Numerical approximation of $\mathcal{D}_t^\gamma u$ with a controlled precision

Nehemie Nguimbous

May 2023

1 Introduction and Background

As indicated in the title of [2], the primary goal of this study is to numerically solve a one-dimensional Gierer-Meinhardt model with sub-diffusion. The fractional operator $\mathcal{D}_t^\gamma u$ from lemma (2.1) in [2] is an inherent part of this system. It is essential to compute it with consistent precision to prevent any form of contamination. Additionally, we chose t instead of y as in [2] as the dependent variable of $\mathcal{D}_t^\gamma u$ to symbolize its time dependency, as it is conceptually the case.

The fractional operator $\mathcal{D}_t^\gamma u$ depends on two parameters: p and γ, with p being a parameter related to the function u. As shown in the next section, numerically computing $\mathcal{D}_t^\gamma u$ implies the integration of a spike-type fractional power function whose shape varies significantly, as shown in Figure (2) of Section (6). Since that function is bell-shaped with a very steep slope, computing its integral requires a significant number of subdivisions. Moreover, since the spikes are defined in an infinite domain, the number of subdivisions required increases even further. Even though computing $\mathcal{D}_t^\gamma u$ requires a high number of subdivisions in all cases, their order of magnitude varies significantly depending on the shape of the spikes. For example, a normal spike with a bell width of order $\mathcal{O}(\epsilon)$ requires a number of subdivisions of order $\mathcal{O}(1/\epsilon)$. Moreover, the spike tail decays exponentially, which makes it easy to be captured in a relatively small interval. For instance, a spike of width 0.1 centered at the origin on a domain of (-1, 1) will need $1/0.1 = 10$ subdivisions within (-0.05, 0.05) to be properly captured.

By contrast, an anomalous spike's tail decays algebraically (i.e., considerably more slowly). It will then need a very large and finely meshed interval to be properly captured. For example, an anomalous spike's tail could decay to 10^{-5} over a domain (-100, 100). The situation can worsen when dealing with interactions between several spikes, where the domain increases even further, or in the case of chaotic behavior where extremely fine time resolution is required. Furthermore, the number of subdivisions also increases with the precision being required.

An important consideration arises: while employing numerical methods to approximate subdivisions, these methods often provide error-bounded formulas based on subdivisions. It is logical to question why we do not use such a formula to determine n based on the desired error. Let us explore this using Simpson's method as an example, which we will utilize further. According to [1, page 203], considering $|f^{(4)}|$ as the continuous fourth derivative of a function f, and M as any upper bound for the values of $|f^{(4)}|$ on $[a, b]$, then the error $|E_f|$ in the Simpson's rule approximation of the integral of f from a to b satisfies the inequality:

$$|E_f| \leq \frac{(b-a)^5}{180} \cdot \frac{M}{n^4}.$$

However, this approach faces two primary challenges. Firstly, it involves M, supposedly the maximum for the fourth derivative of f. Yet, determining a maximum for a complex function like $u^{(4)}$ is unfeasible. Secondly,

an issue also arises with the term $b-a$. Operating within a theoretically infinite domain where a and b can tend to infinity, the term $b-a$ becomes an indeterminate form.

In summary, to make the best possible use of computer resources, it is essential to find a way to determine the number of subdivisions required for controlling precision, depending on the shape of those spikes and the domain in which they are defined. Answering this question will be the subject of our study.

2 Procedure

1. We split the main term of $D_t^\gamma u$ using two integrations by parts to simplify it and replace the original improper integral by proper computable terms.
2. The results of these integrations is a sum of constants and a yet to be evaluated integral. We evaluate the integral using the composite Simpson method. We selected the C programming language due to its speed and efficiency, providing a significant advantage when dealing with memory-intensive and power-demanding data.
3. We gradually increment the number of subdivisions used by our program to compute these integrals until a precision of 10^{-10} is reached. These computations are performed for a predefined set of values of the parameters p and γ.
4. We attempted to plot the correspondence between the number of subdivisions n and the value of the integral obtained , but the resulting functions were neither smooth nor continuous. We then shifted our focus toward the residual function.
5. The shapes of the residual functions are similar to that of the hyperbolic arctangent function, which is then used as the fitting function.
6. Finally, we use the inverse functions of the fitting curves to approximate the number of subdivisions n given a residual value r.

In summary, by implementing the above procedure, we expect to build a program capable of computing $D_t^\gamma u$ for all values of p and γ with a precision of 10^{-10}.

3 Plan

1. Section 4 covers Step 1 in the procedure for regularizing $\mathcal{D}_t^\gamma u$.
2. Steps 2 and 3, starting from Section 6 up to Subsection 6.1.4, involve determining the number of subdivisions n needed to compute I. This calculation is for (t, p, γ) values within the ranges: $\{0.1, 1, 5\} \times \{1.5, 2, 2.5, \ldots, 4.5\} \times \{0.1, 0.2, 0.3, \ldots, 0.9\}$.
3. Steps 4, 5, and 6, spanning from Subsection 5.2 to 5.3.2, describe the fitting process and can be visualized as follows:

   We fit the correspondence

   $$n_{(t,p,\gamma)} \to R_{(t,p,\gamma)}(n),$$

   using a variant of the hyperbolic-arctangent function, denoted as f. Consequently,

   $$n_{(t,p,\gamma)} \to f_{(t,p,\gamma)}(n) \sim R_{(t,p,\gamma)}.$$

To find the number of subdivisions based on a specific residual (the inverse path), we determine the inverse f^{-1} of f so that

$$f^{-1}_{(t,p,\gamma)}(R) \to n_{(t,p,\gamma)},$$

where $n_{(t,p,\gamma)}$ and $R_{(t,p,\gamma)}$ represent the number of subdivisions and the corresponding residual values for specific t, p, and γ values.

4. The process is now generalized for $(t, p, \gamma) \in [0, 5] \times [1.5, 4.5] \times [0.1, 0.9]$ from Section 7 to 9 using a series of linear and bilinear interpolations.

5. Section 10 involves the verification process where the accuracy of our results is assessed.

4 Regularization of $\mathcal{D}_t^\gamma u$

From section (2.1) in [2], we have

$$\mathcal{D}_t^\gamma u(t) = \operatorname{sign}\left(\frac{dx_i}{d\sigma}\right) \frac{1}{\Gamma(-\gamma)} \int_0^\infty \left\{u(t) - u\left(t + \operatorname{sign}\left(\frac{dx_i}{d\sigma}\right) y\right)\right\} \left(-\frac{dx_i}{d\sigma}\frac{1}{y}\right)^{\gamma+1} dy, \tag{1}$$

with

$$\mathcal{D}^\gamma_{(-t)} u(-t)\Big|_{x_i'>0} = \mathcal{D}_t^\gamma u(t)\Big|_{x_i'<0}.$$

From (1), the expression of $\mathcal{D}_t^\gamma u$ is not directly computable because of the singularity present in the expression. Hence, we perform a double integration by parts and employ a Taylor expansion of u around a certain point to get rid of the singularity.
For $x_i' < 0$ and $t > 0$, we have

$$\mathcal{D}_t^\gamma u(t) = -\frac{1}{\Gamma(-\gamma)} \int_0^{t_\infty} \frac{u(t) - u(t-y)}{y^{\gamma+1}} dy.$$

