University of New South Wales

School of Mathematics and Statistics

Honours Thesis

Fractional Differential Equations

Author:
Adam J. Gray

Student Number:
3329798

Supervisor:
Dr Chris Tisdell

1 Numerical Methods for Fractional Differential Equations

In this section we describe numerical methods which can be used for solving fractional differential equations. We being by looking at two wasy for approximating a fractional derivative and then look at a specific scheme outlined by Diethlem [?].

1.1 Approximations of Fractional Derivatives and Integrals

We first recall that the Grünwald-Letnikov derivative of a function f is defined by

$$\left({}^{GL}_{\ a}\mathcal{D}^{\alpha}_{t} f\right)(t) = \lim_{h \longrightarrow 0} \frac{\left({}_{a}\Delta^{\alpha}_{h} f\right)(t)}{h^{\alpha}} \qquad \left({}_{a}\Delta^{\alpha}_{h} f\right)(t) = \sum_{j=0}^{\lfloor \frac{t-a}{h} \rfloor} (-1)^{j} \binom{\alpha}{j} f(t - jh)$$

For a very large class of functions this coincides with the Riemann-Liouville fractional derivative [?] which we again recall here

$$\left({}_{a}\mathcal{D}^{\alpha}_{t} f\right)(t) = \left(\frac{d}{dt}\right)^{\lfloor \alpha \rfloor + 1} \int_{a}^{t} (t - \tau)^{\lfloor \alpha \rfloor - \alpha} f(\tau) d\tau.$$

These two definitions suggest two general methods for approximating a fractional derivative.

1.1.1 Grünwald-Letnikov Approximation

In much the same was as we might use a finite difference method to approximate an integer order derivative we apply a similar approach here. By choosing some small, but non-zero value of h we can approximate the fractional derivative of a function by

$$\left(\widehat{{}^{GL}_{\ a}\mathcal{D}^{\alpha}_{t} f}\right)(t) = \frac{1}{h^{\alpha}} \sum_{j=0}^{n} (-1)^{j} \binom{\alpha}{j} f(t - jh). \tag{1}$$

It can be shown [?] that this yeilds a first order approximation of the Riemann-Liouville fractional derivative.

1.1.2 Quadrature Approximation

A completely different approach to take is to use a quadrature rule to evalute the integral from the Riemann-Liouville definition. If one is simply evaluating a derivative this method is not ideal because it also requires the approximation of an integer order derivative. If, however, we wish to calculate a fractional integral, such as might be required in a numerical FDE solver, then the quadrature method may be a more practical method.

There is no *set way* to do the quadrature approximation and it will all depend on the scheme being used.

In the following sections we will examine a general initial value problem and an Adams Moulton Bashforth scheme which is based of a quadrature method. [?]

1.1.3 An Initial Value Problem

Lets consider the initial value problem

$$\left({}^{C}_{0}\mathcal{D}^{\alpha}_{x} y\right) = f(x,y) \tag{2}$$

$$y^{(k)}(0) = y_0^{(k)} \tag{3}$$

for $0 \leq k < \lfloor \alpha \rfloor$. Note that this initial value problem is stated in terms of Caputo derivatives. The motivation for such intial value problems was discussed in [INSERT CC REF].

One of the most important things to note about this equation is that for non-integer values of α it is non-local, in the sense that the value of the solution at $y(x_0 + h)$ depends not only on $y(x_0)$ but also on $y(x), x \in [0, x_0]$ [?]. This is in contrast to ordinary differential equations and this fact is what makes fractional differential equations considerably more complex to solve. Even multistep methods which can be used to solve ordinary differential equations, only rely on some *fixed* number previous time steps. As a solution to a fractional differential equation progresses it relies on more and more previous time steps. As one might suspect this fundamentally increases the computational complexity of the schemes used to solve fractional differential equations as compared with the schemes used to solve traditional ordinary differential equations. [1]

1.1.4 The Adams Moulton Bashforth Scheme

We briefly outline a method explained and analysed in detail in [?]. As shown in section [INSERT CC REF] the initial value problem (??), (??) is equivelent to the Volterra equation.

$$y(x) = \sum_{k=0}^{\lceil \alpha \rceil - 1} y_0^{(k)} \frac{x^k}{k!} + \frac{1}{\Gamma(\alpha)} \int_0^x (x-t)^{\alpha-1} f(t, y(t)) dt \tag{4}$$

In order to approximate the solution to this integral equation over the interval $[0, T]$, we select a number of grid points N so that $h = T/N$ and $x_k = hk$ where $k \in \{0, 1, ..., N\}$.

We apply the approximation

$$\int_0^{x_{k+1}} (x_{k+1} - t)^{\alpha-1} g(t) dt \approx \int_0^{x_{k+1}} (x_{k+1} - t)^{\alpha-1} \tilde{g}_{k+1}(t) dt$$

where $g(t) = f(t, y(t))$ and $\tilde{g}_{k+1}(t)$ is the piecewise linear approximation of $g(t)$ with nodes at the gridpoints x_k. As outlined in [?] we can approximate the integral in (??) as

$$\int_0^{x_{k+1}} (x_{k+1} - t)^{\alpha-1} \tilde{g}_{k+1}(t) dt = \sum_{j=0}^{k+1} a_{j,k+1} g(x_j) \tag{5}$$

where

$$a_{j,k+1} = \frac{h^\alpha}{\alpha(\alpha+1)} \times \begin{cases} (k^{\alpha+1} - (k-\alpha)(k+1)^\alpha) & \text{if } j = 0 \\ ((k-j+2)^{\alpha+1} + (k-j)^{\alpha+1} - 2(k-j+1)^{\alpha+1}) & \text{if } 1 \leq j \leq k \\ 1 & \text{if } j = k+1. \end{cases} \tag{6}$$

[1]It is possible to reduce these computations to look at some large but fixed number of previous gridpoints but this results in a speed vs. accuracy tradeoff which will be discussed in section ??

