UNSW AUSTRALIA

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

Although we have focused on fractional differential equations which involved Riemann-Liouviille derivatives or Caputo derivatives up until this point, it turns out that in many cases the Grünwald-Letnikov definition is equivelent [3]. This is a useful fact because the Grünwald-Letnikov definition can be used for numerical evaluation of fractional derivatives.

To do this we first recal the definition

$$\left({}_a\mathcal{D}_t^\alpha f\right)(t) = \lim_{h\longrightarrow 0} \frac{\left({}_a\Delta_h^\alpha f\right)(t)}{h^\alpha}$$

where

$$\left({}_a\Delta_h^\alpha f\right)(t) = \sum_{j=0}^{\lfloor \frac{t-a}{h} \rfloor} (-1)^j \binom{\alpha}{j} f(t-jh).$$

By essentially choosing some small value of h we can arrive at a reasonable numerical approximation of the Riemann-Liouville fractional derivative of a function f [3].

We can also use quadrature rules to numerically fractionally integrate and differentiate functions [3]. We will, however, not investigate that approach further and instead fucus on this finite differences idea outlined above.

One of the things which sould become immediatly apparent from the above definitions is that at t goes far away from a the number of terms in the sum which defines $\left({}_a\Delta_h^\alpha f\right)(t)$ grows with $O(t-a)$. This is not the case for integer order derivatives and this particular fact introduces numerous problems which we will attempt to solve or mitigate in the following sections.

1.1 A Parallel Adams-Moulton-Bashforth FDE Solver via Modern Task Based Concurrency

In this section we aim to take the work of Diethelm in [1] and apply modern task based concurrency approaches to improve performance and scalability. We do this first via an implementation in MONO (.NET) with C# and then gain performance increases by using C++ and Intel$^{®}$'s Threading Building Blocks and then gain collosal performance increases by utilising NVIDIA$^{®}$'s CUDA$^{®}$.

These performance increases can mitigate the issue that arose from the growth in the number of terms of the sum that defined $\left({}_a\Delta_h^\alpha f\right)(t)$ but they do not fundamentally change the complexity class of the algorithms discussed.

1.1.1 Explanation of the Adams-Moulton-Bashforth Method

The method that we explain here is designed to solve fractional differential equations of the form,

$$\left({}_0^C\mathcal{D}_x^\alpha y\right)(x) = f(x, y) \tag{1}$$

along with initial conditions

$$y(0) = y_0, y'(0) = y_0^{(1)}, \ldots, y^{(\lceil\alpha\rceil)}(0) = y_0^{(\lceil\alpha\rceil)}. \tag{2}$$

We use an Adams-Moulton-Bashford method described in [1]. Firstly we need to choose some time T for the interval $[0, T]$ over which we will solve the differential equation and some number of steps to use N. This allows us to define a step size $h := T/N$ and therefore the gridpoints $x_j = jh$.

From that we can calculate the predictor approximation

$$y_{j+1}^P = \sum_{k=0}^{\lceil\alpha\rceil-1} \frac{x_{j+1}^k}{k!} y_0^{(k)} + h^\alpha \sum_{k=0}^{j} b_{j-k} f(x_k, y_k) \tag{3}$$

with coefficients b_μ given by

$$b_\mu = \frac{(\mu+1)^\alpha - \mu^\alpha}{\Gamma(\alpha+1)}. \tag{4}$$

We then calculate the actual approximation of y_{j+1} with

$$y_{j+1} = \sum_{k=0}^{\lceil\alpha\rceil-1} \frac{x_{j+1}^k}{k!} y_0^{(k)} + h^\alpha \left(c_j f(x_0, y_0) + \sum_{k=1}^{j} a_{j-k} f(x_k, y_k) + \frac{f(x_{j+1}, y_{j+1}^P)}{\Gamma(\alpha+2)} \right) \tag{5}$$

with coefficients a_μ and c_μ given by

$$a_\mu = \frac{(\mu+2)^{\alpha+1} - 2(\mu+1)^{\alpha+1} + \mu^{\alpha+1}}{\Gamma(\alpha+2)} \tag{6}$$

and

$$c_\mu = \frac{u^{\alpha+1} - (\mu-\alpha)(\mu+1)^\alpha}{\Gamma(\alpha+2)}. \tag{7}$$

As discussed in [2] and [1] this algorithm converges in the sense that

$$\max_{j=0,1,\ldots,N} |y(x_j) - y_j| = \begin{cases} O(h^{1+\alpha}) & 0 < \alpha < 1 \\ O(h^2) & \alpha \geq 1 \end{cases} \tag{8}$$

under very unrestrictive conditions.

