Internship Report

Aanjishnu Bhattacharyya

April 3, 2025

Contents

1 Introduction

Numerical Analysis is an very important part of mathematics. It provides us with the tools nessesary to tackle real-life math problems which are hard to solve analytically. Functions such as the gamma functions and the normal distribution are extremely difficult to calculate analytically and sometimes impossible. Improper Integrals and integrals of higher dimentional functions may also be solved in a efficient manner utilizing these methods.

We have tried to build a simple yet powerful software utility which is able to integrate functions with vector valued inputs and vector valued outputs. It is possible to have multiple nested integrals with varying limits including improper integrals. Simpsons and Trapezoidal methods are used since they provide a good compromise speed and correctness. The software utility also provids a rudimentary 3-d visualization framework to plot relevant functions utilizing 3d-acceleration hardware present in most modern computers. The software utility also utilizes the feature of runtime link libraries provided by most modern operating system to provide a seameless user experience.

2 Learning Phase

In the learning phase we first develop the mathematical background required for this task.

2.1 Numerical Method of calculating Integrals

To compute the integrals of any function, we must first consider the problem of approximating the function. Approximation in this context reffers to the reconstruction of a function from n sampled points obtained from the function. This problem becomes much simpler if the sample points are spaced equally apart from eachother. Fortunately in our particular case we have direct access the the function itself thus it is trivial to obtain such points. Simpsons method and Trapezoidal method perform the best under these given constraints.

Let, $f : \mathbb{R} \to \mathbb{R}$

$f_i = f(x_i) \qquad \forall x_i \in \mathbb{R} \quad \wedge \quad 0 \leq i \leq n$

Let, $L : \mathbb{R} \to \mathbb{R}$

$$L(x) = \sum_{i=0}^{n} \omega_i^n(x) f_i$$

$$\omega_i^n(x) = \prod_{\substack{j=0 \\ i \neq j}}^{n} \frac{x - x_j}{x_i - x_j} \qquad i \neq j$$

Now, $$I = \int_a^b f(x)\mathrm{d}x \qquad a = \min\{x_i\} \wedge b = \min\{x_i\}$$

$$= \int_a^b L(x)\mathrm{d}x$$

$$= \int_a^b \sum_{i=0}^{n} \omega_i^n(x) f_i \mathrm{d}x$$

$$= \sum_{i=0}^{n} \int_a^b \omega_i^n(x) f_i \mathrm{d}x$$

$$= \sum_{i=0}^{n} f_i \int_a^b \prod_{\substack{j=0 \\ i \neq j}}^{n} \frac{x - x_j}{x_i - x_j} \mathrm{d}x$$

Let, $h = x_i - x_{i-1} \qquad \because x_i$ is equispaced

$$u = \frac{x - x_0}{h}$$

$$\mathrm{d}u = \frac{1}{h}\mathrm{d}x$$

Substituting u,

$$= \sum_{i=0}^{n} f_i \int_0^n h \prod_{\substack{j=0 \\ i \neq j}}^{n} \frac{u - j}{i - j} \mathrm{d}u$$

$$= \sum_{i=0}^{n} f_i \frac{h \cdot -1^{n-i}}{i! \cdot (n-i)!} \int_0^n \prod_{\substack{j=0 \\ i \neq j}}^{n} (u - j)\mathrm{d}u \tag{1}$$

From this general formulae we can derive equations for both Simpsons ($n = 2$) and Trapezoidal ($n = 1$) rules,

From (1),

$$I = \sum_{i=0}^{n} f_i \frac{h \cdot -1^{n-i}}{i! \cdot (n-i)!} \int_0^n \prod_{\substack{j=0 \\ i \neq j}}^{n} (u-j)\mathrm{d}u$$

For n = 1,

$$= \sum_{i=0}^{1} f_i \frac{h \cdot -1^{1-i}}{i! \cdot (1-i)!} \int_0^1 \prod_{\substack{j=0 \\ i \neq j}}^{1} (u-j)\mathrm{d}u$$

$$= -f_0 \cdot h \cdot \int_0^1 (u-1)\mathrm{d}u + f_1 \cdot h \cdot \int_0^1 u\mathrm{d}u$$
$$= -f_0 \cdot h \cdot \left[\frac{u^2}{2} - u\right]_0^1 + f_1 \cdot h \cdot \left[\frac{u^2}{2}\right]_0^1$$
$$= (f_0 + f_1)\frac{h}{2}$$

For n = 2,

$$= \sum_{i=0}^{2} f_i \frac{h \cdot -1^{2-i}}{i! \cdot (2-i)!} \int_0^2 \prod_{\substack{j=0 \\ i \neq j}}^{2} (u-j)\mathrm{d}u$$

$$= f_0 \cdot h \cdot \int_0^1 (u-1)(u-2)\mathrm{d}u - f_1 \cdot h \cdot \int_0^1 u(u-2)\mathrm{d}u + f_2 \cdot h \cdot \int_0^1 u(u-1)\mathrm{d}u$$
$$= (f_0 + 4f_1 + f_2)\frac{h}{3}$$

2.2 Integrals with improper limits

Integrals with improper limits can often be broken down into subpieces which can be computed separately.

e.g,

$$\int_{-1}^{1} \frac{1}{x}\mathrm{d}x = \lim_{z \to 0} \left(\int_{-1}^{z} \frac{1}{x}\mathrm{d}x + \int_{z}^{1} \frac{1}{x}\mathrm{d}x \right)$$

We can also have limits which have infinities as their limits, in these cases we can construct a series with the integrals themselves and try to understand if the series converges. If it does we can also compute for what values.

e.g,

$$\int_0^\infty e^{-x^2}\mathrm{d}x$$

Let us consider the series,

$$S_n = \int_0^n e^{-x^2}\mathrm{d}x$$

We can say S_k converges if at a large enough value of k equals S_{k+1} given a particular precision.

$$S_{n+1} - S_n < |\epsilon| \qquad n \geq k \in \mathbb{R}$$

2.3 Computing Multiple Integrals

Multiple integrals to compute the volumes and other higher dimentional constructs can be computed through the same methods. Computing the limits from the outer most integral and then computing limits for these integrals from the inner most integral. How ever on of the many contrains of this system requires that each limit be defined as a function instead of conditional constraint, and it may be nesessary to convert conditional constraints into limits

e.g,

$$\iint\limits_{x^2+y^2\leq r^2} f(x,y)\mathrm{d}A$$
$$= \int_{-r}^{r}\left(\int_{-\sqrt{r^2-y^2}}^{\sqrt{r^2-y^2}} f(x,y)\mathrm{d}x\right)\mathrm{d}y$$

3 Training

3.1 Building an simple prototype of a function integrator and plotter.

We build a simple prototype[**?**] in javascript to understand how we should aproach building the integrator software[1]. During this process we avoid using several built in features and external libraries of javascript as to undestand the internals of such a system and also to keep the program as light weight as possible.

The prototype was capable of integrating any function whose domain was $\mathbb{R}^2$ and range was $\mathbb{R}$. The interface developed in this prototype resembled that of graphing software desmos. The major things that we learned from building this prototype were:

- The computation time increased rapidly with decrease in the magnitude of h so it was imperative that we language which does not have a very high overhead.
- We need to separate out the graphing part of the program from the computation part to make it effective and fast.
- We must allow the user to input data in a known syntax, and allow them to edit the functions at runtime.

