Numerical Methods: Assignment #1

Hong Chenhui
drredthered@gmail.com

Zhejiang University — March 11, 2024

Info: In addition to the issue that the significant number might not precisely give the ideal one because of overly tight Scarborough methods, it is important to remember that the inaccuracy may vary because of various integration techniques. One may only regards the results as a rough estimate rather than the actual value.

1 Problem

- **Question 1:** The taylor series of $cosx$ is described as $\sum_{i=0}^{\infty} \frac{(-1)^n x^{2i}}{(2i)!}$.
  - With four significant digits, approximate its value using single-precision and double-precision variables.
  - Explain the consequence while attempting to approximate with floating point relative accuracy using single-precision variables.
- **Question 2:** Calculate the following integral using reduction formulae. Each item should have its true error less than 10^{-2}.

$$E_n = \int_0^1 x^n e^{-x} dx, (n \in \mathbb{N})$$

1.1 Theoretical viewpoint

Question 1

Nothing to discuss one the first project. Noting that Scarborough methods may give strictest significant number, which could be modified. As such, $\frac{\epsilon_s}{(0.5\times 10^{2-n})\%} < 1$. Machine epsilon for single-precision are $\epsilon_{ps} = 2^{-23}$ according to widespread variant definition. Click here for more details.

(a) Specify ϵ_s.

(b) Iterate.

(c) Output the results.

Question 2

One would easily find its recurrence relation using basic calculus as follows:

$$\begin{aligned} E_n &= \int_0^1 x^n e^{-x} dx \\ &= -x^n e^{-x}\Big|_0^1 + n\int_0^1 x^{n-1} e^{-x} dx \\ &= -e^{-1} + nE_{n-1} \end{aligned} \tag{1}$$

with a terminal $E_0 = \int_0^1 e^{-x} dx = -e^{-x}\Big|_0^1 = 1 - \frac{1}{e}$. However, truncation error must occurs since $1 - \frac{1}{e}$ is irrational. Hence the actual formula looks like this,

$$\begin{cases} \tilde{E_n} = -e^{-1} + n\tilde{E_{n-1}} & (2) \\ \tilde{E_0} \approx 1 - \dfrac{1}{e} & (3) \end{cases}$$

which is terrible. The true error would appear at least as $|E_n - \tilde{E_n}| = |n!(\tilde{E_0} - E_0)|$, with $(\tilde{E_0} - E_0)$ roughly $\frac{1}{2}\epsilon_{ps}$,making this method extremely unstable. We would discuss this in the analysis section, and for now, we would only imply the algorithm by such:

(a) calculate E_0.

(b) Iterate using recurrence equation.

(c) Do $\epsilon_{anew} = \frac{1}{2}\epsilon_{ps} + n\epsilon_a$.

(d) Continue iterate using Eq.(1) until ϵ_a becomes bigger than 10^{-2}.

2 Implementation

$MATLABR2023a$ codes seen at the codes section as well as attached files.

1. The results are as follows:

	Single-precision	Double-precision
True value	0.5403023	0.5403023058681398
Approximation	0.5403026	0.540302579365079
Relative error%	0.0045892	0.004590314436534
Iteration times	5	5

Machine epsilon one as follows:

	Single-precision
True value	0.5403023
Approximation	0.5403023
Relative error%	0
Iteration times	7

2. The results are as follows. It iterated 16 times before reaching the desired accuracy limits.

E	results	E	results	E	results
0	0.632120558828558	6	0.059933627487352	12	0.030463418970410
1	0.264241117657115	7	0.051655951240024	13	0.028145005443888
2	0.160602794142788	8	0.045368168748749	14	0.026150635042994
3	0.113928941256923	9	0.040434077567295	15	0.024380084473469
4	0.087836323856248	10	0.036461334501509	16	0.022201910404061
5	0.071302178109799	11	0.033195238345154	ϵ_a	0.067853638742031

3 Analysis

3.1 Relative Error Issues at Single-precision

It is evident that the relative error ϵ_a reached 0 with single-precision mechanics, matching the actual error value ϵ_t, which, not surprisingly, is below $\frac{1}{2}\epsilon_s$. This is because any smaller number below machine epsilon cannot be distinguished by a binary machine, or at least one would assume this is the explanation. If it holds true, we could be more confident that **any convergent iteration algorithm would eventually give results yielding the actual value in corresponding precision.**

3.2 Improvements on the Reduction Formulae

As we've already observed, the algorithm from the previous part only managed to fulfill our objective up until E_{17}. The cause of which has previously been covered by the following formulae.

$$\begin{cases} \tilde{E_n} = -e^{-1} + n\tilde{E_{n-1}} & (4) \\ \tilde{E_0} \approx 1 - \dfrac{1}{e} & (5) \end{cases}$$

The reason behind such unstableness is the truncation error, which is $|E_n - \tilde{E_n}| = |n!(\tilde{E_0} - E_0)|$. This is a direct consequence of the fact that $1 - \frac{1}{e}$ is irrational, and the error would grow factorially with the increase of n.

