08, 1.00000e+00, 1.19209e-07

Floats have a precision of 1.19209e-07.
```

Part ii (Doubles):

- Input:

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

int main(){

double i;
double epsilon = 1;

for (i=0; i<100; i++) {
        double x = 1 - epsilon;
        double y = 1 - x;
        if (y > 0) {
                epsilon *= .1;
                printf("%.5e, %.5e, %.5e\n", epsilon, x, y);
        }
        else {
                break;
        }
}
printf("\nDoubles have a precision of about 1.11022e-16.\n");
}
```

- Output:

```
1.00000e-13, 1.00000e+00, 9.99978e-13
1.00000e-14, 1.00000e+00, 1.00031e-13
1.00000e-15, 1.00000e+00, 9.99201e-15
1.00000e-16, 1.00000e+00, 9.99201e-16
1.00000e-17, 1.00000e+00, 1.11022e-16

Doubles have a precision of about 1.11022e-16.
```

3 Question 3

3.1 Part a

- Input:

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

int main(){
float n, m, p, sum, othersum;
int i, array[7] = {100, 1000, 10000, 100000, 1000000, 10000000, 100000000};

for (i = 0; i<6; i++) {
        p = array[i];

        for (n = 1; n <= array[i]; n++) {
                sum += (1/n);
                }
                printf("The sum from 1 to %d of 1/n is %f.\n", array[i],  sum);

        for (m = array[i]; m >= 1; m--) {
                othersum += (1/m);
                }
                printf("The sum from %d to 1 of 1/n is %f.\n", array[i], othersum);
}
return 0;
}
```

- output:

```
The sum from 1 to 100 of 1/n is 5.187378.
The sum from 100 to 1 of 1/n is 5.187377.
The sum from 1 to 1000 of 1/n is 12.672853.
The sum from 1000 to 1 of 1/n is 12.672853.
The sum from 1 to 10000 of 1/n is 22.460485.
The sum from 10000 to 1 of 1/n is 22.460478.
The sum from 1 to 100000 of 1/n is 34.555584.
The sum from 100000 to 1 of 1/n is 34.550098.
The sum from 1 to 1000000 of 1/n is 48.572968.
The sum from 1000000 to 1 of 1/n is 48.567486.
The sum from 1 to 10000000 of 1/n is 62.590355.
The sum from 10000000 to 1 of 1/n is 62.584873.
```

3.2 Part b

- Input:

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

int main(){
double n, m, p, sum, othersum;
int i, array[7] = {100, 1000, 10000, 100000, 1000000, 10000000, 100000000};

for (i = 0; i<6; i++) {
        p = array[i];

        for (n = 1; n <= array[i]; n++) {
                sum += (1/n);
                }
                printf("The sum from 1 to %d of 1/n is %f.\n", array[i],  sum);

        for (m = array[i]; m >= 1; m--) {
                othersum += (1/m);
                }
                printf("The sum from %d to 1 of 1/n is %f.\n", array[i], othersum);
}
return 0;
}
```

- Output:

```
The sum from 1 to 100 of 1/n is 5.187378.
The sum from 100 to 1 of 1/n is 5.187378.
The sum from 1 to 1000 of 1/n is 12.672848.
The sum from 1000 to 1 of 1/n is 12.672848.
The sum from 1 to 10000 of 1/n is 22.460454.
The sum from 10000 to 1 of 1/n is 22.460454.
The sum from 1 to 100000 of 1/n is 34.550601.
The sum from 100000 to 1 of 1/n is 34.550601.
The sum from 1 to 1000000 of 1/n is 48.943327.
The sum from 1000000 to 1 of 1/n is 48.943327.
The sum from 1 to 10000000 of 1/n is 65.638639.
The sum from 10000000 to 1 of 1/n is 65.638639.
```

3.3 Part c

- Input:

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

int main(){
double n, m, p, sum, othersum, difference;
int i, array[7] = {100, 1000, 10000, 100000, 1000000, 10000000, 100000000};

for (i = 0; i<6; i++) {
        p = array[i];

        for (n = 1; n <= array[i]; n++) {
                sum += (1/n);
                }
        for (m = array[i]; m >= 1; m--) {
                othersum += (1/m);
                }
        difference = sum - othersum;
        printf("The difference between S(up) and S(down) for range(1, %d)
        is %f.\n", array[i], difference);
}
return 0;
}
```

- Output:

```
The difference between S(up) and S(down) for range(1, 100) is -0.000000.
The difference between S(up) and S(down) for range(1, 1000) is 0.000000.
The difference between S(up) and S(down) for range(1, 10000) is 0.000000.
The difference between S(up) and S(down) for range(1, 100000) is 0.000000.
The difference between S(up) and S(down) for range(1, 1000000) is 0.000000.
The difference between S(up) and S(down) for range(1, 10000000) is -0.000000.
```

3.4 Part d

- Input:

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

int main(){
float n, m, p;
double q, r, sum, othersum, sum1, othersum1, difference, difference1;
int i, array[8] = {100, 1000, 10000, 100000, 1000000, 10000000, 100000000};

for (i = 0; i<6; i++) {
        p = array[i];

        for (n = 1; n <= array[i]; n++) {
                sum += (1/n);
                }
        for (m = array[i]; m >= 1; m--) {
                othersum += (1/m);
                }

        for (r = array[i]; r >=1; r--) {
                othersum1 += (1/r);
                }
        for (q = 1; q <= array[i]; q++) {
                sum1 += (1/q);
                }

difference1 = sum - sum1;
difference = othersum - othersum1;

printf("The precision of the float to the double summing from 1 to %d is %1.8e.
\n", array[i], difference);

printf("The precision of the float to the double summing down from %d to 1 is
%1.8e. \n", array[i], difference1);

}
return 0;
}
```

- Output:

```
The precision of the float to the double summing
from 1 to 100 is 5.81453001e-08.
The precision of the float to the double summing down
from 100 to 1 is 5.81453010e-08.
The precision of the float to the double summing
from 1 to 1000 is 1.21422914e-07.
The precision of the float to the double summing down
from 1000 to 1 is 1.21422897e-07.
The precision of the float to the double summing
from 1 to 10000 is 1.86948942e-07.
The precision of the float to the double summing down
from 10000 to 1 is 1.86948920e-07.
The precision of the float to the double summing
from 1 to 100000 is 2.52482579e-07.
The precision of the float to the double summing down
from 100000 to 1 is 2.52482032e-07.
The precision of the float to the double summing
from 1 to 1000000 is 3.18087054e-07.
The precision of the float to the double summing down
from 1000000 to 1 is 3.18086506e-07.
The precision of the float to the double summing
from 1 to 10000000 is 3.83666858e-07.
The precision of the float to the double summing down
from 10000000 to 1 is 3.83405578e-07.
```

Comment: As shown above, the precision of the float with respect to the double of Sum$^{(up)}$ is less accurate than Sum$^{(down)}$. It is also worth noting that the precision of the float's Sum$^{(up)}$ gets worse with increasing N.

3.5 Part e

S$^{(up)}$ is less accurate than S$^{(down)}$ because adding the smaller terms last rounds them to zero due to these terms being less than the precision of the outputted double at that point. In adding the small terms first, this eliminates the error made by the computer and effectively creates a more accurate representation of the harmonic series.

4 Question 4

4.1 Part a

- Input:

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

int main(){
double i, j, phi;

for (i=1; i <= n; i++){  /* Set n equal to the desired term */
        if (i <= 1) {
                 j= i;
        }
        else {
        phi= (sqrt(5)-1)/2;
        j *= phi;
        }
        printf("The value of the golden mean to the power of %.0f is %.5e.\n",
        i-1, j);
}