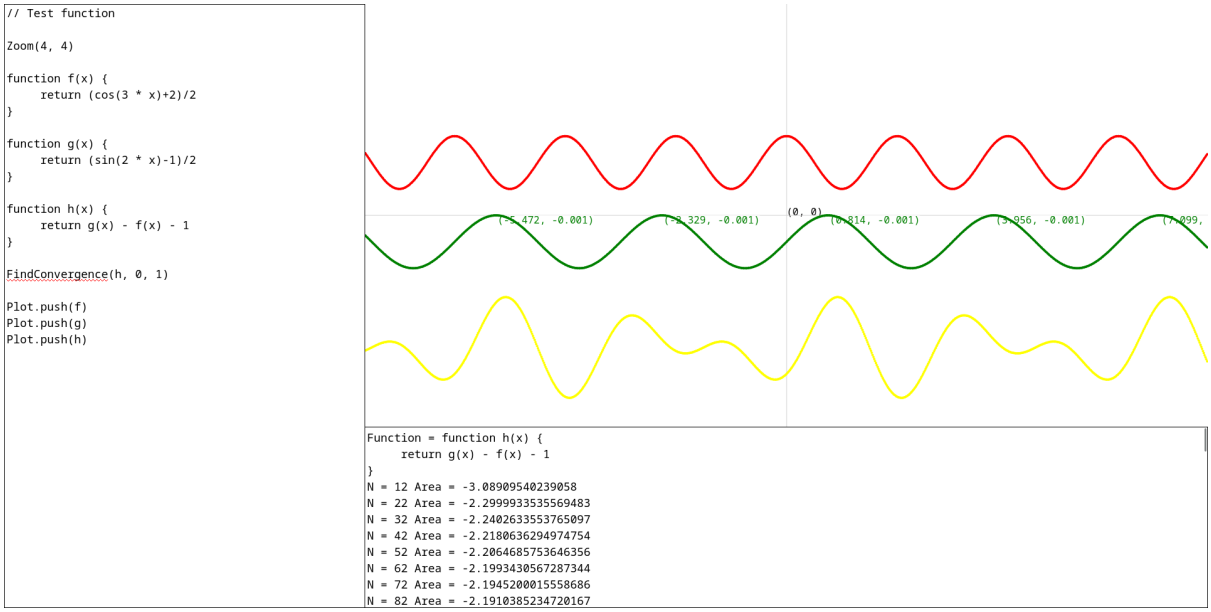


Figure 1: Integrating and plotting a trigonometric function

4 Project

We describe the different parts of the software that we built to implement these numerical methods.

4.1 Algorithmic approach to numerically solving integrals

Computing a single integrals is relatively simple. Iteratively we apply trapezoidal or simpson's rule over multiple samples taken on the function.

Algorithm 1 Computing an integral of the function f.

procedure TRAPEZOIDAL(f_0, f_1) ▷ $f_i = f(x_i)$
  return $(f_0 + f_1)/2$
end procedure

procedure INTEGRATE(s: start, e: end, f: function, n: samples)
  $h \leftarrow (e - s)/n$
  $a \leftarrow 0$ ▷ area under the curve
  for $x \leftarrow s$ **to** $e - h$ **step** h **do**
    $a \leftarrow a + h \cdot Trapezoidal(f(x), f(x + h))$
  end for
  return a
end procedure

To extend this integral to multiple variables we perform integrals on each axis at a time. Implementation of this method algorithmically requires us to treat inner integrals as functions which we can integrate normally.

$$g(x) = \int_a^b f(x, y)dy$$

This means that each inner integrals has access the current state of the variables of the outer integrals. We developed the idea of a 'context' which holds values of all previously occuring integrals. In this algorithm we express this with the function $f^*(x)$ which essentially means that this function not only operates on variable x but also has access to all other previous variables (now treated as constants).

$$f_i^*(x) = f(x_0, x_1, ..., x_{i-1}, x)$$

Describing this as a algorithm turns the functions assuming each integrals described from outermost (0_{th}) to innermost (n_{th}) and the these integrals operate in order variables from x_0 to x_n. This does not need to be case and indeed in our implementation we avoid this positional lockin by utilizing associative arrays for the arguments of the function instead of positional arguments. However that does not affect the way the algorithm oprates,

Algorithm 2 Computing multiple integrals of the function f.

1: **procedure** MULTI-INTEGRAL($l \equiv [(s, e)], f, d, c$)
   l are the limits, f is the function to be integrated
   d depth of recursion, c reffers to the variable contexts
2: **if** $|l| = 0$ **then** ▷ We do not have any more integrals
3: **return** $f_n^*(x_n)$ ▷ x_i are accessed through the 'context' variable c
4: **else**
5: $h \leftarrow (l_d.e - l_d.s)/n$
6: $a \leftarrow 0$ ▷ area under the curve
7: **for** $x_d \leftarrow l_d.s$ **to** $l_d.e - h$ **step** h **do**
8: add_context_variable (c, x_d)
9: $r_0 \leftarrow$ Multi-Integral$(l, f, d + 1, c)$ ▷ inner integrals as functions
10: $r_1 \leftarrow$ Multi-Integral$(l, f, d + 1, c)$
11: remove_context_variable (c, x_d)
12: $a \leftarrow a + h \cdot (r_0 + r_1)/2$ ▷ Trapezoidal rule
13: **end for**
14: **return** a
15: **end if**
16: **end procedure**

4.2 Console Interface

The graphical user interface of the prototype was discarded in favour of a very simple console interface to improve portability of the software. To provide quality of life features we utilize a console input library Crossline[3]

A part of the user input also involved accepting a C source file which contained all the function expressions. This file is used to allow runtime expression changing which is further discussed in the later sections.

The users interact with internal integrator using simple commands like 'integrate' and 'plot'. Commands like 'list' and 'pointer' were added to aid with debugging perposes.

A special set of macros were developed to allow the 'list' function to work and to allow a simple way to return vector values.

4.3 Runtime expression change

A major problem with our integrator was the slow execution times of a interpreted language. Since each function had to be executed multiple times the cost of interpreting added up extremely quickly. We noticed this problem in the prototype, the website would sometimes stop responding due to the high integrator execution times.

To address this problem it was nessesary to compile the expression down to a native binaries which would run orders of magnitude faster. To achieve this we utilized the very small c99 complient 'tiny c compiler'[2]. We spawned a process with the tcc binaries and compiled the expression C file into a Dynamic link library (or Shared Object). The generated executable is a Position Independent (reffered to as PIE from here). This allowed use to load this executable at runtime and execute the native binary version of the expressions. An important factor which allowed this was that all the expressions did not have any side effects i.e. they did not affect memory of the main process. This meant that we did not have to deal with a situation where the structure of the code changed but the memory held in the process did not leading to fatal memory corruptions or undefined accesses of memory locations.

We also considered developing a simple scripting language, but that idea was found to be largly out of the scope of this project.

Some of the hurdles we faced while using the tcc compiler included the lack of support for complex data types which is standard for any c11 compilent compiler like gcc. We had to implement our own version of complex data type.

4.4 Multiple Integration

Functionally Multiple Integrals operate similarly to single integrals as demostrated in previous sections. The way we deal with this in our software is we have a C structure which represents any given integrals with its limits and the variable it is operating on. We can construct a linked list of such integrals resolving the limits of each integral from outer outer most to inner most. Then integrating the function from inner to to outer integrals. We also incorportate a special integral computation module which would allow us to compute integrals of common functions which are known not to behave well under simpsons and trapezoidal rule of integration.

4.5 Plotting of Functions

Each vector valued function is assumed to be of the form $f : \mathbb{R}^2 \to \mathbb{R}$ This assumption allows us to plot any given 2d or 1d function.

Values for multiple points are sampled from the function. A group of 3 adjescent points are selected a triangle is formed. In this manner a contiguous mesh of triangles is generated. each triangle is shaded on the basis of their normal values. The shading gives a sense of depth to the plot. Each point is recomputed each turn of the plot. To achieve higher resolution plot the number of points sampled is increased.

