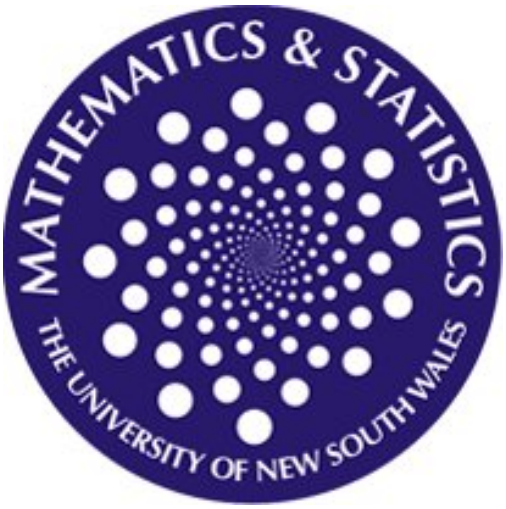
UNSW
AUSTRALIA



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 Analytical Results for Ordinary Fractional Differential Equations

In this section we establish some analytical results for ordinary fractional differential equations. These results allow us to solve some simple initial value problems and establish a theoretical framework from which results in other sections can draw.

1.1 Basic definitions

Definition 1 (Fractional Derivatives and Integrals)**.** *For $\alpha > 0$ we define*

$$(I^{\alpha}_{a+}f)(x) := \frac{1}{\Gamma(\alpha)}\int_a^x \frac{f(t)}{(x-t)^{1-\alpha}}dt$$
$$(\mathcal{D}^{\alpha}_{a+}f)(x) := \frac{1}{\Gamma(n-\alpha)}\frac{d^n}{dx^n}\int_a^x \frac{f(t)}{(x-t)^{\alpha-n+1}}dt$$
$$({}^{C}\mathcal{D}^{\alpha}_{a+}f)(x) := I_0^{n-\alpha}\frac{d^n}{dx^n}f(x)$$

where $n = \lfloor \alpha \rfloor + 1$. We will refer to $I^{\alpha}_{a+}f$ as the (Riemann Louiville) integral f of over α (based at a). Likewise we refer to $\mathcal{D}^{\alpha}_{a+}f$ as the (Riemann Louiville) derivative of order α (based at a). We also refer to ${}^{C}\mathcal{D}^{\alpha}_{a+}f$ as the Caputo derivative of order α (based at a).

For the rest of our considerations in this section we will take $a = 0$ (based at 0).

1.1.1 Notes on the Caputo derivative

Although the definitions for the Riemann-Liouville differential operators can be reasoned from the Cauchy formula for repeated integration and historical contexts [INSERT CC REF] [6, 7], the Caputo derivative is different in a number of subtle ways which make it useful from a practical standpoint.

While one can view the difference between the Riemann-Liouville fractional derivative and the Caputo fractional derivative as moving the derivative from one side of the integral sign to the other. In some sense one could say that the Caputo derivative *loses* some information about the function by differentiating first.

The Caputo derivative first turned up in a 1967 paper by Caputo linear models of dissipation [**?**] and again in 1971 in a paper by Caputo and Mainard. When specifying differential equations (both of the ordinary and partial type) Riemann-Liouville derivatives typically lead to intial conditions and boundary values of fractional order [**?**, **?**, 6, 7], however, physical intuition for what sensible values for these initial and boundary values should be is somewhat hard to come by [**?**, 6, 7]. The Caputo derivative, however, leads to systems where the initial and boundary values are of integer order, which leads to much greater physical usefulness.

Despite the fact that fractional differential equations with Riemann-Liouville operators in them often lead to initial and boundary conditions which are fractional, recent work has been done on interpreting these systems physically. c.f. [**?**]

1.2 Solution to a linear intial value problem via Laplace transforms

We aim to get a solution to the following fractional differential equation (in terms of Caputo derivatives)

$$\left({}^{C}\mathcal{D}_0^{\alpha} y\right)(t) = \beta y(t) \tag{1}$$

along with the initial conditions

$$y^{(k)}(0) = \begin{cases} 1 & k = 0 \\ 0 & 1 \leq k \leq \lfloor \alpha \rfloor - 1 \end{cases} \tag{2}$$

has the solution $y(t) = E_{\alpha}(\beta t^{\alpha})$. Where E_{α} is the one parameter Mittag-Lefler function.

