University of New South Wales

School of Mathematics and Statistics

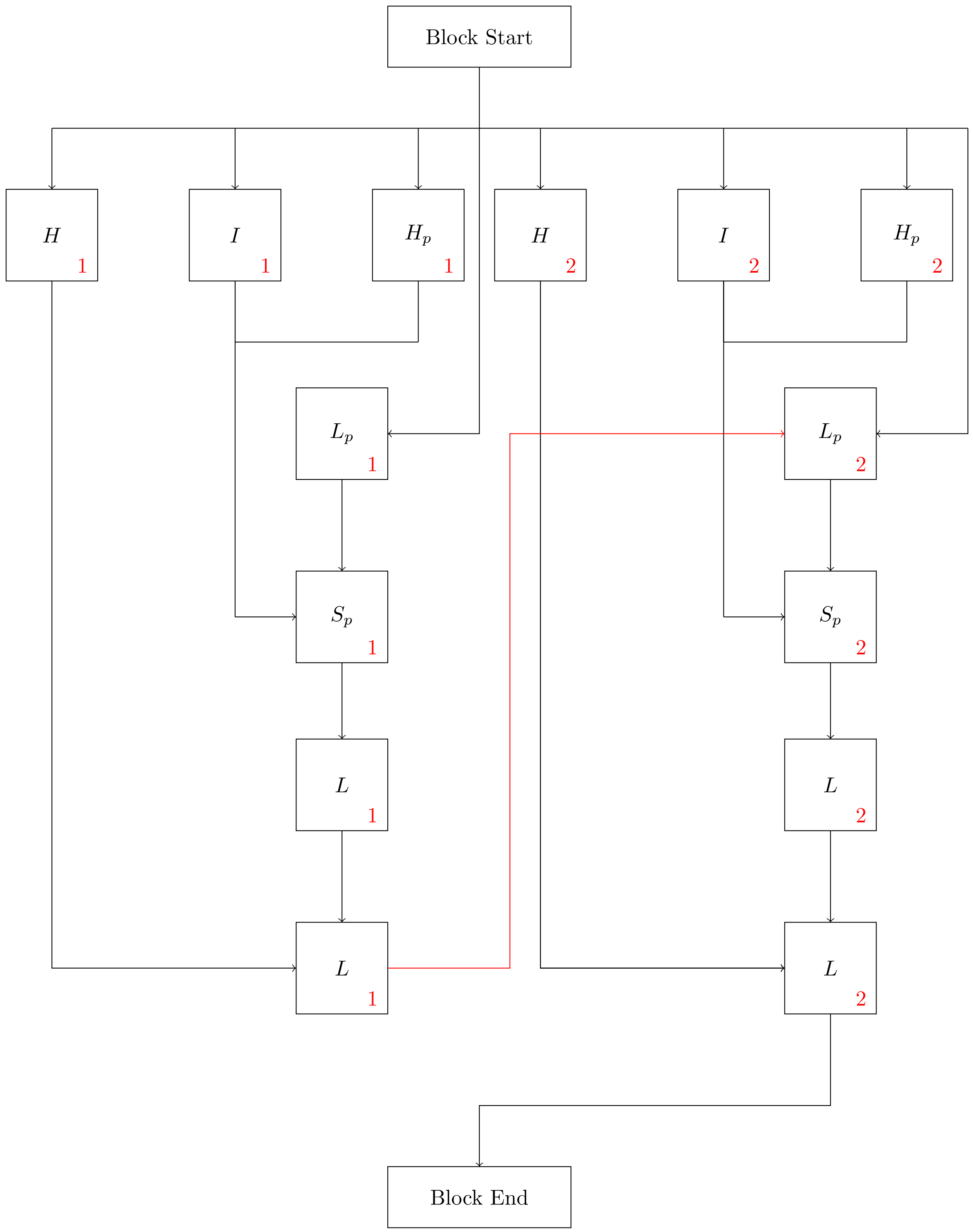
Honours Thesis

Fractional Differential Equations

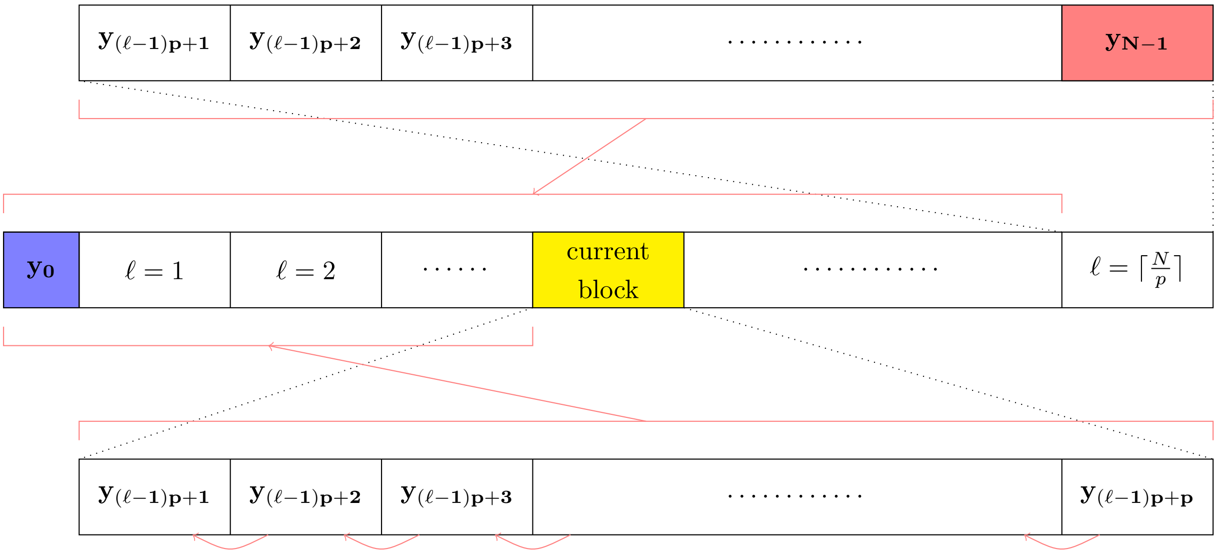
Author:
Adam J. Gray

Student Number:
3329798

Supervisor:
Dr Chris Tisdell



Block Start
H 1
I 1
H_p 1
H 2
I 2
H_p 2
L_p 1
L_p 2
S_p 1
S_p 2
L 1
L 2
L 1
L 2
Block End



1 Adams-Moulten-Bashford FDE Solver Code

1.1 C# Implementation of AMB FDE Solver

1.1.1 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;
        }


    }
}
```

1.1.2 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;
        }
    }
}
```