The triangles are displayed on the screen using the cross platform graphics library Raylib[4]. Raylib allows a streamlined interfacing with accelerated graphics hardware using the OpenGL API and a simple way to accept user keyboard input.

4.6 Special Functions

Some functions like the gamma function do not behave well with generalized integral computation methods, thus special methods have been developed to specifically deal with these functions. Currently the only function that has been implemented under this module is the Gamma function. We utilize the Lanczos Aproximation[1] for the gamma function to compute their values at a resonably high speed and acuracy.

4.7 Examples of Function Integration and Plotting

4.7.1 Integrating $sin(x)$ function from 0 to π^2

Expression C file to integrate sin function from 0 to π,

```
#include "function.h"

FUNCTION ( X )
vec_t function(vec_t v)
{
        RETURN_VEC ( sin(v.it[0]) );
}

FUNCTION ( )
vec_t limit_start(vec_t v)
{
        RETURN_VEC ( 0 );
}

FUNCTION ( )
vec_t limit_end(vec_t v)
{
        RETURN_VEC( PI );
}
```

Integrating on variable X from 0 to 3.141593.

Step	Variable	h	Start	End	Result
1	x	0.031416	0.000000	3.141593	1.999507
2	x	0.030800	0.000000	3.141593	1.999526
3	x	0.030208	0.000000	3.141593	1.999544
4	x	0.029638	0.000000	3.141593	1.999561
5	x	0.029089	0.000000	3.141593	2.000000
6	x	0.028560	0.000000	3.141593	2.000000

4.7.2 Integrating $sin(x) + sin(y)$ function[3]

Expression C file to integrate $sin(x) + sin(y)$ under the triangle with vertices $(0, 0)(1, 0)(1, 1)$

```
#include "function.h"

FUNCTION  ( X, Y )
vec_t function(vec_t v)
{
        RETURN_VEC ( sin(v.it[0]) + sin(v.it[1]) );
}

FUNCTION ( )
vec_t limit_start_outer (vec_t v)
{
        RETURN_VEC ( 0 );
}

FUNCTION ( )
vec_t limit_end_outer  (vec_t v)
{
        RETURN_VEC ( 1 );
}

FUNCTION ( )
vec_t limit_start_inner(vec_t v)
{
        RETURN_VEC ( 0 );
}

FUNCTION ( Y )
vec_t limit_end_inner  (vec_t v)
{
        RETURN_VEC ( v.it[0] );
}
```

Integrating on variable Y then on the inner variable X under the trianglular region with verticles $(0,0)(1,0)(1,1)$

Step	Variable	h	Start	End	Result
1	x	0.000000	0.000000	0.000167	0.000000
2	x	0.000000	0.000000	0.000333	0.000000
3	x	0.000000	0.000000	0.000500	0.000000
4	x	0.000001	0.000000	0.000667	0.000001
5	x	0.000001	0.000000	0.000833	0.000001
			...		
n+1	x	0.001000	0.000000	0.999500	1.300057
n+2	x	0.001000	0.000000	0.999667	1.302110
n+3	x	0.001000	0.000000	0.999833	1.302481
n+4	x	0.001000	0.000000	1.000000	1.301169
n+5	y	0.001000	0.000000	1.000000	0.459997