The first integration by parts

$$a(y) = u(t) - u(t-y), \quad \frac{db}{dy} = \frac{1}{y^{\gamma+1}},$$

leads to

$$\mathcal{D}_t^\gamma u(t) = -\frac{1}{\Gamma(-\gamma)} \left\{ \frac{u(t) - u(t-t_\infty)}{\gamma t_\infty^\gamma} - \lim_{t\to 0} \frac{1}{\gamma} \frac{u(t) - u(t-t_\infty)}{t^\gamma} + \int_0^{t_\infty} \frac{u'(t-y)}{\gamma y^\gamma} dy \right\},$$

with $\lim\limits_{t\to 0} \dfrac{u(t) - u(t-y)}{t^\gamma} = 0$. The proof for this is straightforward since

$$\frac{u(t) - u(t-y)}{t^\gamma} \sim \frac{u(t) - \left(u(t) + u'(t)(-t) + \dfrac{u''(t)t^2}{2!} + \cdots + \cdots\right)}{t^\gamma} = u'(t)t^{1-\gamma} - \frac{u''(t)t^{2-\gamma}}{2} + \cdots$$

with $0 < \gamma < 1$. We obtain

$$\mathcal{D}_t^{\gamma}u(t) = -\frac{1}{\Gamma(-\gamma)}\left\{\frac{u(t)-u(t-t_\infty)}{\gamma t_\infty^{\gamma}} + \int_0^{t_\infty}\frac{u'(t-y)}{\gamma y^{\gamma}}dy\right\}.$$

By applying a second integration by parts on the last terms in the brackets,

$$a(y) = u'(t-y), \quad \frac{db}{dy} = \frac{1}{y^{\gamma}},$$

we obtain

$$\int_0^{t_\infty}\frac{u'(t-y)}{\gamma y^{\gamma}}dy = -\frac{t_\infty^{1-\gamma}}{\gamma-1}u'(t-t_\infty) - \frac{1}{\gamma(\gamma-1)}\int_0^{t_\infty}u''(t-y)t^{1-\gamma}dy.$$

This finally leads to

$$\mathcal{D}_t^{\gamma}u(t) = -\frac{1}{\Gamma(-\gamma)}\left\{\frac{t_\infty^{-\gamma}}{\gamma}\left(u(t)-u(t-t_\infty) - \frac{t_\infty^{1-\gamma}}{\gamma(\gamma-1)}u'(t-t_\infty) - \frac{1}{\gamma(\gamma-1)}\int_0^{t_\infty}u''(t-y)t^{1-\gamma}dy\right\}. \quad (2)$$

We have

$$\gamma\Gamma(-\gamma) = \Gamma(1-\gamma) \text{ and } \gamma(\gamma-1)\Gamma(-\gamma) = \Gamma(2-\gamma).$$

Additionally, from lemma (2.2) in [2],

$$u''(t-y) = u(t-y) - u^p(t-y),$$

which leads to

$$\int_0^{t_\infty}u''(t-y)t^{1-\gamma}dy = -\int_0^{t-t_\infty}u''(t)(t-y)^{1-\gamma}dy.$$

We finally obtain

$$\mathcal{D}_t^{\gamma}u(t) = -\frac{t_\infty^{-\gamma}}{\Gamma(1-\gamma)}\Big(u(t)-u(t-t_\infty)\Big) + \frac{t_\infty^{1-\gamma}}{\Gamma(2-\gamma)}u'(t-t_\infty) - \frac{1}{\Gamma(2-\gamma)}\int_y^{t-t_\infty}u''(y)(t-y)^{1-\gamma}\,dy. \quad (3)$$

Similarly, for $x_i' > 0$, $t < 0$ and since u is even, we have

$$\begin{aligned}
D_{(-t)}^{\gamma}u(-t) &= \frac{t_\infty^{-\gamma}}{\Gamma(1-\gamma)}\Big(u(-t)-u(-t+t_\infty)\Big) + \frac{t_\infty^{1-\gamma}}{\Gamma(2-\gamma)}u'(t_\infty-t) - \frac{1}{\Gamma(2-\gamma)}\int_{-t}^{-t+t_\infty}u''(y)(t+y)^{1-\gamma}\,dy \\
&= \frac{t_\infty^{-\gamma}}{\Gamma(1-\gamma)}\Big(u(t)-u(t-t_\infty)\Big) - \frac{t_\infty^{1-\gamma}}{\Gamma(2-\gamma)}u'(t-t_\infty) - \frac{1}{\Gamma(2-\gamma)}\int_{-t}^{-t+t_\infty}u''(-y)\Big(t-(-y)\Big)^{1-\gamma}\,dy \\
&= \frac{t_\infty^{-\gamma}}{\Gamma(1-\gamma)}\Big(u(t)-u(t-t_\infty)\Big) - \frac{t_\infty^{1-\gamma}}{\Gamma(2-\gamma)}u'(t-t_\infty) + \frac{1}{\Gamma(2-\gamma)}\int_{t}^{t-t_\infty}u''(y)(t-y)^{1-\gamma}\,dy
\end{aligned}$$

with $0 < \gamma < 1$, and u defined as

$$u(t) = \left(\frac{p+1}{2}\operatorname{sech}^2\frac{(p-1)t}{2}\right)^{\frac{1}{p-1}},$$

and verifying the following differential equation

$$u'' - u + u^p = 0, \quad -\infty < t < \infty. \tag{4a}$$

$$u'(0) = 0, \quad u(0) > 0, \quad \text{and} \quad \lim_{|t|\to\infty} u = 0. \tag{4b}$$

Let us have

$I_1 = \dfrac{t_\infty^{-\gamma}}{\Gamma(1-\gamma)}\Big(u(t-t_\infty) - u(t)\Big), I\_2 = \dfrac{t_\infty^{1-\gamma}}{\Gamma(2-\gamma)}u'(t-t_\infty),$ and $I\_3 = \dfrac{t_\infty^{1-\gamma}}{\Gamma(2-\gamma)}\displaystyle\int_t^{t-t_\infty} u''(y)(t-y)^{1-\gamma}\,dy.$
In both cases, $\mathcal{D}_t^\gamma u$ is expressed as the sum of three terms, the first two being constants, and the third being an integral. The first two terms are exact values, hence our approximation will focus solely on the integral. Furthermore, it's important to note that for both $t > 0$ and $t < 0$, the integrand as well as the integration bounds remain the same. Therefore, the number of subdivisions required to compute the integral with the desired precision will be the same for both signs of t. Hence, we will concentrate solely on determining $I = \displaystyle\int_t^{t-t_\infty} u''(y)(t-y)^{1-\gamma}\,dy$ for a specific sign of t, in our case, $t > 0$.

5 Numerical approximation of I with a controlled precision

5.1 Structure of the project

Below is the repository tree of the project. At the root of the project, you have two main repositories, Code and Data.

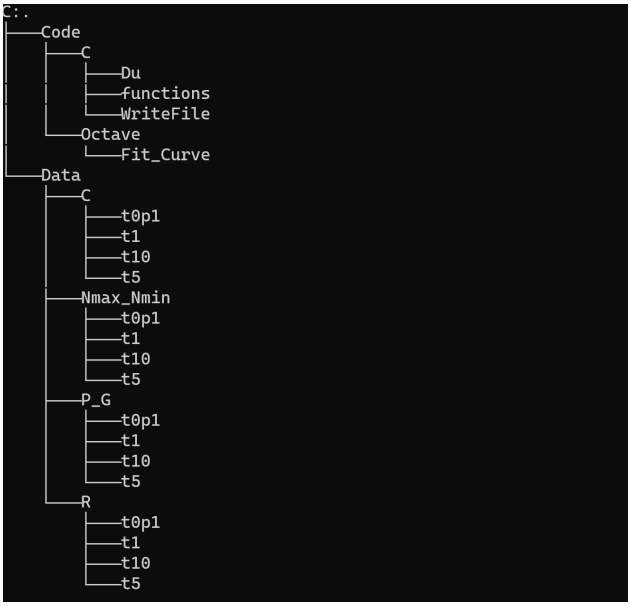


Figure 1: Tree diagram of the project repository.

