Обработка выходных данных

Первичный результат при запуске программы показал:
(Сейчас и далее первый пункт означает опыт с использованием $float$, второй - $double$)

1) Цикл не остановился и шел до переполнения $count$
2) Цикл остановился на значении $count$ = 21.

Test 0

Почему так происходит?

Уменьшим допустимое значение счетчика и будем выводить промежуточные вычисления: значения x и y.

count	x_double	y_double
0	-0.000000	2.000000
1	0.500000	1.500000
2	0.750000	1.250000
3	0.875000	1.125000
4	0.937500	1.062500
5	0.968750	1.031250
6	0.984375	1.015625
7	0.992187	1.007813
8	0.996094	1.003906
9	0.998047	1.001953
10	0.999023	1.000977
11	0.999512	1.000488
12	0.999756	1.000244
13	0.999878	1.000122
14	0.999939	1.000061
15	0.999969	1.000031
16	0.999985	1.000015
17	0.999992	1.000008
18	0.999996	1.000004
19	0.999998	1.000002
20	0.999999	1.000001
21	1.000000	1.000000

count	x_float	y_float
0	0.002052	1.997948
1	0.500347	1.499653
2	0.750687	1.249313
3	0.877048	1.122952
4	0.939037	1.060963
5	0.970032	1.029968
6	0.984337	1.015663
7	0.993874	1.006126
8	0.996258	1.003742
9	0.998642	1.001358
10	1.001026	0.998974
11	1.001026	0.998974
12	1.001026	0.998974
13	1.001026	0.998974
14	1.001026	0.998974
15	1.001026	0.998974
16	1.001026	0.998974
17	1.001026	0.998974
18	1.001026	0.998974
19	1.001026	0.998974
20	1.001026	0.998974
21	1.001026	0.998974

Test 1

Заметим, что значения $double$ близки к реальным значениям, когда $float$ больше похож на несвязный набор цифр.

Рассмотрим (1) более подробно: увеличим кол-во знаков после запятой

count	x_float	y_float
0	0.00205230712890625000	1.99794769287109375000
1	0.50034713745117187500	1.49965286254882812500
2	0.75068664550781250000	1.24931335449218750000
3	0.87704849243164062500	1.12295150756835937500
4	0.93903732299804687500	1.06096267700195312500
5	0.97003173828125000000	1.02996826171875000000
6	0.98433685302734375000	1.01566314697265625000
7	0.99387359619140625000	1.00612640380859375000
8	0.99625778198242187500	1.00374221801757812500
9	0.99864196777343750000	1.00135803222656250000
10	1.00102615356445312500	0.99897384643554687500
11	1.00102615356445312500	0.99897384643554687500
12	1.00102615356445312500	0.99897384643554687500
13	1.00102615356445312500	0.99897384643554687500
14	1.00102615356445312500	0.99897384643554687500
15	1.00102615356445312500	0.99897384643554687500
16	1.00102615356445312500	0.99897384643554687500
17	1.00102615356445312500	0.99897384643554687500
18	1.00102615356445312500	0.99897384643554687500
19	1.00102615356445312500	0.99897384643554687500
20	1.00102615356445312500	0.99897384643554687500
21	1.00102615356445312500	0.99897384643554687500

совпадают

Test 2

Бессмысленно еще увеличивать кол-во знаков после запятой, эти данные заполнены нулями.
Заметим, что начиная с $count$ равного 10 числа остаются неизменными. Что странно ведь они должны приближаться к 1.

Рассмотрим по отдельности каждую операцию вычисления x и y:
Выведем и сравним $(2.0001f + delta)$, $(2.0001f + delta - 2.0f)$, (y)

count	sum	sub = sum - 2.0f	div = sub * 10^4
0	2.00019979476928710938	0.00019979476928710938	1.9979476928710937500
1	2.00014996528625488281	0.00014996528625488281	1.49965286254882812500
2	2.00012493133544921875	0.00012493133544921875	1.24931335449218750000
3	2.00011229515075683594	0.00011229515075683594	1.12295150756835937500
4	2.00010609626770019531	0.00010609626770019531	1.06096267700195312500
5	2.00010299682617187500	0.00010299682617187500	1.0299682617187500000
6	2.00010156631469726562	0.00010156631469726562	1.0156631469726562500
7	2.00010061264038085938	0.00010061264038085938	1.0061264038085937500
8	2.00010037422180175781	0.00010037422180175781	1.00374221801757812500
9	2.00010013580322265625	0.00010013580322265625	1.00135803222656250000
10	2.00009989738464355469	0.00009989738464355469	0.99897384643554687500
11	2.0000$\boxed{9989738464355469}$	0.0000$\boxed{9989738464355469}$	0.$\boxed{99897384643554687}$500

Test 3

Заметим, что операция вычитания **в данном** случае определена корректно, т.е. без потерь в точности.
Аналогично с делением на 10^{-4} (умножением на 10^{4}), т.е. относительная погрешность не изменилась.
Значит значительные отклонения создало сложение.
Действительно результаты подтверждают данную гипотезу.