}
```

Note: The above code must be modified by setting n equal to whatever value term is needed. The "for" loop will then print all the terms up to the inputted value.

4.2 Part b

Comment: Refer to the last page of this assignment for an analytical solution.

4.3 Part c

part i (Floats):

- Input:

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

int main() {
        float n, phi_0 = 1, phi_1 = (sqrt(5)-1)/2, phi_ho;
        for (n = 0; n < 50; n++) {
                if (n < 1){       /* Defines the phi raised to the zeroeth power */
                        phi_ho = phi_0;
                }
                else if (n < 2){
                        phi_ho = phi_1;
                else {            /* Establishes recursive relationship*/
                        phi_ho = phi_0 - phi_1;
                        phi_0 = phi_1;
                        phi_1 = phi_ho;
                }
                printf("The recursive formula shows %.5e for term %.0f. \n",
                phi_ho, n);
        }
        return 0;
}
```

- Output:

```
The recursive formula shows 1.00000e+00 for term 1.
The recursive formula shows 6.18034e-01 for term 2.
The recursive formula shows 3.81966e-01 for term 3.
...
The recursive formula shows 1.18005e-03 for term 15.
The recursive formula shows 7.43151e-04 for term 16.
The recursive formula shows 4.36902e-04 for term 17.<----Diverges from 4a
The recursive formula shows 3.06249e-04 for term 18.
The recursive formula shows 1.30653e-04 for term 19.
The recursive formula shows 1.75595e-04 for term 20.
The recursive formula shows -4.49419e-05 for term 21.
...
The recursive formula shows -3.01434e+01 for term 47.
The recursive formula shows 4.87731e+01 for term 48.
The recursive formula shows -7.89165e+01 for term 49.
The recursive formula shows 1.27690e+02 for term 50.
```

Part ii:

- Input:

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

int main() {
        double n, phi_0 = 1, phi_1 = (sqrt(5)-1)/2, phi_ho;
        for (n = 0; n < 50; n++) {
                if (n < 1){      /* Defines the phi raised to the zeroeth power */
                        phi_ho = phi_0;
                }
                else if (n < 2){
                        phi_ho = phi_1;
                }
                else {           /* Establishes recursive relationship*/
                        phi_ho = phi_0 - phi_1;
                        phi_0 = phi_1;
                        phi_1 = phi_ho;
                }
             printf("The recursive formula yields %.5e for term %.0f. \n",
             phi_ho, n+1);
        }
        return 0;
}
```

- Output:

```
The recursive formula yields 1.00000e+00 for term 1.
The recursive formula yields 6.18034e-01 for term 2.
The recursive formula yields 3.81966e-01 for term 3.
...
The recursive formula yields 2.91423e-08 for term 37.
The recursive formula yields 1.98244e-08 for term 38.
The recursive formula yields 9.31784e-09 for term 39. <----Diverges from 4a
The recursive formula yields 1.05066e-08 for term 40.
The recursive formula yields -1.18878e-09 for term 41.
The recursive formula yields 1.16954e-08 for term 42.
The recursive formula yields -1.28842e-08 for term 43.
...
The recursive formula yields 1.61550e-07 for term 48.
The recursive formula yields -2.61057e-07 for term 49.
The recursive formula yields 4.22608e-07 for term 50.
```

Comment: When the the code in 4a is executed to fifty terms, I found that there was consistencies in the continually decreasing terms. When comparing the recursive relationship of floats and doubles to these terms, I found that the floats hold accuracy to within $\frac{1}{10}$ until about the 17^{th} term, while the doubles held to within $\frac{1}{10}$ until about the 39^{th} term. After this point, the code begins returning incohesive values oscillating between positive and negative. This behavior is proof that the recursive nature of the golden mean is an unstable algorithm.

4.4 Part D

Comment: In part b, I proved that the general solution for the recursive relation is

$$\phi = A\phi_1 + B\phi_2$$

where $\phi_1 = \frac{-1+sqrt(5)}{2}$ and $\phi_2 = \frac{-1-sqrt(5)}{2}$. This general solution is instable because, while ϕ_1 is purely recursive (not taking into account rounding errors), adding in the ϕ_2 term plays a greater role as the recursion formula reduces the value to the same magnitude as it. For example, when $B = .001$, the recursive relation holds until the value of ϕ reduces to a number of order 10^{-3}. The amplification of this error within the algorithm thus increases exponentially each time the recursive algorithm is carried out. An example of the general solution is displayed below.

- Input:

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

int main() {
        double n, phi_ho phi_0 = 1,
        phi_1 = .999*((sqrt(5)-1)/2) + .001*((-sqrt(5)-1)/2);
        for (n = 0; n < 50; n++) {
                if (n < 1){       /* Defines the phi raised to the zeroeth power */
                        phi_ho = phi_0;
                }
                else if (n < 2){
                        phi_ho = phi_1;
                }
                else {            /* Establishes recursive relationship*/
                        phi_ho = phi_0 - phi_1;
                        phi_0 = phi_1;
                        phi_1 = phi_ho;
                }
                printf("The recursive formula shows %.5e for term %.0f. \n",
                phi_ho, n+1);
                }
}
```