- Repository path: `\Code`.
- Repository description: Contains all the code for the project, either in C or in Octave.
- Repository content: `C`, `Octave`.
  - Repository path: `\Code\C`.
  - Repository description: Contains all the C code for the project.
  - Repository content: `Du`, `functions`, `WriteFile`.
    - Repository path: `\Code\C\functions`.
    - Repository description: Contains all the functions and their prototypes used to approximate the number of subdivisions.
    - Repository content: `functions.h`, `functions.c`.
      - File name: `functions.h`.
      - Contains all the functions' prototypes used to approximate the number of subdivisions.
      - File name: `functions.c`.
      - File description: Contains the implementation of all the functions used in the approximation process.
      - File content: `sech`, `u`, `u1`, `u2`, `integrand`, `I_1`, `I_2`, `I_3`, `Du`, `simpson`, `findPosition_p`, `findPosition_g`, `linearN_T`, `bilinearN_p_g`, `n_approx`, `n_approx_general`, `N_T`, `N_logT`, `logN_T`, `logN_logT`, `even`, `reverse_n`, `join`.
      - File name: `functions.h`.
      - File description: Contains the prototypes of all the functions used in the approximation process.
    - Repository path: `\Code\C\Du`.
    - Repository description: Contains the C file used to reconstruct the curve of $\mathcal{D}_t^\gamma u$.
    - Repository content: `Du.c`.
      - File name: `Du.c`.
      - File description: Contains the code used to reconstruct the curve of $\mathcal{D}_t^\gamma u$.
    - Repository path: `\Code\C\WriteFile`.
    - Repository description: Contains the C file used to implement the composite Simpson method and to print its returned value in text files.
    - Repository content: `WriteFile.c`.
      - File name: `WriteFile.c`.
      - File description: Contains the C code used to implement the composite Simpson method and to print its returned value in text files.
  - Repository path: `\Code\Octave`.
  - Repository description: contains all the Octave code of the project.
  - Repository content: `Fit_Curve`.
    - Repository path: `\Code\Octave\Fit_Curve`.

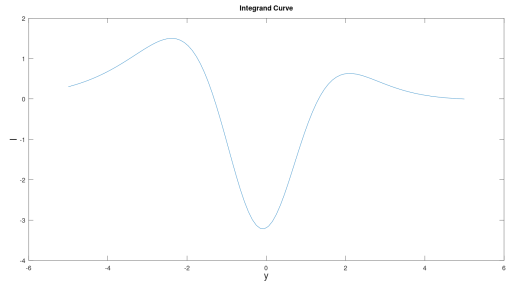
    - Repository description: Contains all the Octave files used to fit the residuals.
    - Repository content: `test1.m`, `fit_curve.m`, `fit_curvef.m`
        - file name: `test1.m`.
        - file description: Main file containing the overall logic of the fitting process.
        - file name: `fit_curve.m`.
        - file description: Contains the implementation of the `fit_curve` function which use is explained later in this document.
        - file name: `fit_curvef.m`.
        - file description: Contains the implementation of the `fit_curvef` function which use is explained later in this document.

- Repository path: `\Data`.
- Repository description: Contains all the data used in our project.
- Repository content: `P_G`, `Nmax_Nmin`, `R`, `C`.
    - Repository path: `\Data\P_G`.
    - Repository description: Holds the files containing the returned value of the Simpson method for $t = 0.1$, 1, 5.
    - Repository content: `t0p1`, `t1`, `t5`.
    - Repository path: `\Data\C`.
    - Repository description: Holds the files containing the $c(c_1, c_2)$ parameters for $t = 0.1$, 1, 5.
    - Repository content: `t0p1`, `t1`, `t5`.
    - Repository path: `\Data\Nmax_Nmin`.
    - Repository description: Holds the files containing the minimum and maximum number of subdivisions for $t = 0.1$, 1, 5.
    - Repository content: `t0p1`, `t1`, `t5`.
    - Repository path: `\Data\R`.
    - Repository description: Holds the files containing the residuals for $t = 0.1$, 1, 5.
    - Repository content: `t0p1`, `t1`, `t5`.

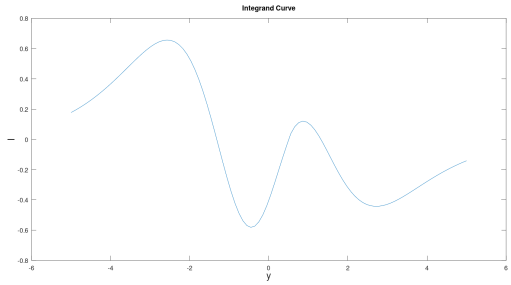
6 Finding the number of subdivisions required to approximate I with a controlled precision

The integrand of I depends on three parameters: p, γ, and t_∞. Based on extensive numerical trials and as demonstrated in the results section, the value of $t_\infty = 5$ has proved to be sufficient to approach I with the desired precision. In the subsequent sections of this document, when referring to discrete values of p and γ, we respectively mean $p \in \{1.5, \ldots, 4.5\}$ and $\gamma \in \{0.1, \ldots, 0.9\}$. Conversely, when discussing continuous

values of p and γ, we are referring to $p \in [1.5, 4.5]$ and $\gamma \in [0.1, 0.9]$. As mentioned earlier, the shape and gradient exhibit significant variation based on these parameters, influencing the number of subdivisions required for computation. To streamline the process and avoid redundant calculations of I, we seek a method to predict the necessary number of subdivisions and assess precision as we compute the integral. Below are the curves of the integrand for some values of the parameters.



(a) $u''(y)(t-y)^{1-\gamma}$ for $t = 5$



(b) $u''(y)(t-y)^{1-\gamma}$ for $t = 0.5$

Figure 2: Plots of $u''(t)(t-y)^{1-\gamma}$ for $p = 2$, $\gamma = 0.1$ for $t = 5$(right) and $t = 0.5$(left). The x-axis represents range of y-values and the y-axis the corresponding integrand values.

6.1 Computing I using the composite Simpson method

The following composite Simpson algorithm from [1, page 204] is used to compute the value I.

$$\int_a^b f(x)dx \approx \frac{h}{3} \sum_{i=0}^{n/2-1} \Big[f(x_{2i}) + 4f(x_{2i+1}) + f(x_{2i+2}) \Big] \quad \text{with} \quad h = \frac{b-a}{n}.$$

6.1.1 Implementation of the Simpson method in C: located in functions.c

This function computes the integral of the function `integrand` using the composite Simpson method.

Parameter list:

- `lower`: lower bound of the integral.
- `upper`: upper bound of the integral.
- `subInterval`: number of sub-intervals.
- `g`: gamma parameter.
- `A = pow((p+1.0)/2.0, 1.0/(p-1.0))`.
- `B = 2.0/(p-1.0)`.
- `C = (p-1.0)/2.0`.

Process:

- Calculate the step size.

- Sum the image by `integrand` of the lower and upper bounds of the integral first.
- Add the images by `integrand` of the values between the lower and upper bounds depending on whether the discretization index is odd or even.
- Multiply the result by one-third of the `stepSize`.

```
1
2
3
4  double simpson(double lower, double upper, long int subInterval, double p, double
       g) {
5
6    double stepSize = (upper - lower / subInterval);
7
8    int i;
9    double integration = integrand(lower, p, g, y) + integrand(upper, p, g, y);
10   for (i = 1; i <= subInterval - 1; i++) {
11     double k = lower + i * stepSize;
12     if (i % 2 == 0) {
13       integration = integration + 2 * integrand(k, p, g, y);
14     } else {
15       integration = integration + 4 * integrand(k, p, g, y);
16     }
17   }
18   integration = integration * stepSize / 3;
19   return integration;
20
21 }
22   // implementation of the integrand
23
24 double integrand(double y, double p, double g, double t) {
25
26   return (u(y, p) - pow(u(y, p), p)) * pow(t - y, 1 - g);
27
28 }
29
30 // implementation of u
31
32 double u (double t,double p)
33 {
34     double A = pow( (p+1.0)/2.0, 1.0/(p-1.0));
35     double B =  2.0/(p-1.0);
36     double C = (p-1.0)/2.0;
37
38     return (A*pow(sech(C*t), B));
39 }
```

Listing 1: Implementation of the composite Simpson method with its dependencies in C

Note: The number of `subInterval` used by the Simpson method must be even. Lines 48 to 50 from Listing 11 always ensure it is the case.

6.1.2 Printing the result of the Simpson method using the *WriteFile* function: located in *WriteFile.c*

For each combination of t, p, and γ, the `WriteFile` function creates a file and prints, side by side, the return values of the Simpson method with their corresponding number of subdivisions until the desired precision is reached. Considering that we are dealing with very large values of n, and only the last values of n are of interest, we double the increment value each time we run the first loop until we reach a precision of 5×10^{-10}. Once that precision is achieved, new values of the increments are used until 1×10^{-10} is reached. The value of the increments used in the second loop is returned by the `getIncrement` function and represents 5% of the number of subdivisions required to reach the 5×10^{-10} precision in the first loop. This `WriteFile` function is used to print the returned value of the Simpson method for discrete values of p and γ.

Parameter list:

- `p`: discrete values of p.
- `g`: discrete values of γ.
- `t`: $\in \{0.1, 1, 5\}$.
- `n`: max number of subdivisions allowed.

Process:

- Extract the integer and decimal part of `p` and `g`.
- Write or create a file for a specific combination of `p` and `g` where the returned value of the Simpson method will be printed.
- Run the first loop and double the increment until $1 \times 10^{-10} <$ `error` $< 5 \times 10^{-10}$.
- Run the second loop with a custom increment until `error1` $< 10^{-10}$.
- `error` and `error1` have the same expression as `Simpson(lower, upper, n) - Simpson(lower, upper, 2n)`.

Please note that "`Simpson(lower, upper, n)`" represents the result of the Simpson method with the given parameters `lower`, `upper`, and `n`.

```
1  void writeFile(double p, double g, double t, long int n) {
2
3    double A = pow((p + 1.0) / 2.0, 1.0 / (p - 1.0));
4    double B = 2.0 / (p - 1.0);
5    double C = (p - 1.0) / 2.0;
6    double upper = t;
7    double tmax = 5;
8    double lower = t-tmax;
9
10   int int_g = 10 * g;
11   double p1 = (int) p;
12   double p2 = 10 * (p - p1);
13   long int i = 0;
```

```
14
15   double g1 = (int) g;
16   double g2 = 10 * (g - g1);
17   long int increment = 1000;
18   long int next;
19   double precision1 = 5e-10;
20   double precision2 = 1e-10;
21
22   long int j;
23   char str_i[25];
24   FILE * diag;
25   FILE * diag1;
26
27   sprintf(str_i, "p%0.0fp%0.0f_g%0.0fp%0.0f.txt", p1, p2, g1, g2);
28
29   diag = fopen(str_i, "a");
30   if (!diag) {
31     printf("Failed to open diagonal dominance file.\n");
32
33   }
34
35   for (i = 1000; i < n; i += i) {
36
37     double value = simpson(lower, upper, i, g, A, B, C);
38
39     double value5 = simpson(lower, upper, 2 * i, g, A, B, C);
40
41     double error = fabs(value - value5);
42
43     if (error > precision1) {
44
45       fprintf(diag, "%ld %0.15f\n", i, value5);
46     } else if (error <= precision1 && error > precision2) {
47       next = i;
48       break;
49     }
50
51   }
52
53 increment = getIncrement (t, p, g); // Find the increment depending on t, p and
       gamma
54
55   for (j = next; j < n; j += increment) {
56
57     double value1 = simpson(lower, upper, j, g, A, B, C);
58
59     double value10 = simpson(lower, upper, 2 * j, g, A, B, C);
60
61     double error1 = fabs(value1 - value10);
62
```

```
63      if (error1 < precision2) {
64
65        break;
66      } else {
67
68        fprintf(diag, "%ld %0.15f\n", j, value10);
69
70      }
71
72    }
73
74    fclose(diag);
75  }
```

Listing 2: Implementation *WriteFile* method in C

6.1.3 Calling the *WriteFile* inside the *main* function: located in *WriteFile.c*

This section shows how the `WriteFile` function is called in the main function for every values of p and γ. We fixed the maximum number of subdivisions allowed to 7 billions.

```
1  int main() {
2
3    double p = 1.5;
4    double g = 0.1;
5    long int n = 7e9;
6
7    for (p = 1.5; p <= 4.5; p += 0.5) {
8
9      for (g = 0.1; g < 0.9; g += 0.1) {
10
11       writeFile(p, g, t, n);
12
13     }
14
15   }
16   return 0;
17 }
```

Listing 3: Calling the WriteFile method.

6.1.4 Maximum number of subdivisions required for $t = 0.1$, 1 and 5

When the desired precision is reached, the last line printed by the `WriteFile` function holds the returned value of the Simpson method with the right precision as well as the number of subdivisions required to reach that precision. Below are tables displaying the maximum number of subdivisions for every t, p, and γ.

p/g	1.5	2.0	2.5	3.0	3.5	4.0	4.5
0.1	35290	51740	61610	73870	80450	87030	93610
0.2	99900	135800	163900	199800	217750	235700	253650
0.3	216940	344940	433880	522820	567290	611760	689880
0.4	609778	865778	1101630	1377778	1495704	1731556	1849482
0.5	1693204	2362408	3386408	3721010	4390214	5059418	5394020
0.6	4718250	7300300	9970550	12464400	14066550	15668700	17270850
0.7	16113048	28265572	38166882	46087930	54550882	60491668	68412716
0.8	76333764	132864908	182371458	223956960	263749554	301374532	333058724
0.9	383072000	730144000	1018144000	1028288000	1028288000	1028288000	1028288000

Table 1: Number of subdivisions for $t = 0.1$

p/g	1.5	2.0	2.5	3.0	3.5	4.0	4.5
0.1	28000	26500	11000	22000	32500	39500	42500
0.2	71900	67550	23900	55700	83750	103500	111400
0.3	175500	166000	60500	130500	204000	251500	280000
0.4	444800	397600	134800	316800	515600	633600	700800
0.5	1141700	1078730	318970	826850	1393580	1716670	1905580
0.6	3547200	3172400	886800	2335800	4109400	5233800	5983400
0.7	11817400	10530500	2691450	7956700	14626500	18487200	21061000
0.8	52417200	46626150	12460850	34400600	63653600	83600550	95826100
0.9	260332000	232504000	44212000	162934000	325868000	451094000	506750000

Table 2: Number of subdivisions for $t = 1$

p/g	1.5	2.0	2.5	3.0	3.5	4.0	4.5
0.1	17100	11600	9100	8000	7300	6900	6600
0.2	42200	28000	21700	18800	17300	16100	15500
0.3	98200	63800	48800	41600	38200	35800	34000
0.4	234500	146500	110500	94000	84500	78500	75500
0.5	596400	364800	268600	227900	202000	187200	178700
0.6	1684000	992000	716000	596000	526000	486000	456000
0.7	5458000	3070000	2171000	1768000	1566000	1442000	1349000
0.8	21119000	11412000	7847000	6357000	5551000	5055000	4714000
0.9	101674000	51754000	34392000	27318000	23650000	21292000	19720000

Table 3: Number of subdivisions for $t = 5$

6.2 Fitting the residual function

Once the `WriteFile` function is executed, we are left with files containing two columns: the first representing the number of subdivisions, and the other displaying the returned values of the Simpson method. Table 4 shows a sample file printed by the `WriteFile` function for $t = 1$, $p = 4$, and $\gamma = 0.8$. Since our objective is to find the number of subdivisions required to reach a specific precision or error, it makes sense to establish a relationship or correspondence between N and the error or residual (difference between the last value printed by the `WriteFile` function and the other ones). Table 5 is an update of Table 4, now with residual values as the second column.

N	I
75879150	-0.590274039200124
76522600	-0.590274039199605
77166050	-0.590274039198941
77809500	-0.590274039197867
78452950	-0.590274039197240
79096400	-0.590274039196180
79739850	-0.590274039195590
80383300	-0.590274039194982
81026750	-0.590274039194301
81670200	-0.590274039193123
82313650	-0.590274039192075
82957100	-0.590274039191671
83600550	-0.590274039191116
84244000	-0.590274039189668

Table 4: table4

N	R
75879150	1.2251e-11
76522600	1.0456e-11
77166050	9.9369e-12
77809500	9.2729e-12
78452950	8.1990e-12
79096400	7.5719e-12
79739850	6.5120e-12
80383300	5.9219e-12
81026750	5.3140e-12
81670200	4.6330e-12
82313650	3.4550e-12
82957100	2.4070e-12
83600550	2.0030e-12
84244000	1.4480e-12

Table 5: table5

6.2.1 Inside the test1.m file

The next step is to fit this correspondence with a usual function such that given any value of R, we could find its corresponding N. During the fitting process of the residual function, we opted for the logarithm of the residual instead of the actual residual function. In fact, the residual function is a power function, which implies that its values are either extremely small or large over most of its domain. Then, attempting to get insights from its plot has shown to be quite challenging. Also, the fit would be highly non-uniform. For example, how do we ensure that the small values are fit as well as the large values? How do we measure what is a good fit when the disparity in magnitude is so great? The *log* is the answer: it turns a very strongly varying power function into quasi-straight lines. Straight lines are simpler to handle in terms of quantifying their properties. If the original function is not a single power (which is our case), the *log* would not be a straight line, but the magnitude disparity issue is resolved nonetheless. The other aspect of it all is that we are indeed interested in the error magnitude, not its absolute value. We will always describe the residual as a magnitude. Thus, the *log* treatment also makes sense conceptually, beyond being a technical convenience. In the subsequent sections of the document, when we mention the residual function, we are actually referring to its logarithm. Process:

- Extract the integer and decimal parts of `p` and `g` and convert them into strings.
- Open the file printed by the `writeFile` function for that specific combination of p and g.
- Load this file into the vector U.
- Compute the residual function.
- Fit the residual function.
- Plot the results.

In the context of the `test1.m` file, the goal is to establish a functional relationship between the number of subdivisions N and the error or residual R for a given combination of p and γ. By fitting the plot of the

residual against the number of subdivisions with a continuous and easily invertible function, the program can determine the number of subdivisions required to achieve a specific precision.

```
1
2  for p=1.5:0.5:4.5
3    for g = 0.1:0.1:0.9
4
5    %extract the integer and decimal part of p
6
7      p1=fix(p);
8      p2=10*(p-p1);
9
10 %extract the integer and decimal part of g
11     g1=fix(g);
12     g2=10*(g-g1);
13
14 %convert them into strings
15     sp1=num2str(p1);
16     sp2=num2str(p2);
17     sg1=num2str(g1);
18     sg2=num2str(g2);
19
20  %   open the file containing the return value of the WriteFile
21  method depending on a sepcific combination of p and g
22  %
23
24 file=['C:/Users/nguim/OneDrive/Documents/research_papers
25 /Code/newdata/P_G/t5/p' sp1 'p' sp2 '_g' sg1 'p' sg2 '.txt'];
26
27 % load the file as a 2 dimoentional vector
28     U=load(file);
29
30     N=U(:,1); % the vector N contains the subdivisions
31     G=U(:,2); % the vector G contains the actual return value the  WriteFile method
32
33  % R is the residual vector containing the difference between the last value of G
       and the other ones
34  %
35     R=log10(abs(G(1:end-1)-G(end)));
36     N=N(1:end-1);
37
38 % c is a parameter vector that will minimizes the euclidian distance between
39 the Residual functions and their fitted curves
40 %
41     c0=[-1 -6]; % initial parameter of the fminunc function
42     c=fminunc(@(c) fit_curve(c,R,N), c0);
43
44     [f,imin,imax]=fit_curvef(c,N);
45     N1=N;
46     N1([imin imax])=[];
```

```
47    plot(N,R,N1,f,'--');
48    hold on
49
50   endfor
51 endfor
```

Listing 4: Fitting the residual function.

6.2.2 Inside the fit_curvef.m file

The `fit_curvef` function is responsible for performing the fitting task of the residual using `atanh`.
Parameters list:

- `c`: a two-dimensional vector.
- `n`: the number of subdivisions.

Process:

- Get the minimum value of the vector `n`.
- Get the maximum value of the vector `n`.
- Remove these extreme values from the vector `n`.
- Determine the fitting curve using the parameter `c`.

```
1
2 function [f,imin,imax]=fit_curvef(c,n)
3
4 [minn,imin]=min(n);
5 [maxn,imax]=max(n);
6 n([imin imax])=[];
7 n1=2*(n-minn)/(maxn-minn)-1;
8 f=c(1)*atanh(n1)+c(2);
```

Listing 5: Implementation of the fit_curvef method .

6.2.3 Inside the fit_curve.m file

The `fit_curve` function returns the Euclidean distance between the residuals and their fitting curves.
Parameters list:

- `c`: a two-dimensional vector returned by the `fminunc` function.
- `u`: the function being fitted; in our case, it represents the residual function.
- `n`: represents the number of subdivisions.

Process:

- Call the `fit_curvef` function and pass into it the vector `c`.

- `fit_curvef` returns the fitting curve `f`, as well as `Imax` and `Imin`, which respectively represent the maximum and minimum values of `n`.
- Remove `Imax` and `Imin` from `n`.
- Determine the Euclidean distance between the fitting curve `f` and `u`.

```
1
2 function R=fit_curve(c,u,n)
3
4 [f,imin,imax]=fit_curvef(c,n);
5 u([imin imax])=[];
6 R=sqrt(sum((f-u).^2));
```

Listing 6: Implementation of the *fit_curve* method.