Найдем погрешность суммирования.
Воспользуемся алгоритмом Д. Шевчука по нахождению погрешности сложения чисел с плавающей точкой:

$$x \Leftarrow a \oplus b$$
$$b_{\text{virtual}} \Leftarrow x \ominus a$$
$$a_{\text{virtual}} \Leftarrow x \ominus b_{\text{virtual}}$$
$$b_{\text{roundoff}} \Leftarrow b \ominus b_{\text{virtual}}$$
$$a_{\text{roundoff}} \Leftarrow a \ominus a_{\text{virtual}}$$
$$y \Leftarrow a_{\text{roundoff}} \oplus b_{\text{roundoff}}$$
$$\textbf{return}\ (x, y)$$

Выведем погрешность, $delta$ и сравним их:

count	error	delta	equal
0	0.00000010261283023283	0.00009999999747378752	0
1	0.00000006790287443437	0.00004999999873689376	0
2	0.00000003395143721718	0.00002499999936844688	0
3	0.00000010223357094219	0.00001249999968422344	0
4	0.00000005111678547109	0.00000624999984211172	0
5	0.00000002555839273555	0.00000312499992105586	0
6	0.00000010643009318301	0.00000156249996052793	0
7	0.00000006599424295928	0.00000078124998026396	0
8	0.00000008621216807114	0.00000039062499013198	0
9	0.00000004310608403557	0.00000019531249506599	0
10	0.00000009765624753300	0.00000009765624753300	1
11	0.00000004882812376650	0.00000004882812376650	1
12	0.00000002441406188325	0.00000002441406188325	1
13	0.00000001220703094162	0.00000001220703094162	1
14	0.00000000610351547081	0.00000000610351547081	1
15	0.00000000305175773541	0.00000000305175773541	1

Test 4

Начиная с $count$ 10 погрешность совпадает с одним из слагаемых, именно поэтому мы наблюдали одинаковые значения x_{flat} и y_{float} после значения 10.

Заметим, что начиная с $count$ 10 погрешность и $delta$ **с точностью** равны.
Т.к. $equal = (error == delta)$ **(?)**
Получается $delta$ вносит изменения в результат суммы только своим присутствием,
а не своим значением, потому что оно не входит в погрешность. **(?)**

Таким образом погрешность возникает при суммировании. Она настолько значительна, что даже если функции $sqrtf, powf, fabsf$ не имеют погрешности, конечная погрешность выше допустимой.
Так что в данном случае мы можем пренебречь погрешностью этих функций, т.к. они не вызывают существенного отклонения.

Из всего вышесказанного следует появления бесконечного цикла в (1).

Почему в (2) не возникает таких отклонений?

На самам деле возникают. Если подобрать подходящую допустимую погрешность, то эксперимент с $double$ тоже выведет в бесконечный цикл.

Модифицируем (2) так, чтобы происходило уменьшение допустимой ошибки до т.е. пор пока не случится бесконечный цикл (в данном случае переполнение).
И будем выводить допустимую погрешность и счетчик на котором закончилось уменьшение $delta$ ($exit\ count$).

Переполнение int

```
exit count      allowable err

21               0.00000100000000000000
24               0.00000010000000000000
28               0.00000001000000000000
31               0.00000000100000000000
34               0.00000000010000000000
38               0.00000000001000000000
-2147483648              0.00000000000100000000
```

Test 5

Видим, что при допустимой погрешности 10^{-12} происходит переполнение int и цикл останавливается.

Выводы:

- Вычисления с $float$ и с $double$ не являются абсолютно точными $=>$ найдется конечная погрешность, т.ч. она будет меньше погрешности вычислений.
- Для $float$ эта погрешность будет выше чем для $double$, за счет меньшего объема памяти выделяемого для хранения числа.

Открытый вопросы:

- Почему появляется погрешность (~0.1%) после суммирования?

Компилятор в примерах:
gcc (Debian 8.3.0-6) 8.3.0
Настройки по умолчанию

Для более глубокого изучения:

- Floating-Point Computation by Pat Sterbenz
- What Every Computer Scientist Should Know About Floating-Point Arithmetic
- Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates