Numerical Methods – Exercise 2

CASE 1) Float precision

Taylor series („*brute force*“)

x	exp(-x)	Series	Number of terms in series
0.0	0.100000E+01	0.100000E+01	1
10.0	0.453999E-04	0.49762736E-03	38
20.0	0.206115E-08	0.36236930E+07	29
30.0	0.935762E-13	-0.29545972E+12	26
40.0	0.424835E-17	-0.17180824E+15	24
50.0	0.192875E-21	-0.65447925E+16	22
60.0	0.875651E-26	0.11235196E+18	21
		NaN	40
70.0	0.397545E-30	-0.73521937E+18	20
		NaN	40
80.0	0.180845E-34	-0.95546505E+19	20
		NaN	40
90.0	0.819401E-39	0.19505750E+20	19
		NaN	40
100.0	0.372008E-43	0.13219276E+21	19
		NaN	40

Table 1. Results of calculating exp(-x) using Taylor expansion (in float precision)

When calculating EXP(-x) using its Taylor series, the main problems are high terms and alternating sign in sums.

Let's start with high terms, which are given with $(-x)^n/n!$, and overflow that occurs.

In float precision, the highest possible factorial that can be calculated is (34)!, for anything higher than that, the result is 1.#INF (on Windows system), we have overflow. The same goes for power function *pow*. Table 2. gives highest possible values of power function float precision can handle for given x. So for example, computer would give the result of 10^{40} as 1.#INF, we again have overflow. So, in case we have overflow in either factorial or power function, we can have three possible results. First is overflow in factorial, but not in power, for example x = 10, n = 36. The result would be something finite divided by infinity, which is zero, that term wouldn't contribute to the sum. Second is overflow in power, but not in factorial, resulting in infinity, so the whole sum would be infinity. Lastly, the third would be overflow in both the power and factorial, giving the result for the term as 1.#IND on Windows, or NaN on Linux. If one term in sum is NaN, the whole sum is NaN.

x	**Highest possible integer power before overflow in *pow* function**
10	38
20	29
30	26
40	24
50	22
60	21
70	20
80	20
90	19
100	19

Table 2. Highest possible integer power before overflow in *pow* function (float precision)

Now we know the highest possible number of terms we can use before getting an error.

Let's look at the result themselves. For *x* smaller than 10, the results are somewhat consistent. For *x* = 10, the result is wrong, but at least its around the same order of magnitude as the correct term. That cannot be said for *x* larger than 10. We see from Table 1. that the results are large and getting larger with increasing *x*, when they should be getting smaller. For smaller terms in series, the power is greater than factorial. At one point factorial surpasses exponential growth, but we cannot see that because of overflow in factorial function (factorial is proportional to $(n/e)^n$, so for example *x* = 20, *n* would have to be about *n* = 50, which is higher than computer can handle). So for all the terms we can calculate, we have large numbers with alternating sign, This yields numerical errors seen in Table 1., and is the reason why numerical result is so high.

x	$x^n/n!$	n
20	2.163237E+007	25
	1.664029E+007	26
	1.232614E+007	27
	8.804384E+006	28
	6.071990E+006	29

Table 3. Some higher terms in sum for x = 20 (float precision)

As we see, the terms are becoming smaller, but now fast enough. In order to get correct result we should sum up all the terms up to a term that is very close to zero. But since we can't sum up to the close-to-zero term, we aren't suming up all the relevent terms, thus resulting in numerical errors.

Another error we get is roundoff error, since we are subtracting large numbers. When we subtract two large numbers, not all relevent digits are retained. In our sum we have a lot of subtracting of large numbers, leading to big errors, as opposed to smaller errors from subtracting just two numbers.

Lastly, we also have loss of precision. Some high terms in series can be to the powers greater then first few terms (for example for x = 20, term n = 20 is order of magnitude 1E+07, while term n = 3 is order of magnitude 1E+03). These lower terms can sometimes get neglected by the computer, which also contributes to numerical error of our calculation.

Recurrence relation

The main problem we had in our brute-force approach was overflow. This method gets rid of that, resulting in better results. The way it does that is that it deals with products. Rather than dividing giant numbers like factorials and powers and than summing it up, this method takes the previous term, *n - 1*, which isn't nearly as large as factorials or powers, and multiplies it with a *x/n*, thus getting the next term in series. Also, instead of getting larger with increasing *n,* like factorial and power, multiplier *x/n* gets smaller, avoiding overflow. So we can use more terms in series than before.

x	exp(-x)	Series	Number of terms in series
0.0	0.100000E+01	0.100000E+01	1
10.0	0.453999E-04	-0.52342311E-04	44
20.0	0.206115E-08	-0.17970337E+01	72
30.0	0.935762E-13	-0.72959523E+05	100
40.0	0.424835E-17	-0.85724339E+09	127
50.0	0.192875E-21	-0.52854849E+14	155
60.0	0.875651E-26	-0.74734699E+18	182
70.0	0.397545E-30	-0.14865830E+23	209
80.0	0.180845E-34	0.16145024E+27	231
90.0	0.819401E-39	-0.50603102E+31	241
100.0	0.372008E-43	NaN	246

Table 4. Results of exp(-x) using recurrence relation (float precision)

The next term in series will be larger than previous up to the point where *n* > *x*, after that we have multiplying previous terms with something smaller than 1, resulting in smaller terms. In case of *x* = 100, *n* would have to be at least 100. However, the previous terms get so large, that we get overflow before *n* can get that big. So overflow was not exposed off entirely. In case of *x* smaller than 100, we don't get overflow, for *x* larger than that, overflow occurs.

With this method we got rid of overflow and we can sum up many more terms, so we can sum up all the relevant terms. However, errors like roundoff error and loss of precision are still present, since we are still dealing with alternating, large numbers. The results are more precise, but for large *x,* they are still incorrect. So the method is better, but it can be improved even more with this next algorithm.

Inverse of exp(x)

This method consists of finding exp(x) using recurrence relation of its Taylor series, and then calulating 1/exp(x). It gives the best results out of three methods.

x	exp(-x)	Series	Number of terms in series
0.0	0.100000E+01	0.100000E+01	1
10.0	0.453999E-04	0.453999E-04	44
20.0	0.206115E-08	0.206115E-08	72
30.0	0.935762E-13	0.935762E-13	100
40.0	0.424835E-17	0.424835E-17	127
50.0	0.192875E-21	0.192875E-21	155
60.0	0.875651E-26	0.875651E-26	182
70.0	0.397545E-30	0.397545E-30	209
80.0	0.180845E-34	0.180850E-34	237
90.0	0.819401E-39	0.0E+0	264
100.0	0.372008E-43	0.0E+0	291

Table 5. Results of exp(-x) using inverse method (float precision)

This method, like the recurrence one, doesn't have a factorial function, so we can get higher terms. Also, here we sum only positive numbers, so loss of precision and roundoff error are greatly reduced. Once we get exp(x), which can be a very large number, we just calculate 1/exp(x). The results are correct for small enough *x*, in this case for *x* smaller than 80. For *x* larger than that, exp(x) is too large of a number to be stored in float, the results is 1.#INF, and 1/1.#INF results in zero, which is still a much better result than large numbers that we get in first two algorithms.

CASE 2) Double precision

Taylor series („*brute force*“)

The main difference between CASE 1 and CASE 2 is that in CASE 2, double precision, we can calculate greater span of numbers, larger and smaller numbers than float, meaning we get better results, and are less susceptible to overflow. In case of Taylor series expansion, this means we can calculate factorial up to 170!, which is significantly higher than its float equivalent. The errors that occur are the same as already described in float case, the difference is that those errors are smaller in case of double. Since double can have more decimal places, the roundoff error is reduced (but it is still there). Same goes with loss of precision.

x	exp(-x)	Series	Number of terms in series
1.0	0.100000E+01	0.100000E+01	1
10.0	0.453999E-04	0.453999E-04	50
20.0	0.206115E-08	0.347871E-06	65
30.0	0.935762E-13	-0.341977E-04	95
40.0	0.424835E-17	-0.221029E+01	115
50.0	0.192875E-21	-0.833851E+05	140
60.0	0.875651E-26	-0.850381E+09	170
70.0	0.397545E-30	1.#INF	169
		NaN	170
80.0	0.180845E-34	1.#INF	163
		NaN	170
90.0	0.819401E-39	NaN	170
100.0	0.372008E-43	NaN	170

Table 6. Results of calculating exp(-x) using Taylor expansion (double precision)

Everything described in float case still stands here, all the errors are the same, the difference is that double is more precise so the errors are smaller and less visible, and we can use more terms in series.

The reason that the sum for *x* = 70 is infinite for *n* = 169 is that pow(70, 169) is greater than largest accepteble power, as seen in table below, resulting in overflow. Same goes for *x* = 80. In order to get a finite sum, we would have to use fewer terms. However, results would still be wrong, as seen for *x* = 40, 50, 60…, so it doesn't matter that much.

x	Highest possible integer power before overflow in *pow* function
10	308
20	236
30	208
40	192
50	181
60	173
70	167
80	161
90	157
100	154

Table 7. Highest possible power before overflow (double precision)

Recurrence relation

x	exp(-x)	Series	Number of terms in series
0.0	0.100000E+01	0.100000E+01	1
10.0	0.453999E-04	4.539990E-005	44
20.0	0.206115E-08	6.087065E-009	72
30.0	0.935762E-13	6.103000E-006	100
40.0	0.424835E-17	3.116952E-001	127
50.0	0.192875E-21	2.041833E+003	155
60.0	0.875651E-26	7.227457E+008	182
70.0	0.397545E-30	4.594081E+012	209
80.0	0.180845E-34	2.450820E+017	237
90.0	0.819401E-39	-5.865799E+021	264
100.0	0.372008E-43	8.144653E+025	291

Table 8. Results of exp(-x) using recurrence relation (double precision)

As in CASE 1, recurrence got rid of overflow, even in case of *x* = 100, which float had trouble with. However, results are still wrong because of roundoff error and loss of precision.

Inverse of exp(x)

x	exp(-x)	Series	Number of terms in series
0.0	0.100000E+01	0.100000E+01	1
10.0	0.453999E-04	0.453999E-04	44
20.0	0.206115E-08	0.206115E-08	72
30.0	0.935762E-13	0.935762E-13	100
40.0	0.424835E-17	0.424835E-17	127
50.0	0.192875E-21	0.192875E-21	155
60.0	0.875651E-26	0.875651E-26	182
70.0	0.397545E-30	0.397545E-30	209
80.0	0.180845E-34	0.180845E-34	237
90.0	0.819401E-39	0.819401E-39	264
100.0	0.372008E-43	0.372008E-43	291

Table 9. Results of exp(-x) using inverse method (double precision)

As we see, with double precision and inverse method, we are able to get all the correct results for all the given *x*.

In conclusion, the best algorithm is the third one, with inverse of exp(x), since it doesn't sum alternating large numbers and uses recurrence relation as to avoid overflow.

As for data type, double yields far better results. So in case computing time and storage isn't a problem, when dealing with real numbers, its always better to use double precision.