Diethelm outlines a method of multithreading the Adams-Moulton-Bashford scheme in [1] and we will base the main ideas for our method on those. The method developed by Diethelm use MPI (message passing interface). We seek to recast these ideas in the context of a task based paradigm. The benefit of this method is that it makes the code more readable, greatly simplifies synchronisation and potentially makes the code more scalable.

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 the following diagram. We have broken the vector (array) $\mathbf{y}$ up into $\ell = \lceil \frac{N}{p} \rceil$ blocks of p variables. The centre row shows how $\mathbf{y}$ is broken up into blocks, and the bottom row is

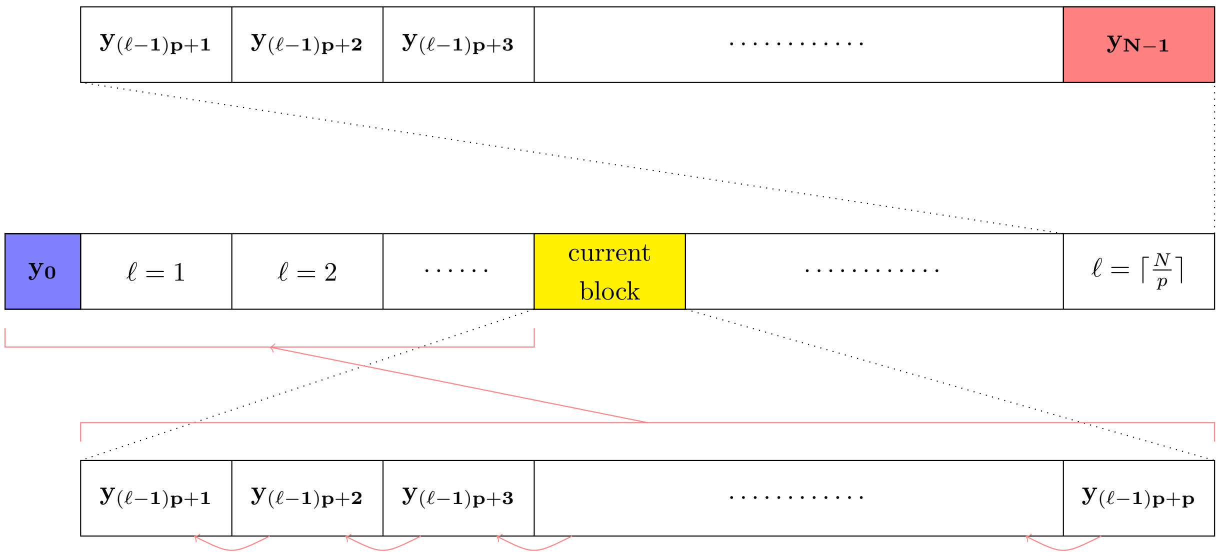
$\mathbf{y}_{(\ell-1)p+1}$
$\mathbf{y}_{(\ell-1)p+2}$
$\mathbf{y}_{(\ell-1)p+3}$
....
$\mathbf{y}_{N-1}$
$\mathbf{y}_0$
$\ell = 1$
$\ell = 2$
......
current block
....
$\ell = \lceil \frac{N}{p} \rceil$
$\mathbf{y}_{(\ell-1)p+1}$
$\mathbf{y}_{(\ell-1)p+2}$
$\mathbf{y}_{(\ell-1)p+3}$
....
$\mathbf{y}_{(\ell-1)p+p}$