One way to reverse this is to do a minor modification the equation. Instead of recurrent from the first term, we could start from the last, hence the following formulae.

$$\begin{cases} \tilde{E_{n-1}} = \dfrac{1}{n}(e^{-1} + \tilde{E_n}) & (6) \\ \tilde{E_n} \approx \displaystyle\int_0^1 x^n e^{-x} dx & (7) \end{cases}$$

E_n could be calculated through any integration methods, such as **The Newton-Cotes Algorithm** or **Romberg Integration**.

This might seem pointless at first glance, until one realizes that the error would now shunk factorially with the increase of n, which is a significant improvement over the previous method. If desired item number is below n, one could simply use the recurrence relation to calculate the desired value, rather than using the **The Newton-Cotes Algorithm** or **Romberg Integration** again, which is very slow. Below portraits the true error of both methods. We could clearly see that reverse sequence method is much more stable than the positive one.

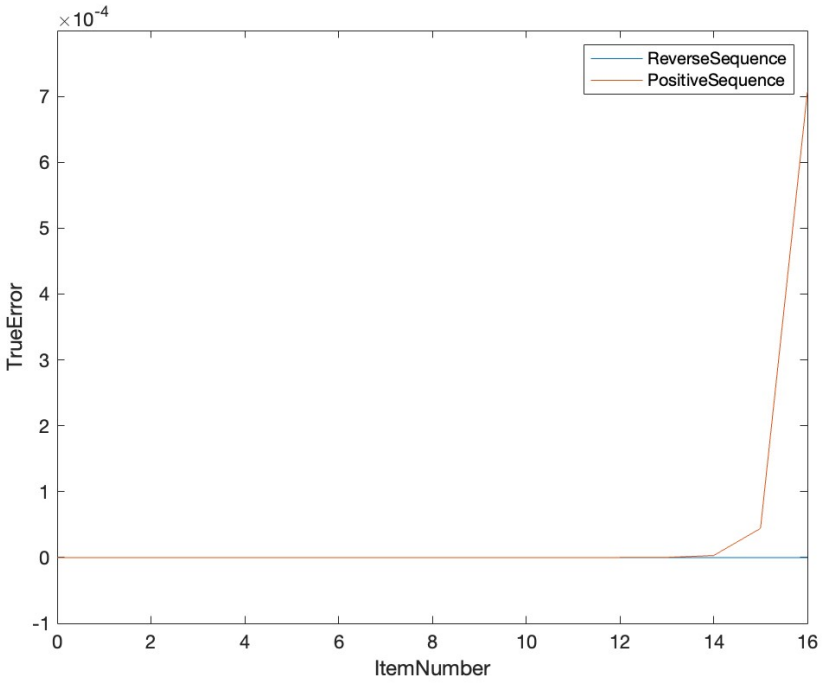

Fig.1: Comparison of the two methods.

4 Codes

Matlab Codes are attached below and in the file. Also at my private blog: drredthered.github.io.

IterCos.m

```
function [v,ea,iter] = IterCos(x,es)
% initialization
iter = 0;
sol = 0;
ea = 100;

iter = single(iter);
sol = single(sol);
ea = single(ea);
es = single(es);

% iterative calculation
while (1)
 solold = sol;
 sol = sol + (x ^ (2*iter)) * ((-1)^(iter))
 /factorial(2*iter);
 iter = iter + 1;
 if sol~=0
 ea=abs((sol - solold)/sol)*100;
 end
 if ea<=es,break,end
end
v = sol;
end
```

IterCosDouble.m

```
function [v,ea,iter] = IterCosDouble(x,es)
% initialization
iter = 0;
sol = 0;
ea = 100;

% iterative calculation
while (1)
 solold = sol;
 sol = sol + (x ^ (2*iter)) * ((-1)^(iter))
/factorial(2*iter);
 iter = iter + 1;
 if sol~=0
 ea=abs((sol - solold)/sol)*100;
 end
 if ea<=es,break,end
end
v = sol;
end
```

Command Line

```
>>[v,ea,iter] = IterCos(1,0.5e-2)
>>[v,ea,iter] = IterCosDouble(1,0.5e-2)
>>[v,ea,iter] = IterCos(1,eps) //Operation methods have been modified
```

IterE.m

```
function [v,ea,iter] = IterE(es)
% initialization
iter = 0;
sol(1) = 1 - 1/exp(1);
ea = 0.5 * eps;

% iterative calculation
while (1)
 sol(iter+2) = -1/exp(1) + (iter+1)*(sol(iter+1));
 iter = iter + 1;
 ea = (iter+1)*ea + 0.5 * eps;
 if ea>=es,break,end
end
v = sol;
end
```

Command Line

```
>>[v,ea,iter] = IterE(0.01)
```

IterRevE.m

```
function [v,iter] = IterERev(n)
% initialization
iter = n + 1;

fun = @(x) x.^(n).*(exp(1).^(-x));
sol(iter) = integral(@(x) fun(x),0,1);

% iterative calculation
while ( iter - 1 > 0 )
 sol(iter - 1) = ((1.0/exp(1)) + sol(iter))
/ (iter - 1) ;
 iter = iter - 1;
end
v = sol;
end
```

Command Line

```
>>[v,ea,iter] = IterRevE(16)
```