By seperating the final term in the sum we can write

$$y_{k+1} = \sum_{j=0}^{\lceil \alpha \rceil - 1} y_0^{(j)} \frac{x_{k+1}^j}{j!} + \frac{1}{\Gamma(\alpha)} \left(\sum_{j=0}^{k} a_{j,k+1} f(x_j, y_j) + a_{k+1,k+1} f(x_{k+1}, y_{k+1}^P) \right) \tag{7}$$

where y_{k+1}^P is a *predicted* value for y_{k+1} which must be calculated due to the potential non-linearity of f [?].

This predicted value is calculated by taking a rectangle approximation to the integral in (**??**), to get

$$\int_0^{x_{k+1}} (x_{k+1} - t)^{\alpha - 1} g(t) dt \approx \sum_{j=0}^{k} b_{j,k+1} g(x_j) \tag{8}$$

where

$$b_{j,k+1} = \frac{h^\alpha}{\alpha} \left((k + 1 - j)^\alpha - (k - j)^\alpha \right). \tag{9}$$

and thus a predicted value of y_{k+1} can be calculated by

$$y_{k+1}^P = \sum_{j=0}^{\lceil \alpha \rceil - 1} \frac{x_{k+1}^j}{j!} y_0^{(j)} + \frac{1}{\Gamma(\alpha)} \sum_{j=0}^{k} b_{j,k+1} f(x_j, y_j). \tag{10}$$

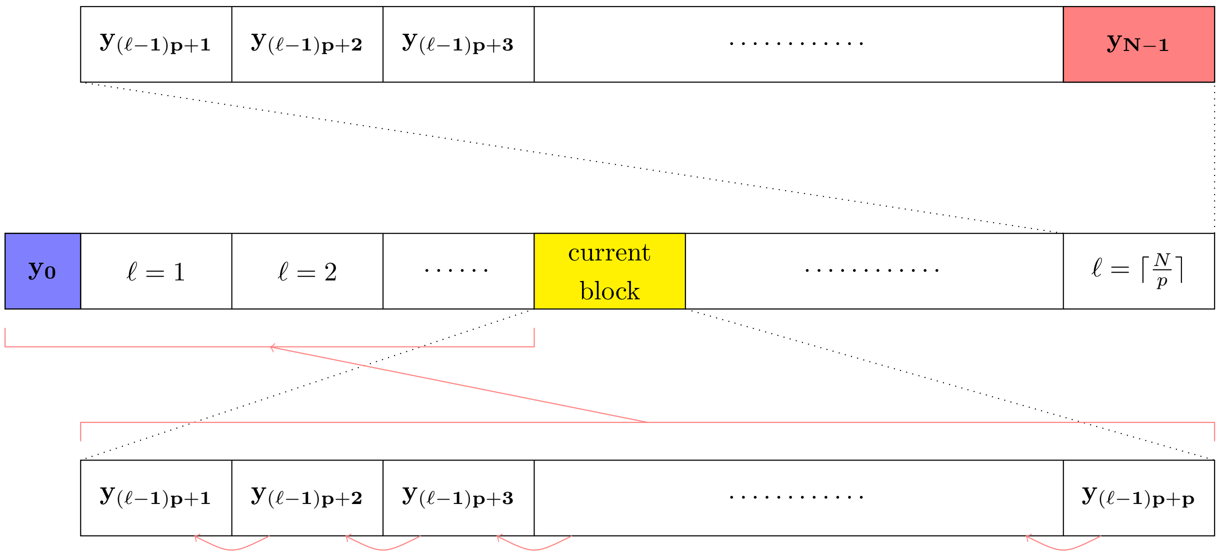
This outlines a fractional Adams Moulton Bashforth scheme. Of particular note are the sums which arise in (**??**) and (**??**). These sums do not arise in the integer order Adams Moulton Bashforth method. They arise as a result of the non-local nature of the fractional derivative [?]. These sums represent a significant computational cost as the number of terms grow linearly as the solution progress. This means in the fractional case the Adams Moulton Bashforth method has computational complexity $O(N^2)$ [?]. Section **??** will outline a task based parallel approach and section **??** will outline a massively parallel approach to solving (**??**), (**??**) with NVIDIA's CUDA.

1.1.5 Parallelizing: A Task Based Approach

Diethelm's paper [?] outlines a parallel method of numerically approximating the solution to (**??**), (**??**) by using a thread based parallel implementation of the Adams Moulton Bashforth method outlined in **??**. We base our approach in this section broadly on those of [?] but reformulate the scheme in terms of tasks instead of threads. This has a number of distinct benefits including scalability, especially from an implementation standpoint and clarity.

After we have setup the gridpoints $x_0, \ldots, x_{N-1}$ we create an array of solution values $y_0, \ldots, y_{N-1}$ which are to be populated with the calculated solution. From the initial conditions we can immediatly calculate y_0. We then break up the vector (array) $\mathbf{y}$ into blocks of size p. Suppose that $p = 2$ for example. Then the first block would contain y_1 and y_2. Each of the p variables in each block can be calculated almost entirely in parallel. There is some dependency between values in each block (i.e. y_2 depends on y_1) but this can be done after the bulk of the parallel computations have been completed.

To illustrate this idea we consider figure **??**. We have broken the vector (array) $\mathbf{y}$ up into $\ell = \lceil \frac{N}{p} \rceil$ blocks of p variables.