- Output:

```
The recursive formula shows 1.00000e+00 for term 1.
The recursive formula shows 6.15798e-01 for term 2.
The recursive formula shows 3.84202e-01 for term 3.
The recursive formula shows 2.31596e-01 for term 4.
The recursive formula shows 1.52606e-01 for term 5.
The recursive formula shows 7.89896e-02 for term 6.
The recursive formula shows 7.36166e-02 for term 7.
The recursive formula shows 5.37297e-03 for term 8.
The recursive formula shows 6.82437e-02 for term 9.
The recursive formula shows -6.28707e-02 for term 10.
The recursive formula shows 1.31114e-01 for term 11.
The recursive formula shows -1.93985e-01 for term 12.
...
The recursive formula shows -6.64384e+06 for term 48.
The recursive formula shows 1.07500e+07 for term 49.
The recursive formula shows -1.73938e+07 for term 50.
```

Comment: Note the increasing loss of accuracy as the recursive formula generates values to term eight, followed by utter nonsense reaching high positive powers by term 50. The number used for this calculation is:

$$\tilde{\phi} = .999\phi_1 + .001\phi_2$$

5 Problem 5

5.1 Part a

Comment: Refer to the last page of this assignment for an analytical solution.

5.2 Part b

Comment: The function used for the below algorithm is:

$$f(x) = \frac{1}{1+x}$$

The technique used to find the error is taking the difference between f_{mid}"(x) defined in question $5a$ and $f''(x)$ calculated numerically, scaled by a factor of h^2. This should (and does) approach a constant which is of the same order as h^2.

- Input:

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

int main(){
int i;
double x = 4, f_mid, f_exact, error, h = 1, a, b, c;
for (i = 1; i < 40; i++) {
        a = 1/(1 + x + h);
        b = 1/(1 + x - h);
        c = 1/(1 + x);
        f_mid = (a + b - 2*c)/pow(h, 2); /*2nd derivative as defined in 5a*/
        f_exact = 2 / pow(x+1, 3);     /*2nd derivative derived analytically*/
        error = (f_mid - f_exact)/pow(h, 2);
        printf("The error of f_mid''(x) with value h = (%.5e)
        is %f. \n", h, error);
        h *= .9;
        }
return 0;
}
```

- Output:

```
The error of f_mid''(x) with value h = (1.00000e+00) is 0.000667.
The error of f_mid''(x) with value h = (9.00000e-01) is 0.000661.
The error of f_mid''(x) with value h = (8.10000e-01) is 0.000657.
The error of f_mid''(x) with value h = (7.29000e-01) is 0.000654.
...
The error of f_mid''(x) with value h = (3.43368e-02) is 0.000640.
The error of f_mid''(x) with value h = (3.09032e-02) is 0.000640.
The error of f_mid''(x) with value h = (2.78128e-02) is 0.000640.
The error of f_mid''(x) with value h = (2.50316e-02) is 0.000640.
The error of f_mid''(x) with value h = (2.25284e-02) is 0.000640.
The error of f_mid''(x) with value h = (2.02756e-02) is 0.000640.
The error of f_mid''(x) with value h = (1.82480e-02) is 0.000640.
```