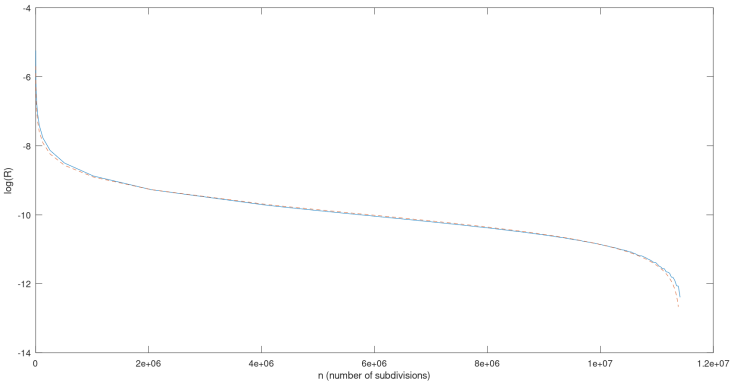


Figure 3: specific case for $t = 5$, $p = 2$, and $\gamma = 0.8$. The x-axis represents the number of subdivisions and the y-axis the logarithm of the corresponding residuals. The solid curve represents the numerical results while the dashed one represents the fitting curve.

6.3 Approximating N using the fitting curve

Once completing the fitting with the `atanh` function , its inverse can now be utilised to determine the corresponding R for every value of n. From the `fit_curvef.m` file, we have $n_1 = \dfrac{2 \times (n - \text{minn})}{(\text{maxn} - \text{minn})} - 1$ and $f = c_1 \times \text{atanh}(n_1) + c_2$. The inverse process leads to $n_1 = \dfrac{\tanh(f - c_2)}{c_1}$ and $n = \dfrac{(n_1 + 1) \times (\text{maxn} - \text{minn})}{2} +$ 1. This expression is used to approximate n in the `n_approx` function present in the `functions.c` file.

6.3.1 Inside the *n_approxfunction*: located in functions.c

This function computes and returns the number of subdivisions `n` given `t`, `p`, `g`, and the residual `R`.

Parameters list:

- `t`: upper bound of the integral.
- `p`: between 1.5 and 4.5 with an increment of 0.5.
- `g`: between 0.1 and 0.9 with an increment of 0.1.
- `R`: the residual value.

Process:

- Load the `c.txt` containing the different values of the `c` vector for a given `t`.
- Load the `Nmax_Nmin.txt` file containing the maximum and minimum number of subdivisions for a given `t`.
- Compute `n` using the new formula.

```
double n_approx(double t, double p, double g) {

  double n1, n, pos_p, pos_g;
  FILE * fp1;
  FILE * fp2;
  FILE * fp3;
  char file_name1[25];
  char file_name2[33];
  char file_name3[25];
  float c1 = 0, c2 = 0, nMax = 0, nMin = 0, R;
  char * param;
  int i = 0, location = 0;

  pos_p = 1 + (p - 1.5) / 0.5;
  pos_g = g / 0.1;

  // find the location of c1 and c2 in the C.txt file as well as the location of
      Nmax and Nmin in the N.txt file knowing p and g

  location = (int)((pos_p - 1) * 9 + pos_g);

  // assign a value to param based on the value of t

  if (t == 0.1) {
    param = "0p1";
  } else if (t == 1) {
    param = "1";
  } else  {
    param = "5";
  }

  // create a path to the C.txt, N.txt, and R.txt files using the param variable

  sprintf(file_name1, "../../Data/C/t%s/C.txt", param);
  sprintf(file_name2, "../../Data/Nmax_Nmin/t%s/N.txt", param);
```

```
35    sprintf(file_name3, "../../Data/R/t%s/R.txt", param);
36
37    // open N.txt, C.txt, and R.txt
38
39    fp1 = fopen(file_name1, "r");
40    fp2 = fopen(file_name2, "r");
41    fp3 = fopen(file_name3, "r");
42
43    if (!fp2) {
44      printf("fail_2");
45    }
46
47    if (!fp1) {
48      printf("fail_1");
49    }
50    if (!fp3) {
51      printf("fail_3");
52    }
53
54    for (i = 0; i < location; i++) {
55      fscanf(fp1, "%f %f  ", & c1, & c2); // find c1 and c2 using the location
          variable
56      fscanf(fp2, "%f %f  ", & nMin, & nMax); // find nMin and nMax using the
          location variable
57      fscanf(fp3, "%f ", & R); // find R  using the location variable
58
59    }
60   ;
61    R = R - 1.5;
62
63    // close the files
64
65    fclose(fp1);
66    fclose(fp2);
67
68    n1 = tanh((R - c2) / c1);
69    n = ((n1 + 1) * (nMax - nMin)) / 2 + 1; // find n
70
71    return n;
72
73  }
```

Listing 7: Implementation of the *n_approx* method.

6.3.2 Approximated number of subdivisions required for $t = 0.1$, 1, and 5

Below are the approximate number of subdivisions returned by the `n_approx_general` function for $t = 0.1$, $t = 1$ and $t = 5$.

g/p	1.5	2.0	2.5	3	3.5	4	4.5
0.1	35065	51414	61242	73570	80108	86651	93201
0.2	99002	134710	162937	198630	216504	234382	252262
0.3	214565	342453	430919	519457	563799	608143	686814
0.4	602747	858551	1093366	1370345	1487766	1722571	1840192
0.5	1674913	2341552	3365120	3698389	4365537	5033725	5366720
0.6	1674913	7241978	9904136	12403810	13999256	15597980	17196034
0.7	4662043	28071526	37953058	45866233	54342976	60260601	68207443
0.8	15930379	132684487	182123946	223677191	263586831	301086078	332831576
0.9	76155108	727802772	1015373018	1025834015	1025841827	1025918580	1025859852

Table 6: Approximated number of subdivisions for $t = 0.1$

g/p	27873	26378	10927	21895	32362	39371	42358
0.1	27873	26378	10927	21895	32362	39371	42358
0.2	71597	67150	23662	55344	83377	103044	110919
0.3	174554	165106	59775	129525	202917	250255	278688
0.4	442094	395096	132836	314216	512656	630341	697983
0.5	1134337	1071579	313754	820575	1385528	1708829	1897254
0.6	1134337	3153872	871089	2317025	4088223	5210846	5958598
0.7	3527550	10468585	2640180	7900759	14564452	18417018	20991875
0.8	11753526	46563914	12411139	34340174	63593768	83514848	95754535
0.9	52352342	231613175	43366292	162079821	324972385	449967670	505825475

Table 7: Approximated number of subdivisions for t = 0.1

g/p	17083	11585	9085	7986	7286	6886	6586
0.1	17083	11585	9085	7986	7286	6886	6586
0.2	42163	27968	21670	18772	17273	16074	15474
0.3	98128	63737	48741	41544	38147	35748	33949
0.4	234320	146338	110344	93857	84362	78364	75366
0.5	595947	364389	268200	227526	201649	186865	178355
0.6	595947	990878	714928	595009	525021	485046	455084
0.7	1682968	3066607	2168010	1765265	1563165	1439166	1346239
0.8	5453895	11407642	7844868	6353968	5548342	5052182	4711303
0.9	21115912	51737121	34370523	27302158	23631082	21275601	19703359

Table 8: Approximated number of subdivisions for $t = 5$

7 Number of subdivisions required to compute I for continuous values of t and discrete values of p and γ

Now that we have a program capable of predicting the number of subdivisions n required to compute I with controlled precision, our goal is to expand it for continuous values of t, discrete values of p and γ. For that, we implemented four different variants of linear interpolation formulas and selected the most conservative one, which returns the highest value of n. For $t \in [t_1, t_2]$ their general formulas are

1. $n_t = n_{t_1} + \frac{(t-t_1)(n_{t_2}-n_{t_1})}{(t_2-t_1)}$,
2. $n_t = n_{t_1} + \frac{(log(t)-log(t_1))(n_{t_2}-n_{t_1})}{(log(t_2)-log(t_1))}$,
3. $log(n_t) = log(n_{t_1}) + \frac{(t-t_1)(log(n_{t_2})-log(n_{t_1}))}{(t_2-t_1)}$,
4. $log(n_t) = log(n_{t_1}) + \frac{(log(t)-log(t_1))(log(n_{t_2})-log(n_{t_1}))}{(log(t_2)-log(t_1))}$,

with log referring to the log_{10} function.