$\mathbf{y_{(\ell-1)p+1}}$
$\mathbf{y_{(\ell-1)p+2}}$
$\mathbf{y_{(\ell-1)p+3}}$
$\cdots\cdots\cdots\cdots$
$\mathbf{y_{N-1}}$
$\mathbf{y_0}$
$\ell = 1$
$\ell = 2$
$\cdots\cdots$
current block
$\cdots\cdots\cdots\cdots$
$\ell = \lceil \frac{N}{p} \rceil$
$\mathbf{y_{(\ell-1)p+1}}$
$\mathbf{y_{(\ell-1)p+2}}$
$\mathbf{y_{(\ell-1)p+3}}$
$\cdots\cdots\cdots\cdots$
$\mathbf{y_{(\ell-1)p+p}}$

Figure 1: Computation diagram for the values in the vector (array) $\mathbf{y}$.

The centre row shows how $\mathbf{y}$ is broken up into blocks, and the bottom row is a detailed view of the contents of the *current block* being computed.

The red lines indicate dependency, in that $\mathbf{y}_{(\ell-1)p+1}$ depends on all the y_j values calculated in the previous blocks. $\mathbf{y}_{(\ell-1)p+2}$ depends on all the y_j values calculated in previous blocks *and* on $\mathbf{y}_{(\ell-1)p+1}$.

The idea is that in each block we can do all of the computations which only depend on previous blocks in parallel and then perform the calculations which are dependent on other values within the same block in sequence. After all the values in a particular block are calculated we move on to the next block and repeat the process.

An exploded view of the last block is also provided to emphasise the fact that this block might not contain p variables if p does not divide $N - 1$. This fact causes us no bother as it is easy to handle in code.

As in [?] we rewrite the equations (??) and (??) as

$$y^P_{j+1} = I_{j+1} + h^\alpha H^P_{j,\ell} + h^\alpha L^P_{j,\ell} \tag{11}$$

and

$$y_{j+1} = I_{j+1} + h^\alpha H_{j,\ell} + h^\alpha L_{j,\ell} \tag{12}$$

where

$$I_{j+1} := \sum_{k=0}^{\lceil\alpha\rceil-1} \frac{x_{j+1}^k}{k!} y_0^{(k)} \tag{13}$$

$$H_{j,\ell}^P := \sum_{k=0}^{(\ell-1)p} b_{j-k} f(x_k, y_k) \tag{14}$$

$$L_{j,\ell}^P := \sum_{k=(\ell-1)p+1}^{j} b_{j-k} f(x_k, y_k) \tag{15}$$

$$H_{j,\ell} := c_j f(x_0, y_0) + \sum_{k=1}^{(\ell-1)p} a_{j-k} f(x_k, y_k) \tag{16}$$

$$L_{j,\ell} := \sum_{k=(\ell-1)p+1}^{j} aj - kf(x_k, y_k) + \frac{f(x_{j+1}, y_{j+1}^P)}{\Gamma(\alpha + 2.)} \tag{17}$$

In each block the values of H, I and H^P can be calculated in parallel for each y_j in the block.Then there is a somewhat more complex dependency between these sums.

To best explain how the processes procedes in each block and what dependencies there are we will consider a specific example where there are two values to be calculated per block (i.e. $p = 2$). Consider figure **??**.

Each box in the figure represents a task. A task can execute when all the tasks with arrows pointing to it have completed. The little red number represents the index of the variable *within* the block which is being calculated. All the task names are reasonablly self explanatory except for perhaps S^P and S. These take the values calculated in the other sums and add them together to get y^P and y respectivly. These are very small tasks which take very little time to execute.

The red arrow illustrate the interdependency of variables within a block. The second variable cannot have the sum L^P calculate until the first variable is calculated. This means that in effect each of the L tasks have to execute in series but these sums are relatively small so the time taken is quite short. Each block is then executed in sequence as each block depends on the blocks before it. See Appendix A for a C# implementation of this.

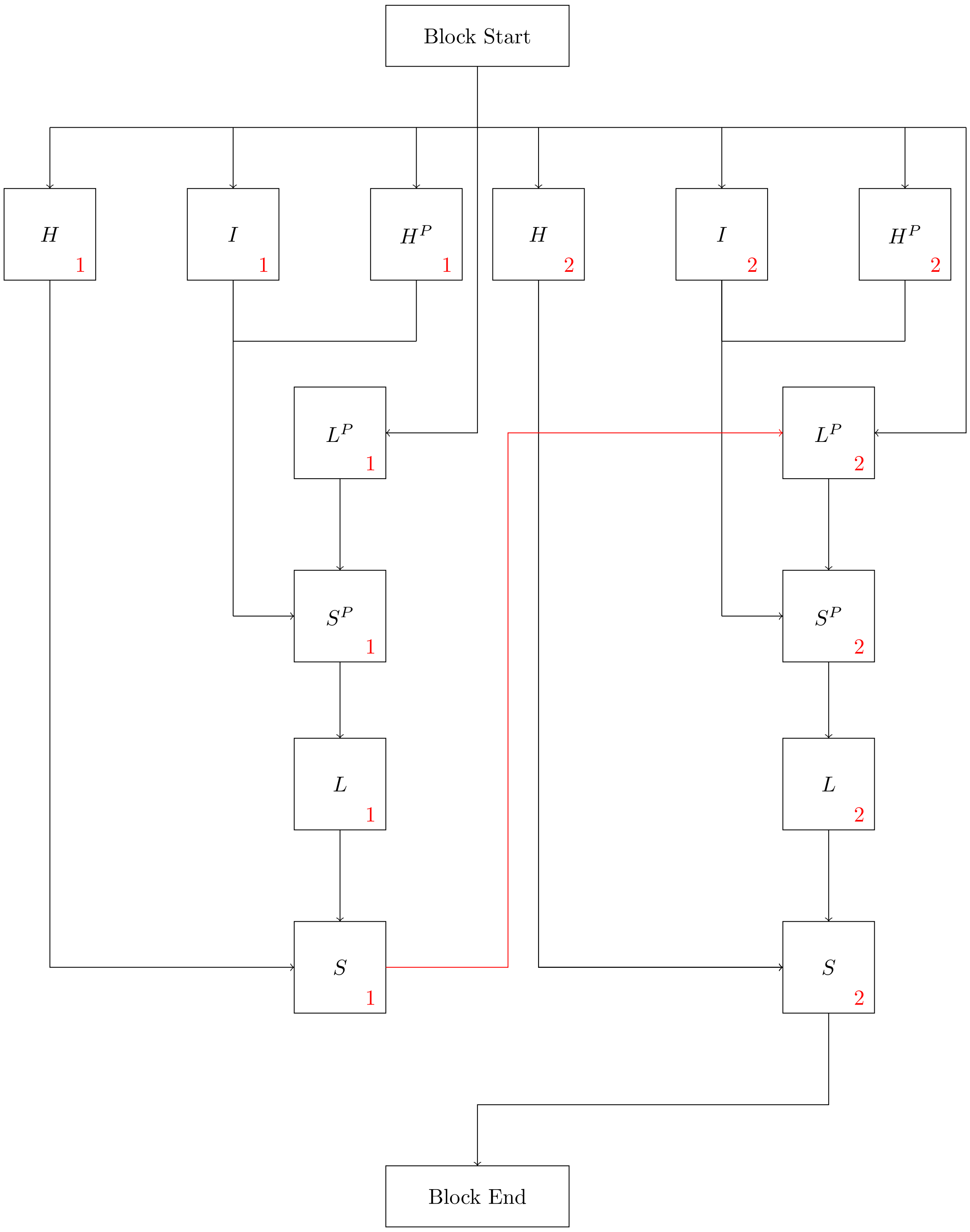


Figure 2: Task flow diagram for each block in the case when $p = 2$

1.1.6 Analysis of the Performance of the Scheme in C# and MONO

We wish to analyse the performance of this scheme both from the a theoretical standpoint and from at practical standpoint.

Principle 1 (Amdahl)**.**

We wish to analyse the performance of this scheme, and specifically its implementation in C# (MONO). Although this scheme is extremely similar to that outlined in [**?**] it differs mainly in the fact that this is a task based scheme which does not have an explicit number of threads which are executed, this decission is handed off to a task scheduler. In this scheme we let p denote the number of variables per block (and then spawn up to 4 simultaneous tasks per variable) instead of having p explicit threads for each block.

As can be seen from the code in Appendix A this results in easy to define dependencies and the bulk of the multithreading logic being done in just 40 lines of code.

To understand the performance characteristics of this scheme see figure **??** for a *performance surface* which shows the average runtime of the scheme for different values of p and N.

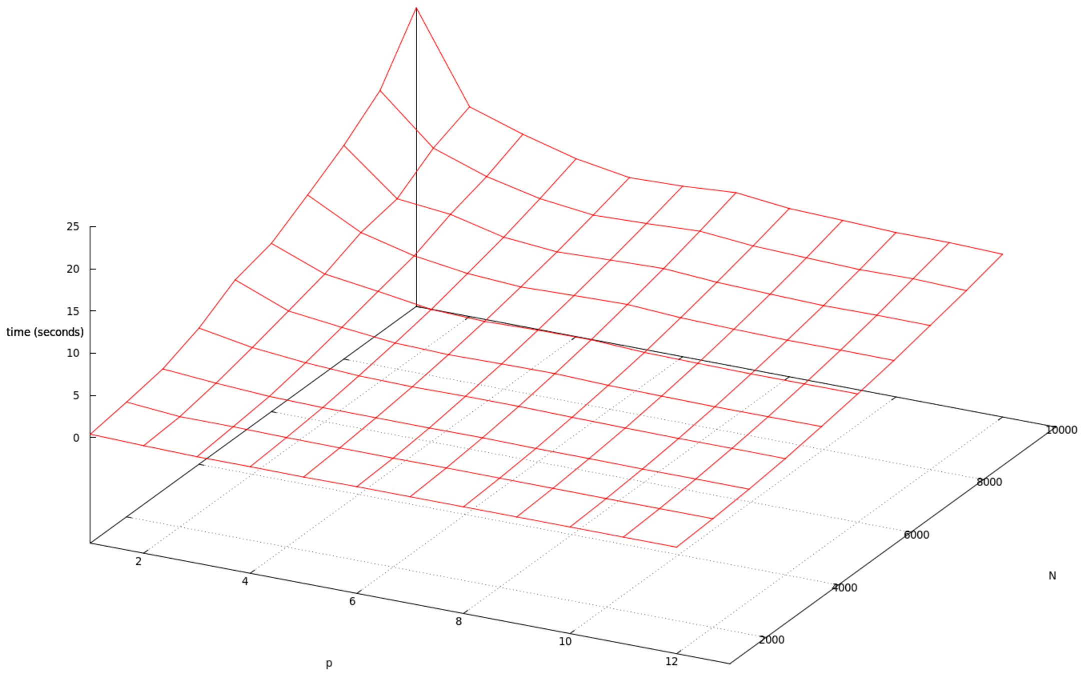


Figure 3: Average runtime of the scheme for different values of p and N

Firstly the $O(N^2)$ complexity of the algorithm is quite clearly visible, especially in the case where $p = 1$. Secondly the law of diminishing returns is also clearly visible in this plot. This affect can be attributed to Amdahl's law c.f. [**?**].

In the case where $p = 1$ it should be noted that there are up to 4 tasks running just for that one variable (I, L^P, H, H^P). In Diethelm's method, $p = 1$ would refer to only 1 thread running at all times, which is not the case here. While this limits the validity of any comparisons made with the method of Diethelm, it increases scalability and flexability.

1.1.7 Parallelizing: A Naive CUDA Approach

1.1.8 Reducing the Complexity Class: Speed - Accuracy Tradeoffs

2 Appendix A: Adams-Moulten-Bashford FDE Solver Code

2.1 C# Implementation of AMB FDE Solver

2.1.1 Program.cs

```
using System;
using System.Threading;
using System.Threading.Tasks;
using System.Collections.Generic;
using System.IO;
using System.Text;

namespace FDE_Solver
{
        class MainClass
        {
                /// <summary>
                /// This is the delegate use for the right hand side of the
                /// differential equation. It define the signature that the
                /// forcing function / right hand side must satisfy.
                /// </summary>
                public delegate double ForcingFunction (double x, double y);

                /// <summary>
                /// Main entry point of the program.
                /// </summary>
                /// <param name="args">The command-line arguments.</param>
                public static void Main (string[] args)
                {
                    double[] y = Compute (0.5, new double[] { 1 }, 10, 10000, 12, new
                        ForcingFunction (ff));
                }
                /// <summary>
                /// This is the RHS of the differential equation.
                /// For the purposes of demonstation this code is setup to
                /// solve D^{1/2}y = -y
                /// </summary>
                public static double ff(double x, double y)
                {
                        return -y;
                }

                /// <summary>
                /// This calculates the a coefficient described in the K. Diethelm paper
                /// referenced in the body of thesis.
                /// </summary>
                public static double a(int mu, double alpha)
                {
                        return (Math.Pow(mu + 2, alpha + 1) - 2 * Math.Pow(mu + 1, alpha + 1) +
                            Math.Pow(mu, alpha + 1))/SpecialFunctions.Gamma(alpha + 2);
                }

                /// <summary>
                /// This calculates the c coefficient described in the K. Diethelm paper
                /// referenced in the body of thesis.
                /// </summary>
                public static double c(int mu, double alpha)
                {
                        return (Math.Pow(mu, alpha+1) - (mu - alpha)*Math.Pow(mu + 1, alpha)) /
                            SpecialFunctions.Gamma(alpha + 2);
                }

                /// <summary>
                /// This calculates the b coefficient described in the K. Diethelm paper
                /// referenced in the body of thesis.
                /// </summary>
                public static double b(int mu, double alpha)
                {
                        return (Math.Pow (mu + 1, alpha) - Math.Pow (mu, alpha)) /
                            SpecialFunctions.Gamma (alpha + 1);
                }

                /// <summary>
                /// This calculates the sum I_{j+1} described in the K. Diethelm paper
                /// referenced in the body of thesis.
                /// </summary>
                public static double I_1(int j, double alpha, double[] y_0_diffs, double[] x)
                {
                        double value = 0;
                        for (int k = 0; k <= Math.Ceiling (alpha) - 1; k++) {
                                value += Math.Pow (x [j + 1], k) / SpecialFunctions.Factorial (k) *
                                    y_0_diffs [k];
                        }
                        return value;
                }

                /// <summary>
                /// This calculates the sum H^p_{j,\ell} described in the K. Diethelm paper
                /// referenced in the body of thesis.
                /// </summary>
                public static double H_p(int j, int ell, int p, double[] x, double[] y, double
                    alpha, ForcingFunction f)
                {
                        double value = 0;
                        for (int k = 0; k <= (ell - 1) * p; k++) {
                                value += b (j - k, alpha) * f (x [k], y [k]);
                        }
                        return value;
                }
                /// <summary>
                /// This calculates the sum L^p_{j,\ell} described in the K. Diethelm paper
                /// referenced in the body of thesis.
                /// </summary>
                public static double L_p(int j, int ell, int p, double[] x, double[] y, double
                    alpha, ForcingFunction f)
                {
                        double value = 0;
                        for (int k = (ell - 1) * p + 1; k <= j; k++) {
                                value += b (j - k, alpha) * f (x [k], y [k]);
                        }
                        return value;
                }

                /// <summary>
                /// This calculates the sum H^_{j,\ell} described in the K. Diethelm paper
                /// referenced in the body of thesis.
                /// </summary>
                public static double H(int j, int ell, int p, double[] x, double[] y, double alpha,
                    ForcingFunction f)
                {
                        double value = 0;
                        value += c (j, alpha) + f (x [0], y [0]);
                        for (int k = 1; k <= (ell - 1) * p; k++) {
                                value += a (j - k, alpha) * f (x [k], y [k]);
                        }
                        return value;
                }
                /// <summary>
                /// This calculates the sum L_{j,\ell} described in the K. Diethelm paper
                /// referenced in the body of thesis.
                /// </summary>
                public static double L(int j, int ell, int p, double[] x, double[] y, double alpha,
                    ForcingFunction f, double y_p_1)
                {
                        double value = 0;
                        for (int k = (ell - 1)*p + 1; k <= j; k++) {
                                value += a(j-k, alpha) * f(x[k], y[k]);
                        }
                        value += f(x[j+1], y_p_1) / SpecialFunctions.Gamma(alpha + 2);
                        return value;
                }
                /// <summary>
                /// This does the actual parallel computation for the method.
                /// This is done by setting up a series of tasks with a carefully defined
                /// continuation / dependency structure which ensures that computations which
                /// can run in parallel are allowed to, and ones which are dependent on other
                /// computations run in the right order. For a full description of the dependency
                    structure
                /// see the body of the thesis.
                /// </summary>
                /// <param name="alpha">The order of differentiation.</param>
                /// <param name="y_0_diffs">An array containing the initial conditions in order of
                    increasing
                /// differentiation order.</param>
                /// <param name="T">The last time to compute to.</param>
                /// <param name="N">The number of time steps to use.</param>
                /// <param name="p">Task granularity. This essentially defined the maximum level or
                    concurrency.</param>
                /// <param name="f">The right hand side of the differential equation.</param>
                public static double[] Compute(double alpha, double[] y_0_diffs, double T, int N,
                    int p, ForcingFunction f)
                {
                        double[] x = new double[N];
                        double[] y = new double[N];
                        double[] y_p = new double[N];
```

```
                    //Drops in the 0th order initial condition.
                    y [0] = y_0_diffs [0];
                    //Calculates the time step.
                    double h = T / N;
                    //Sets up all the x values.
                    for (int i = 0; i < N; i++)
                    {
                            x [i] = h * i;
                    }
                    //Compute each block
                    for (int ell = 1; ell <= Math.Ceiling ((double)N / (double)p); ell++) {
                            Task<double> taskSum_p = null;
                            Task<double> taskSum = null;
                            //Compute each variable in each block.
                            for (int i = 0; i < p && ((ell - 1) * p) + i < N - 1; i++) {
                                    //Calculate the j (index) for this variable.
                                    int j = ((ell - 1) * p) + i;
                                    //Setup the task dependency structure and set each task
                                        running.
                                    Task<double> taskI = Task.Factory.StartNew (() => I_1 (j,
                                        alpha, y_0_diffs, x));

                                    Task<double> taskH_p = Task.Factory.StartNew (() => H_p (j,
                                        ell, p, x, y, alpha, f));
                                    Task<double> taskH = Task.Factory.StartNew (() => H (j, ell,
                                        p, x, y, alpha, f));
                                    Task<double> taskL_p = null;
                                    if (taskSum != null) {
                                            taskL_p = taskSum.ContinueWith ((t) => L_p (j, ell, p,
                                                x, y, alpha, f));
                                    } else {
                                            taskL_p = Task.Factory.StartNew (() => L_p (j, ell, p,
                                                x, y, alpha, f));
                                    }
                                    taskSum_p = Task.Factory.ContinueWhenAll(new [] { taskL_p,
                                        taskH_p, taskI }, (ts) => y_p[j + 1] = taskI.Result +
                                        Math.Pow(h, alpha) * ( taskH_p.Result + taskL_p.Result ) );
                                    Task<double> taskL = taskSum_p.ContinueWith ((t) => L (j, ell,
                                        p, x, y, alpha, f, y_p [j + 1]));
                                    taskSum = Task.Factory.ContinueWhenAll(new [] { taskH, taskL,
                                        taskI }, (ts) => y[j+1] = taskI.Result + Math.Pow(h,
                                        alpha) * (taskH.Result + taskL.Result ));
                            }
                            // Wait for the block to complete.
                            if (taskSum != null) {
                                    taskSum.Wait ();
                            }
                    }
                    //Return the solution.
                    return y;
            }


    }
}
```

2.1.2 SpecialFunctions.cs

```
using System;

namespace FDE_Solver
{
	/// <summary>
	/// Provides special functions that are not available
	/// in System.Math
	/// </summary>
	public class SpecialFunctions
	{
		/// <summary>
		/// Gamma the specified z.
		/// This uses the Lanczos approximation and is only valid
		/// for positive real values of z. This code is essentially
		/// a translation of a python implementation available
		/// at http://en.wikipedia.org/wiki/Lanczos_approximation
		/// on 22nd July 2014
		/// </summary>
		/// <param name="z">The z value.</param>
		public static double Gamma(double z)
		{
			double g = 7;
			double[] p = new double[] { 0.99999999999980993, 676.5203681218851,
			    -1259.1392167224028,
			                                               771.32342877765313,
			                                                   -176.61502916214059,
			                                                   12.507343278686905,
			       -0.13857109526572012, 9.9843695780195716e-6, 1.5056327351493116e-7 };
			if (z < 0.5) {
				return Math.PI / (Math.Sin (Math.PI * z) * Gamma (1 - z));
			} else {
				z -= 1;
				double x = p [0];
				for (int i = 1; i < g + 2; i++)
				{
					x += p [i] / (z + i);
				}
				double t = z + g + 0.5;
				return Math.Sqrt (2 * Math.PI) * Math.Pow (t, z + 0.5) * Math.Exp
				    (-t) * x;
			}
		}
		/// <summary>
		/// Calculates the factorial of k.
		/// One could use the gamma function above but it does have slight inaccuracies
		/// so the factorial function has also been provided which returns an integer.
		/// </summary>
		/// <param name="k">The value to take the factorial of.</param>
		public static int Factorial(int k)
		{
			int value = 1;
			for (int i = 1; i <= k; i++) {
				value *= i;
			}
```

```
                        return value;
                }
        }
}
```

2.2 CUDA C/C++ Implementation of AMB FDE Solver

2.2.1 FDE_Solver.cu

```
#include <iostream>
#include <ctime>

// This is the size of the memory that we will have in each thread block
// Currently thread blocks can be at most 1024 in size so we allocate double
// that for good measure.
#define SHARED_MEMORY_SIZE 2048

using namespace std;

// The RHS of the differential equation
__device__ float f(float x, float y) {
    return -y;
}

// This uses the Lanczos approximation and is only valid
// for positive real values of z. This code is essentially
// a translation of a python implementation available
// at http://en.wikipedia.org/wiki/Lanczos_approximation
// on 22nd July 2014
__device__ float Gamma(float z) {
    float g = 7;
    float p [] = { 0.99999999999980993, 676.5203681218851, -1259.1392167224028,
                               771.32342877765313, -176.61502916214059, 12.507343278686905,
                               -0.13857109526572012, 9.9843695780195716e-6, 1.5056327351493116e-7
                                   };
        float result = 0;
        if (z < 0.5) {
            result = M_PI / (sin(M_PI * z) * Gamma(1-z));
        } else {
            z -= 1;
            float x = p[0];
            for (int i = 1; i < g + 2; i++) {
                x += p[i] / (z +i);
            }
            float t = z + g + 0.5;
            result = sqrt(2 * M_PI) * pow(t, z + 0.5f) * exp(-t) * x;
        }
        return result;
}

// Calculates the factorial of an integer
__device__ int Factorial(int k) {
    int value = 1;
    for (int i = 1; i <= k; i++) {
        value *= i;
    }
    return value;
}

// This calculates the a coefficient described in the K. Diethelm paper
// referenced in the body of thesis.
__device__ float a(int mu, float alpha) {
    return (pow((float)mu + 2, alpha + 1) - 2 * pow((float)mu + 1, alpha + 1) + pow((float)mu,
         alpha + 1))/Gamma(alpha + 2);
}

// This calculates the b coefficient described in the K. Diethelm paper
// referenced in the body of thesis.
__device__ float b(int mu, float alpha) {
    return (pow ((float)mu + 1, alpha) - pow ((float)mu, alpha)) / Gamma (alpha + 1);
}

// This calculates the c coefficient described in the K. Diethelm paper
// referenced in the body of thesis.
__device__ float c(int mu, float alpha) {
    return (pow((float)mu, alpha+1) - (mu - alpha)*pow((float)mu + 1, alpha)) / Gamma(alpha + 2);
}

// This calculates the sum R in parallel on the GPU. The sum R is discussed in the body of the
// thesis.
__global__ void calcR(int j, float* x, float* y, float alpha, int opsPerThread, float* R) {
    __shared__ float temp[SHARED_MEMORY_SIZE];
    int i                          = (blockDim.x * blockIdx.x + threadIdx.x) * opsPerThread;
    float sum              = 0;
        for (int l = 0; l < opsPerThread && i + l <= j; ++l) {
                sum                += b(j - (i + l), alpha) * f(x[i+l], y[i+l]);
        }
        temp[threadIdx.x] = sum;
        __syncthreads();
    if ( 0 == threadIdx.x ) {
        sum            = 0;
        for (int k     = 0; k < blockDim.x; ++k) {
            sum        += temp[k];
        }
        atomicAdd( &R[j], sum );
    }
}


// This calculates the sum S in parallel on the GPU. The sum R is discussed in the body of the
// thesis.
__global__ void calcS(int j, float* x, float* y, float alpha, int opsPerThread, float* S) {
    __shared__ float temp[SHARED_MEMORY_SIZE];
    int i                          = (blockDim.x * blockIdx.x + threadIdx.x) * opsPerThread;
    float sum              = 0;
        for (int l = 0; l < opsPerThread && i + l <= j; ++l) {
                sum                += a(j - (i + l), alpha) * f(x[i+l], y[i+l]);
        }
        temp[threadIdx.x] = sum;
        __syncthreads();
    if ( 0 == threadIdx.x ) {
        sum            = 0;
        for (int k     = 0; k < blockDim.x; ++k) {
            sum        += temp[k];
        }
        atomicAdd( &S[j], sum );
    }
}

// This is used to initialize the very first value in the solution array
// from the initial conditions.
__global__ void setY0(float* y, float y0) {
    y[0] = y0;
}

// This is used to initialize the x array with the correct grid points.
// This will execute in parallel
__global__ void initialize_x(float* x, float h, int N, int opsPerThread) {
    int j = (blockIdx.x * blockDim.x + threadIdx.x) * opsPerThread;
    for (int i = 0; i < opsPerThread && i + j < N; i++) {
        x[j + i] = (j + i) * h;
    }
}

// This calculate the predictor value from the sums already calculated.
// This does not run in parallel and should be called with <<< 1, 1 >>>
__global__ void calcYp(int j, float* I, float* R, float* Yp, float h, float alpha) {
    Yp[j+1] = I[j+1] + pow(h, alpha) * R[j];
}

// This calculate the y value from the sums already calculated.
// This does not run in parallel and should be called with <<< 1, 1 >>>
__global__ void calcY(int j, float* I, float* S, float* Yp, float* y, float* x, float h, float
     alpha) {
    y[j+1] = I[j+1] + pow(h, alpha) * ( c(j, alpha) * f(x[0], y[0]) + S[j] + f(x[j+1], Yp[j+1]) /
        Gamma(alpha + 2) );
}


// This calculates the I sum discussed in the body of the thesis. The entirety of
// the I sums can be precalculated and so that is what this does. This does run
// in parallel with as many threads as there are steps.
__global__ void initialize_I(float* I, float* x, float* y_0_diffs, float alpha, int N) {
        float sum = 0;
        int j = blockIdx.x * blockDim.x + threadIdx.x;
    if (j < N) {
        for (int k = 0; k <= (ceil(alpha) - 1); k++) {
                        sum += pow(x[j], (float)k) / Factorial(k) * y_0_diffs[k];
                }
                I[j] = sum;
    }
}

// This sets up the actual computations on the graphics card and launches them.
// N                  - Number of steps
// T                  - Maximum value of x
// y_0_diffs   - The initial conditions