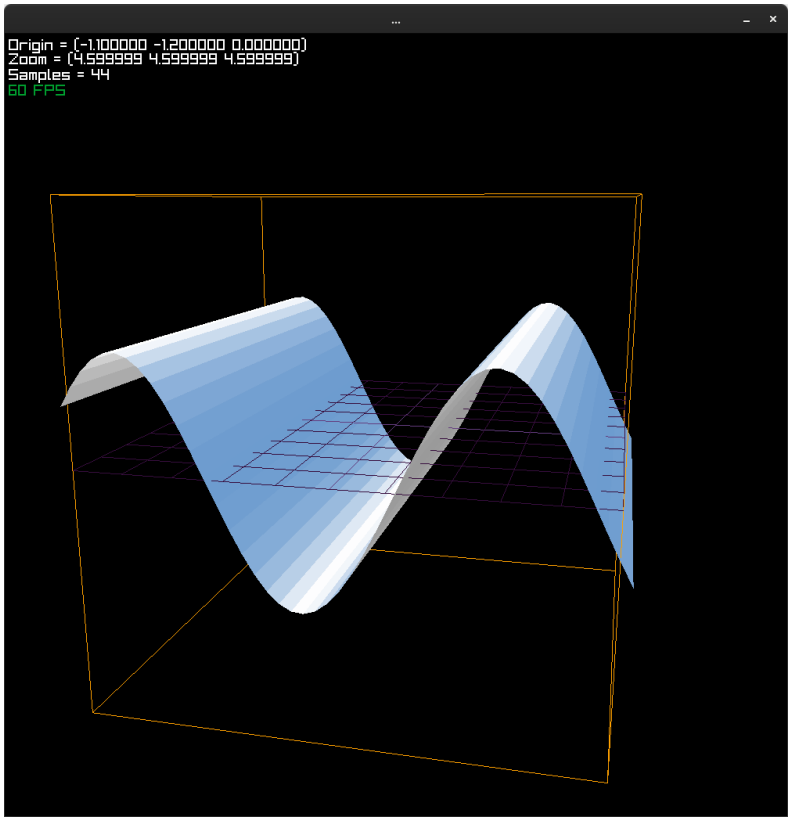


Figure 2: $sin(x)$

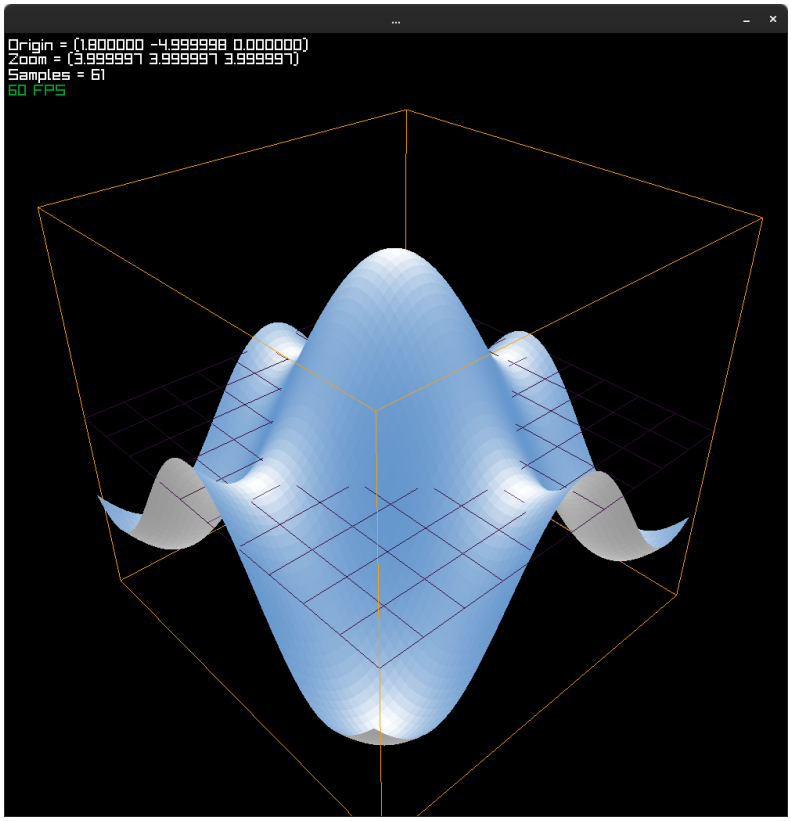


Figure 3: $sin(x) + sin(y)$

5 Results

5.1 Gamma function and methods of computing it numerically

The gamma function can be defined as,

$$\Gamma(z) = \frac{1}{z}\prod_{n=1}^{\infty}\left[\frac{1}{1+\frac{z}{n}}\left(1+\frac{1}{n}\right)^{z}\right]$$

$$= \int_0^{\infty} t^{z-1}e^{-t}dt$$

$$= z\Gamma(z-1)$$

The gamma function is an relatively difficult function to compute with these simple numerical methods from a complexity perspective. The major reasons for this being the case are:

- The gamma function is computed through an improper integrals.
- The function grows at the rate of $O(n!)$ which is very difficult to manage, and precompute.
- The function also has discontinuities at the $-ve$ integers

To understand the limitations of the program we try to compute and plot the values of the gamma function using various methods implemented under the plotting and runtime expression compilation tooling we have built.

In the following sections we describe algorithms that have been used to compute the gamma functions. We also measured the time for computation required to execute these algorithms when implemented under the constraints of the software we have built. We utilize the implementation of gamma function in pythons math libraries to estimate the precision of each method.

Algorithm 3 Iterated Product method of computing the gamma function

```
1: procedure GAMMA(z)
2:     g ← 1/z
3:     for n ← 1 to 10^7 do
4:         a ← 1/(1 + z/n)
5:         b ← (1 + 1/n)^z
6:         g ← a · b · g
7:     end for
8:     return g
9: end procedure
```

The Algorithm describing the integral solving method has been presented in section 4.1 The "Lanczos Implementation of the Gamma Function" has been described in the paper [1].

5.2 Comparisn of Compute times of each method of computing Gamma Function

We perform the gamma function a small sample set of input, and time that operation. This excercise gives us a good idea on the performace of each algorithm on real hardware. This important because the time complexity of each method is $O(n)$.

Compute times for the Gamma Function using the infinite product definition.

- Total Time of execution = 7.4433441162109375 seconds
- Time of execution for each input = 0.9304180145263672 seconds

Input	Output	Actual	Error
0.1	9.513508	9.513507698668732	3.0133126749376515e-07
0.2	4.590844	4.5908437119988035	2.8800119622474085e-07
0.3	2.991569	2.991568987687591	1.2312409314318984e-08
0.4	2.21816	2.2181595437576878	4.562423123743997e-07
2	1.0	1.0	0.0
3	1.999999	2.0	-9.999999999177334e-07
4	5.999996	6.0	-3.9999999996709334e-06
5	23.999976	24.0	-2.3999999999801958e-05

Compute times for the Gamma Function using the Euler integral definition.

- Total Time of execution = 9.676705837249756 seconds
- Time of execution for each input = 1.2095882296562195 seconds