7.1 Implementation in C: located in the functions.c file.

Below are their respective implementation in C.

```
1
2
3  int N_T(double t, double t1, double t2, int n1, int n2)
4  {
5     int  n = (int)(n1 + (t - t1)*(n2 - n1)/(t2 - t1));
6     return n;
7  }
8
9  int N_logT(double t, double t1, double t2, int n1, int n2)
10 {
11    int n = (int)(n1 + (log10(t) - log10(t1))*(n2 - n1)/(log10(t2) - log10(t1)));
12    return n;
13 }
14
15 int logN_T(double t, double t1, double t2, int n1, int n2)
16 {
17    double logn = log10(n1) + (t - t1)*(log10(n2)-log10(n1))/(t2 - t1);
18    return (int)(pow(10, logn));
19 }
20
21 int logN_logT(double t, double t1, double t2, int n1,  int n2)
22 {
23    double logn = log10(n1) + (log10(t) - log10(t1))*(log10(n2)- log10(n1))/
24    (log10(t2) - log10(t1));
25    return (int)(pow(10, logn));
26 }
```

Listing 8: Implementation of the previous interpolation functions in C.

7.2 Determining the most conservative linear interpolation method: inside the $linearN_T$ function, located in the functions.c file

Since we have four variants of linear interpolation, the task of `linearN_T` is to find the most conservative one. `n1` and `n2` are respectively the number of subdivisions required for `t1` and `t2` with `t1 <= t <= t2`. The `linearN_T` function evaluates the number of subdivisions required for a given range of `t` values and then selects the most conservative interpolation method among the four available variants.

```c
int linearN_T(double t, double t1, double t2, int n1, int n2)
{
  int   temp1 = N_T(t, t1, t2, n1, n2);
  int  temp2 = N_logT(t,t1, t2, n1, n2);
  int   temp3 = logN_T(t, t1, t2,  n1, n2);
  int   temp4 = logN_logT(t, t1, t2,  n1, n2);

  int n = temp1;

  if (n < temp2)
  {
    n = temp2;
  }

  if (n < temp3)
  {
    n = temp3;
  }

  if (n < temp4)
  {
    n = temp4;
  }

  return n;
}
```

Listing 9: Code used to compare the different interpolation methods.

8 Number of subdivisions required to compute I with a control precision for continuous values of t, p and and γ

In the last section, we were able to reconstruct the curve of $D_t^\gamma u$ for $p = 2$ and discrete values of γ. Moreover, we can do the same for all discrete values of p as well. However, we face a problem for continuous values of p and γ. For example, for $p = 2.38$ and $\gamma = 0.46$, we need to employ a different approach. Again, we use interpolation, but this time it is bi-linear instead of linear. In fact, for a given (p, γ) such that $p_1 \leq p \leq p_2$ and $\gamma_1 \leq \gamma \leq \gamma_2$, we want to find its corresponding number of subdivisions n. Below are the bi-linear interpolation formula and its implementation in our code.

8.1 Bi-linear interpolation formula

$$n = \frac{\gamma-\gamma_2}{\gamma_1-\gamma_2}\left(\frac{n_{p_1,\gamma_1}(p-p_2)}{p_1-p_2} + \frac{n_{p_2,\gamma_1}(p-p_1)}{p_2-p_1}\right) + \frac{\gamma-\gamma_1}{\gamma_2-\gamma_1}\left(\frac{n_{p_1,\gamma_2}(p-p_2)}{p_1-p_2} + \frac{n_{p_2,\gamma_2}(p-p_1)}{p_2-p_1}\right).$$

8.2 Implementation in C: inside the billinearN_p_g function: inside the functions.c file.

```
1
2  int bilinearN_p_g(double p, double g, double t) {
3
4    int n;
5    double p1, p2, g1, g2;
6    double n_p1g1, n_p1g2, n_p2g1, n_p2g2;
7
8    findPosition_p(p, & p1, & p2); // find p1
9    findPosition_g(g, & g1, & g2); // find p2
10
11   n_p1g1 = n_approx(t, p1, g1); // find n for p=p1 and g=g1
12   n_p1g2 = n_approx(t, p1, g2); // find n for p=p1 and g=g2
13   n_p2g1 = n_approx(t, p2, g1); // find n for p=p2 and g=g1
14   n_p2g2 = n_approx(t, p2, g2); // find n for p=p2 and g=g2
15
16   // find n
17
18   n = (int)((g - g2) / (g1 - g2) * (n_p1g1 * (p - p2) / (p1 - p2) + n_p2g1* (p -
         p1) / (p2 - p1))
19   + (g - g1) / (g2 - g1) * (n_p1g2 * (p - p2) / (p1 - p2) + n_p2g2 * (p - p1) / (
         p2 - p1)));
20
21   return n;
22 }
```

Listing 10: Bi-linear interpolation of n in C.

9 Summary of the interpolation process

On one hand, `n_approx` allows us to predict the number of subdivisions required to approximate I for discrete values of t, p, and γ. On the other hand, `n_approx_general` performs the same task but over their continuous range. It achieves this by employing both linear and bi-linear interpolation methods. The process to find the number of subdivisions required for an unknown triplet (t, p, γ) involves finding two triplets (t_1, p_1, γ_1) and (t_2, p_2, γ_2) such that $t_1 \leq t \leq t_2$, $p_1 \leq p \leq p_2$, and $\gamma_1 \leq \gamma \leq \gamma_2$. Moreover, it finds n_1 required for (t_1, p_1, γ_1) and n_2 required for (t_2, p_2, γ_2), and then computes n required for (t, p, γ) using interpolation. This comprehensive approach allows us to efficiently estimate the number of subdivisions needed to achieve the desired precision for the integral over a wider range of input values.