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

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 [1] we rewrite the sums (3) and (4) as

$$y^P_{j+1} = I_{j+1} + h^\alpha H^P_{j,\ell} + h^\alpha L^P_{j,\ell} \tag{9}$$

and

$$y_{j+1} = I_{j+1} + h^\alpha H_{j,\ell} + h^\alpha L_{j,\ell} \tag{10}$$

where

$$I_{j+1} := \sum_{k=0}^{\lceil\alpha\rceil-1} \frac{x_{j+1}^k}{k!} y_0^{(k)} \tag{11}$$

$$H_{j,\ell}^P := \sum_{k=0}^{(\ell-1)p} b_{j-k} f(x_k, y_k) \tag{12}$$

$$L_{j,\ell}^P := \sum_{k=(\ell-1)p+1}^{j} b_{j-k} f(x_k, y_k) \tag{13}$$

$$H_{j,\ell} := c_j f(x_0, y_0) + \sum_{k=1}^{(\ell-1)p} a_{j-k} f(x_k, y_k) \tag{14}$$

$$L_{j,\ell} := \sum_{k=(\ell-1)p+1}^{j} aj - kf(x_k, y_k) + \frac{f(x_{j+1}, y_{j+1}^P)}{\Gamma(\alpha+2.)} \tag{15}$$

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 2.

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.