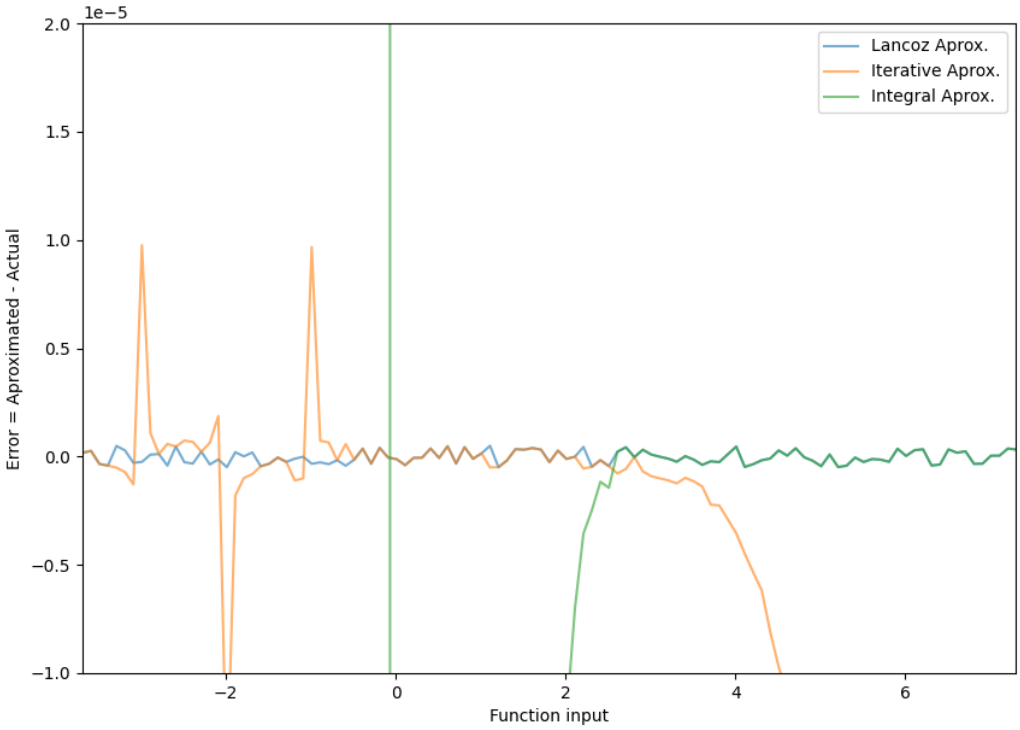
Input	Output	Actual	Error
0.1	5.625076	9.513507698668732	-3.8884316986687324
0.2	3.857274	4.5908437119988035	-0.7335697119988036
0.3	2.811185	2.991568987687591	-0.18038398768759079
0.4	2.167838	2.2181595437576878	-0.0503215437576876
2	0.999988	1.0	-1.2000000000012001e-05
3	2.0	2.0	0.0
4	6.0	6.0	0.0
5	24.0	24.0	0.0

Compute times for the Gamma Function using the Laconzs aproxmiation.

- Total Time of execution = 0.01519465446472168 seconds
- Time of execution for each input = 0.00189933180809021 seconds

Input	Output	Actual	Error
0.1	9.513508	9.513507698668732	3.0133126749376515e-07
0.2	4.590844	4.5908437119988035	2.8800119622474085e-07
0.3	2.991569	2.991568987687591	1.2312409314318984e-08
0.4	2.21816	2.2181595437576878	4.562423123743997e-07
2	1.0	1.0	0.0
3	2.0	2.0	0.0
4	6.0	6.0	0.0
5	24.0	24.0	0.0

5.3 Comparisn of Error of each method of computing Gamma Function



From this plot and the previous execution times recorded we can understand that the Lanczos aproximation performs best in all regards. The integral method performs well for posetive real numbers greater than 2 but fails rapidly for the rest. The iterative method performs well for real numbers less than 2 and starts to fail rapidly for the rest. However the error is not extremely high given the number only deviates at the order of $10^{-}5$.

6 Conclusion

Simpsons method and trapezoidal methods provide good numerical aproximations for most situations but it may be nessesary to develop specialized methods in cases where they start to fail. The major points of faliure has been the rapid increase in the limits. The methods outlined perform well in situtation where the h value is relatively small, but having a very small h is highly impractical since it increases the iteration count rapidly.

7 References

References

[1] Godfrey, Paul (2001). "Lanczos Implementation of the Gamma Function" `http://www.numericana.com/answer/info/godfrey.htm`

[2] Bellard, Fabrice (2018). "Tiny C Compiler" `https://bellard.org/tcc/`.

[3] Wang, JC (2023). "Crossline" "A small, self-contained, zero-config, MIT licensed, cross-platform, readline and libedit replacement." `https://github.com/JunchuanWang80/crossline`.

[4] Santamaria, Ramon (2025). "Raylib" "A simple and easy-to-use library to enjoy videogames programming" `httsp://www.raylib.com`

Source Code for the project

- Integrator Prototype : `https://nimcompoo-04.github.io/numlysis/`
- Numerical Analysis Integrator : `https://www.github.com/NimComPoo-04/NumericalAnalysis`