9.1 Algorithm of the *n_approx_general* function

Algorithm 1: *n_approx_general* Algorithm

Result: Find n for continuous values of y, p and γ
;
if $y \in \{0.1, 1, 5\}$ **then**
  n = bilinear(t, p, γ) ;
end
else
  find $(t_1, t_2) \in \{0.1, 1, 5\} \times \{0.1, 1, 5\}$ such that $t_1 \leq t \leq t_2$;
  n1 = bilinear(t1, p, γ) ;
  n2 = bilinear(t2, p, γ) ;
  n = linear(t, t1, n1, t2, n2) ;
end

9.2 Implementation in C of the n_approx_general function: : inside the functions.c file

```c
int n_approx_general(double t, double p, double g) {

  double t1, t2;

  double p1 = (int) p; // extract the integer part of p
  double p2 = 10 * (p - p1); // extract the decimal part of p
  long int i = 0;
  double g1 = (int) g; // extract the integer part of g
  double g2 = 10 * (g - g1); // extract the decimal part of p

  int n, n1, n2;

  // if t is not in the set {0, 0.1, 1, 5}, find t1 and t2 such that t1 <= t<= t2

  if (t != 0.1 && t != 5 && t != 1 && t != 0) {
    t1 = 0.1;
    t2 = 10;

    if (t < 1) {
      t1 = 0.1;
      t2 = 1;
    } else if (1 < t < 5) {
      t1 = 1;
      t2 = 5;
    } else {
      t1 = 0.1;
      t2 = 5;
    }

```

```
    n1 = bilinearN_p_g(p, g, t1); // find n for p, g and t1 using bilinear
        interpolation
    n2 = bilinearN_p_g(p, g, t2); // find n for p, g and t2 using bilinear
        interpolation

    n = linearN_T(t, t1, t2, n1, n2); // find n knowing n1 and n2 using linear
        interpolation

  } else if (t == 0) // find n for t=0
  {
    t = 0.1;
    n = bilinearN_p_g(p, g, t) + 1;

  } else {

    // if t is  in the set {0.1, 1, 5} use bilinear interpolation to compute it
        directly
    n = bilinearN_p_g(p, g, t);

  }
  // verify that n is even, because the composite Simpson method only works with
      even values of n.

  if (n % 2 != 0) {
    n += 1;
  }

  return n;
}
```

Listing 11: Implementation of the n_approx_general function

10 Assessment of the accuracy of the results

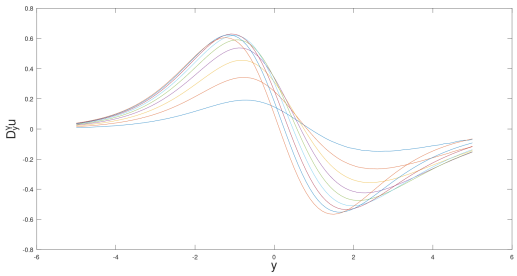
As a primary test for the quality of the results, we aim to reproduce the curve of $\mathcal{D}_t^\gamma u$ on $[-t_\infty, t_\infty]$, and compare its shape with the one from [1, page 12]. The newly computed curve is at the left and the former one at the right.

error	y
-4.744730e-008	3.6
-5.048329e-009	3.7
6.482569e-008	3.8
-4.435881e-009	3.9
2.060108e-007	4.0
-3.872901e-009	4.1
3.676208e-007	4.2
-3.364300e-009	4.3
-3.129406e-009	4.4
6.138696e-007	4.5
-2.703101e-009	4.6
7.505957e-007	4.7
-2.328432e-009	4.8
8.405479e-007	4.9
-2.001432e-009	5.0

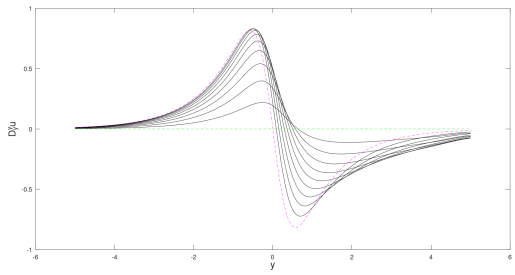
Table 9: Error file for $\gamma = 0.7$

error	y
-1.784137e-010	3.6
-1.721758e-010	3.7
-1.664409e-010	3.8
-1.605218e-010	3.9
-1.547047e-010	4.0
-1.494032e-010	4.1
-1.443059e-010	4.2
-1.395319e-010	4.3
-1.351115e-010	4.4
-1.307046e-010	4.5
-1.269774e-010	4.6
-1.235263e-010	4.7
-1.206676e-010	4.8
-1.180839e-010	4.9
-1.161406e-010	5.0

Table 10: New error file for $\gamma = 0.7$



(a) Plot of the reconstructed $\mathcal{D}_t^\gamma u$.



(b) Plot of the reconstructed $\mathcal{D}_t^\gamma u$.

Figure 4: Comparison between the former and reconstructed Plot of $\mathcal{D}_t^\gamma u$ for sign $\left(\frac{dx_i}{d\sigma}\right) < 0$ and $p = 2$. The x-axis represents range of t-values and the y-axis the associated $\mathcal{D}_t^\gamma u$.

The reproduction of $\mathcal{D}_t^\gamma u$ was just the visual part of the assessment process. In fact, we still need to verify the precision of $\mathcal{D}_t^\gamma u$ values. To achieve this, we compute the absolute difference between $\mathcal{D}_t^\gamma u$ for $N = n$, $N = 2n$ and $p = 2$. Initially, the `error` variable used in the stopping condition of the `writeFile` function was defined as `error = abs(Simpson(lower, upper, n) - Simpson(lower, upper, n + 10))`. Due to the non-monotonic and the uncontrolled character of the errors' magnitude depicted in table 9, we adjusted the `error` variable to `error = abs(Simpson(lower, upper, n) - Simpson(lower, upper, 2*n))`. This modification yielded satisfactory results, as illustrated in table 10.

10.1 Implementation of $\mathcal{D}_t^\gamma u$ in C: Inside the Du.c file

Below is the implementation of the $\mathcal{D}_t^\gamma u$ function.

10.1.1 Implementation of $\mathcal{D}_t^\gamma u$

The function `Du` calculates the value of $\mathcal{D}_t^\gamma u$ for $-5 \leq t \leq 5$.

Parameters list:

- `t`: the function's variable, between -5 and 5.
- `nt`: the number of subdivisions required to compute I_3.

Process:

- Calculate the first operand of $\mathcal{D}_t^\gamma u$ and store its value in `I1`.
- Calculate the second operand of $\mathcal{D}_t^\gamma u$ and store its value in `I2`.
- Calculate the third operand of $\mathcal{D}_t^\gamma u$, the one with the integral, and store its value in `I3`.

```c
double Du(double t, long int nt, double tmax, double p, double g) {

  double I1 = I_1(t, tmax, p, g);
  double I2 = I_2(t, tmax, p, g);
  double I3 = I_3(t, tmax, nt, p, g);

  return I1 + I2 - I3;

}
```

Listing 12: Implementation of the $\mathcal{D}_t^\gamma u$ function in C

10.1.2 Implementation of I_1, I_2, and I_3

I_1, I_2 and I_3 are used to implement respectively the first, second and third operands of $\mathcal{D}_t^\gamma u$

```c
//Implementation of I_1

double I_1(double t, double tmax, double p, double g) {

  return (pow(tmax, -g) / (tgamma(1 - g))) * (u(t-tmax, p) - u(t, p));

}

//Implementation of I_2

double I_2(double t, double tmax, double p, double g) {

  return (pow(tmax, 1 - g) / tgamma(2 - g)) * (u1(t-tmax, p));

}

//Implementation of I_3
```

```
18
19 double I_3(double t, double tmax, long int nt, double p, double g) {
20
21   return (1 / tgamma(2 - g)) * simpson(t, t-tmax, nt, p, g);
22
23 }
```

Listing 13: Implementation of I_1, I_2 and I_3 in C.

References

[1] Richard L Burden, Douglas J Faires, and Annette M Burden. *Numerical Analysis*. Brooks Cole, 10 edition, 2015.

[2] Y. Nec and M. J. Ward. Dynamics and stability of spike-type solutions to a one dimensional gierer-meinhardt model with sub-diffusion. *Physica D*, 241:947—963, 2012.