// y                  - The vector into which the solution should be copied. (This needs to be
//                              malloced out by the caller of this function.
// opsPerThread - Number of operations per thread to actually perform.
void compute(int N, float alpha, float T, float* y_0_diffs, float* y, int opsPerThread){
       float*               dev_x                       = NULL;
       float*               dev_y                       = NULL;
       float*               dev_R                       = NULL;
       float*               dev_S                       = NULL;
       float*               dev_y_0_diffs        = NULL;
       float*               dev_y_p                     = NULL;
       float*               dev_I                       = NULL;
       cudaDeviceProp* props                    = NULL;
       float                h                                  = 0;
       int                  size                       = 0;
       int                  size_ics                   = 0;
       int                  blocks                     = 0;
       float                blocksf                    = 0;
       int                  threads                    = 0;
       float                threadsf                   = 0;


       // The amount of space needed for the main arrays
       size                 = sizeof(float) * N;

       // The amount of space needed for the initial conditions
       size_ics             = sizeof(float) * ceil(alpha);

       // Allocate memory on device for computations
       cudaMalloc( (void**)&dev_x, size );
       cudaMalloc( (void**)&dev_y, size );
       cudaMalloc( (void**)&dev_R, size );
       cudaMalloc( (void**)&dev_S, size );
       cudaMalloc( (void**)&dev_I, size );
       cudaMalloc( (void**)&dev_y_p, size );
       cudaMalloc( (void**)&dev_y_0_diffs, size_ics );

       // Set the device memory up so that it is otherwise set to 0
       cudaMemset( dev_y, 0, size );
       cudaMemset( dev_y_p, 0, size );
       cudaMemset( dev_R, 0, size );
       cudaMemset( dev_S, 0, size );

       // Move the 0th order initial condition across.
       setY0<<< 1, 1 >>>( dev_y, y_0_diffs[0] );

       // Move the initial conditions across.
       cudaMemcpy( dev_y_0_diffs, y_0_diffs, size_ics, cudaMemcpyHostToDevice );

       // Query device capabilities.
       props                = (cudaDeviceProp*) malloc( sizeof(cudaDeviceProp) );
       cudaGetDeviceProperties( props, 0 ); // Assume that we will always use device 1.

       // initialize the x vector
```

```
	h                            = T / N;
	blocksf               = (float) N / (float) (props->maxThreadsPerBlock * opsPerThread);
	blocks                = ceil(blocksf);
	threads               = props->maxThreadsPerBlock;
	if (1 == blocks) {
		threadsf      = (float) N / (float)opsPerThread;
		threads       = ceil(threadsf);
	}
	initialize_x<<< blocks, threads >>>(dev_x, h, N, opsPerThread);

	// initialize the I vector
	blocksf               = (float) N / (float) props->maxThreadsPerBlock;
	blocks                = ceil(blocksf);
	if (1 == blocks) {
		threads       = N;
	} else {
		threads       = props->maxThreadsPerBlock;
	}
	initialize_I<<< blocks, threads >>>(dev_I, dev_x, dev_y_0_diffs, alpha, N);

	// Perform the actual calculations.
	for (int j = 0; j < N; ++j) {
		blocksf               = (float) ( j + 1 )/ (float) (props->maxThreadsPerBlock *
		    opsPerThread);
		blocks                = ceil(blocksf);
		if (1 == blocks) {
			threadsf      = (float) j / (float)opsPerThread;
			threads       = ceil(threadsf) + 1;
		} else {
			threads = props->maxThreadsPerBlock;
		}
		calcR<<< blocks, threads >>>(j, dev_x, dev_y, alpha, opsPerThread, dev_R);
		calcYp<<< 1, 1 >>>(j, dev_I, dev_R, dev_y_p, h, alpha);
		calcS<<< blocks, threads >>>(j, dev_x, dev_y, alpha, opsPerThread, dev_S);
		calcY<<< 1, 1 >>>(j, dev_I, dev_S, dev_y_p, dev_y, dev_x, h, alpha);
	}

	// Copy y vector back onto host memory
	cudaMemcpy( y, dev_R, size, cudaMemcpyDeviceToHost );

	// Free up all resources that were used.
	cudaFree( dev_x );
	cudaFree( dev_y );
	cudaFree( dev_R );
	cudaFree( dev_S );
	cudaFree( dev_I );
	cudaFree( dev_y_0_diffs );
	cudaFree( dev_y_p );

	// Free up all space used on the host.
	free( props );
}

int main( void ) {
	// This code will benchmark the performance of this solver with different
	// paramters for the number of steps and number of operations per thread.
```

```
        float      y_0_diffs[]              = { 1 };
        int        N                               = 10000;
        int        T                               = 10;
        clock_t    start             = 0;
        clock_t    end               = 0;
        double     duration          = 0;
        for (int i = 0; i <= 50; i+=5) {
                // Need to make sure that the correct ammount of memory is available.
                int        size                        = N * i * sizeof( float );
                float*     y                       = (float*) malloc( size );
            for (int j = 1; j <= 20; ++j) {
                start          = clock();
                compute(N * i, 0.5f, T, y_0_diffs, y, j );
                end            = clock();
                duration       = (end - start) / (double) CLOCKS_PER_SEC;
                cout << duration << "\t" << flush;
            }
            free(y);
            cout << "\n";
        }
    return 0;
}
```

References

[1] K. Diethelm. *The Analysis of Fractional Differential Equations*. Springer, 2010.

[2] K. Diethelm. An efficient parallel algorithm for the numerical solution of fractional differential equations. *Fractional Calculus and Applied Analysis*, 14:475–490, 2011.

[3] K. Diethelm, N.J. Ford, and A.D. Freed. Detailed error analysis for a fractional adams method. *Numerical Algorithms*, pages 31–52, 2004.

[4] I. Podlubny. *Fractional Differential Equations*. Academic Press, 1999.