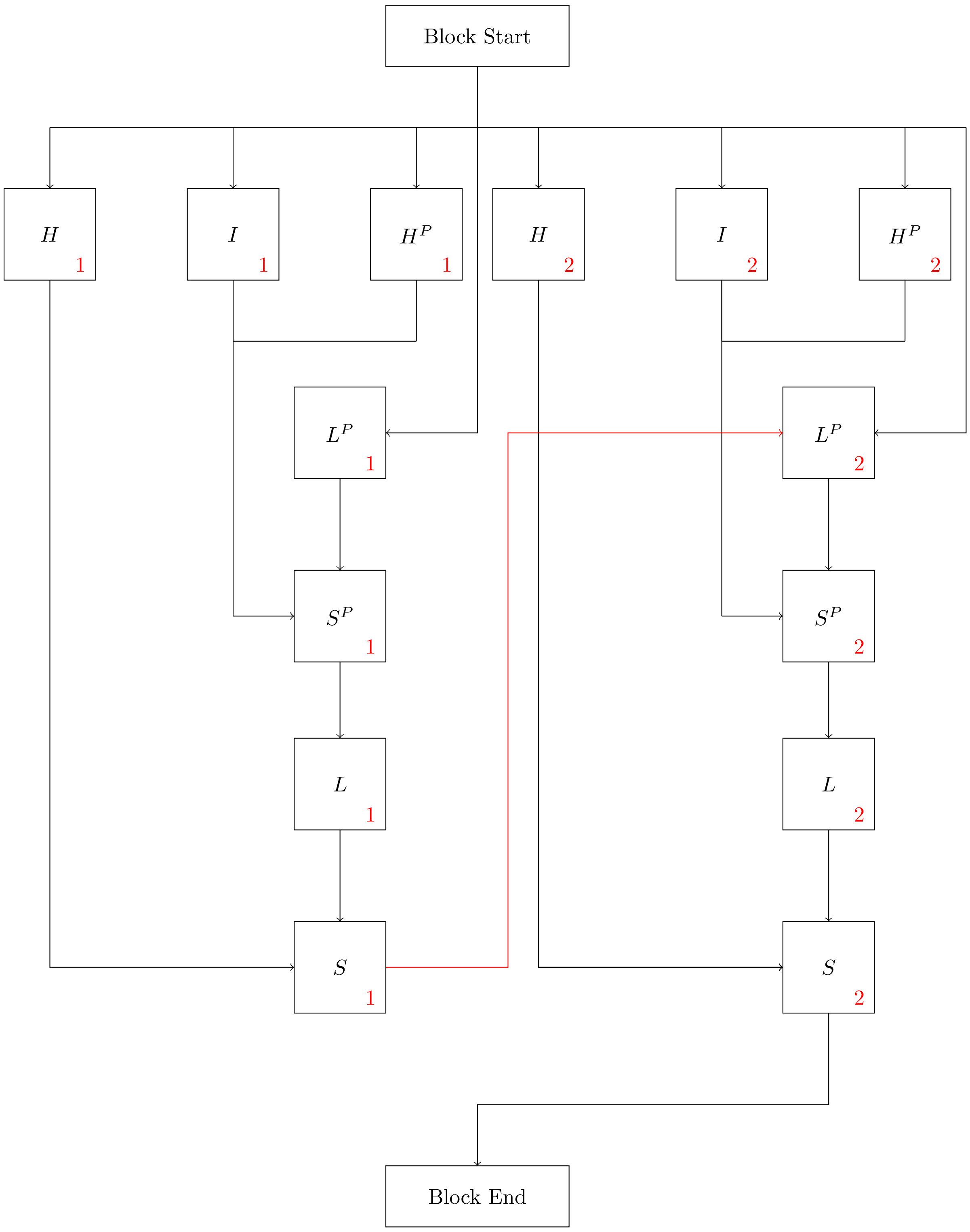

Figure 2: Task flow diagram for each block in the case when $p = 2$

1.1.2 Application of the Scheme to the Relaxation-Oscilation Equation

Before continuing to analyse the performance of this method or discuss possible improvements we pause for a moment of intelectual indulgence and apply this scheme to the relaxation-oscilation equation.

We define the (Caputo) relaxation-oscilation equation as

$$\left({}_0^C\mathcal{D}_\alpha^x y \right)(x) = -y(x) \tag{16}$$

for $0 < \alpha \leq 2$. along with initial conditions

$$y(0) = 1, y'(0) = 0, y''(0) = 0 \tag{17}$$

In the case where $\alpha = 1$ this results in the well known exponential solution and in the case where $\alpha = 2$ this results in the well known trigonometric solution.

In figure 3 we show a plot of all solutions as α ranges from $\frac{1}{2}$ to 2.

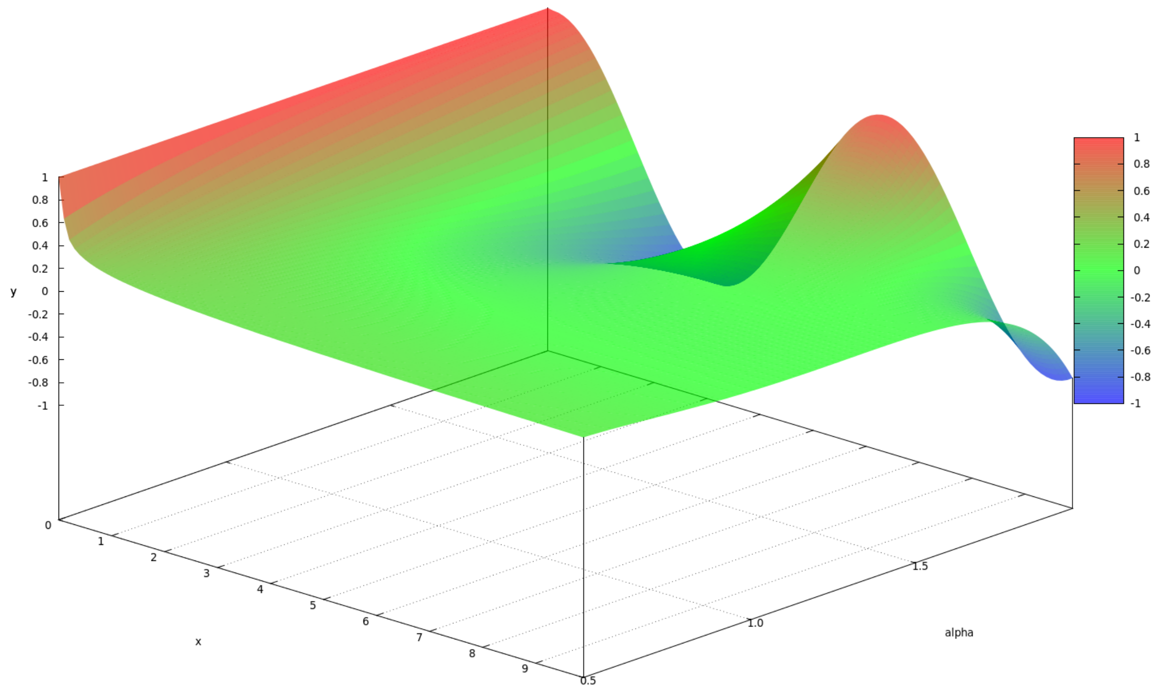

Figure 3: Solution surface of the relaxation-oscilation equation defined in (16) and (17).

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

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 [1] 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 is because we rely on a task scheduler to do that for us. 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.

A possible drawback of this scheme is that it relies on the a task scheduler which may or may not be optimised for the type of workload being placed on it. We will discuss this issue in more detail later on.

To understand the performance characteristics of this scheme see figure 4 for a *performance surface* which shows the average runtime of the scheme for different values of p and N.

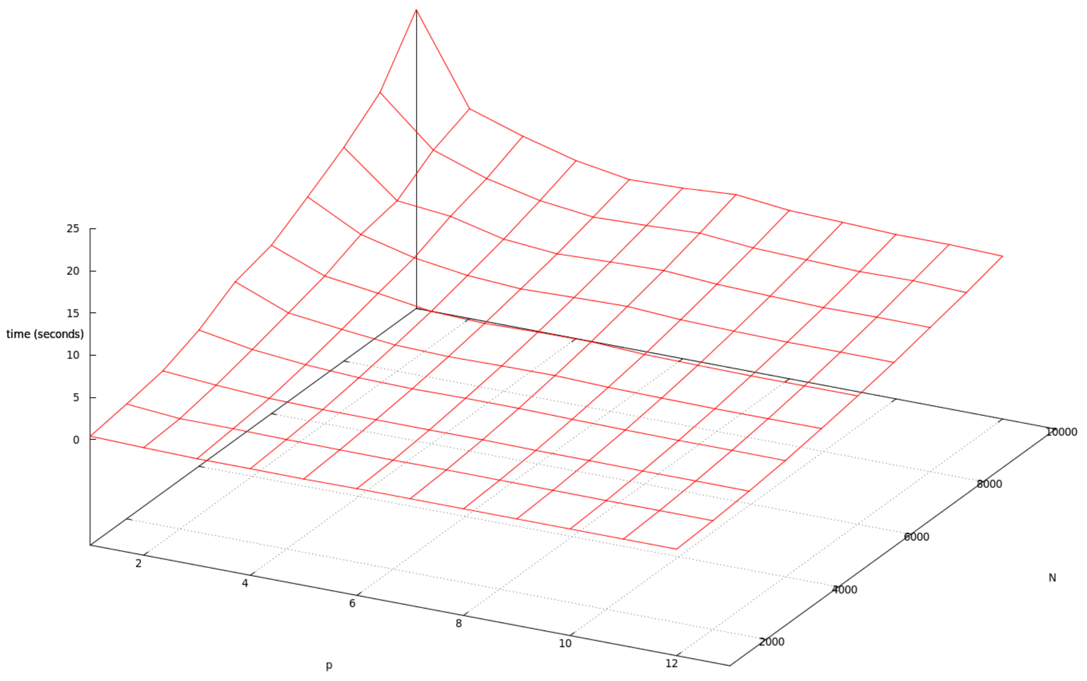

Figure 4: Average runtime of the scheme for different values of p and N

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. 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 [1] $p = 1$ would refer to only 1 thread running at all times, which is not the case here. This is has its advantages and disadvantages. Its clearly advantageous because it results in potentially higher concurrency but disadvantageous because it limits out ability to compare this scheme with the one in [1] on a *thread for thread* basis.

It should also be noted that setting large values for p (more than the number of cores available) should not be as bad as it would be in the case where we are dealing with raw threads. The task scheduler will decide what the maximum number of concurrent tasks should be an execute the tasks accordingly. In the case of the default scheduler it may still incur a performance hit because the scheduler may / will spend time task switching to ensure

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 = 1; 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. An efficient parallel algorithm for the numerical solution of fractional differential equations. *Fractional Calculus and Applied Analysis*, 14:475–490, 2011.

[2] K. Diethelm, N.J. Ford, and A.D. Freed. Detailed error analysis for a fractional adams method. *Numerical Algorithms*, 36:31–52, 2004.

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