This solution can be arrived at by a Laplace transform method. For completeness we define the following fractional integrals and derivatives.

We now consider the Laplace transform of the fractional integration and differentiation operators.

Lemma 1. *The Laplace transform of the Riemann-Liouville integral of a fuction f is as follows*

$$\mathcal{L}\{I_0^{\alpha} f\} = s^{-\alpha}\mathcal{L}\{f\}.$$

Proof. Since

$$(I_0^{\alpha} f)(t) = \frac{1}{\Gamma(\alpha)} \int_0^t f(u)(t-u)^{\alpha-1} du$$

is just $\frac{1}{\Gamma(\alpha)}$ times the convolution of f with $t^{\alpha-1}$ then by the convolution theorem for Laplace transforms we have that

$$\begin{aligned} \mathcal{L}\{I_0^{\alpha} f\} &= \frac{1}{\Gamma(\alpha)} \mathcal{L}\left\{ \int_0^t f(u)(t-u)^{\alpha-1} du \right\} \\ &= \frac{1}{\Gamma(\alpha)} \mathcal{L}\{f(t)\} \underbrace{\mathcal{L}\left\{t^{\alpha-1}\right\}}_{=s^{-\alpha}\Gamma(\alpha)} \\ &= s^{-\alpha}\mathcal{L}\{f\}. \end{aligned}$$

□

Lemma 2. *The Laplace transform the of the Riemann-Liouville derivative of a function f is as follows*

$$\mathcal{L}\{\mathcal{D}_0^{\alpha} f\} = s^{\alpha}\mathcal{L}\{f\} - \sum_{k=0}^{n-1} s^k \left(\mathcal{D}_0^{\alpha-k-1} f\right)(0).$$

Proof. See that

$$\begin{aligned}\mathcal{L}\left\{({}_0\mathcal{D}_t^{\alpha} f)\right\} &= \mathcal{L}\left\{\frac{d^n}{dt^n}\left(I_0^{n-\alpha} f\right)\right\} \\ &= s^n \mathcal{L}\left\{\left(I_0^{n-\alpha} f\right)\right\} - \sum_{k=0}^{n-1} s^k \frac{d^{n-k-1}}{dt^{n-k-1}}\left(I_0^{n-\alpha} f\right)(0)\end{aligned}$$

and by applying the result of 1 we get

$$\mathcal{L}\left\{\mathcal{D}_0^{\alpha} f\right\} = s^{\alpha}\mathcal{L}\left\{f\right\} - \sum_{k=0}^{n-1} s^k \left({}_0\mathcal{D}_{\alpha-k-1}^{f}(\right) 0).$$

□

Lemma 3. *The Laplace transform of the Caputo derivative of a function f is given as follows*

$$\mathcal{L}\left\{\left({}_0^C\mathcal{D}_t^{\alpha} f\right)\right\} = s^{\alpha-n}\left[s^n \mathcal{L}\left\{f\right\} - \sum_{k=0}^{n-1} s^{n-k-1}\left(\frac{d^k f}{dt^k}\right)(0)\right].$$

Proof. See that

$$\begin{aligned}\mathcal{L}\left\{\left({}_0^C\mathcal{D}_t^{\alpha} f\right)\right\} &= \mathcal{L}\left\{\left(I_0^{n-\alpha}\frac{d^n f}{dt^n}\right)\right\} \\ &= \frac{1}{\Gamma(n-\alpha)}\mathcal{L}\left\{\int_0^t (t-u)^{n-\alpha-1}\frac{d^n f}{dt^n}du\right\}\end{aligned}$$

which is the Laplace transform of a convolution so

$$\begin{aligned}\Gamma(n-\alpha)\mathcal{L}\left\{\int_0^t (t-u)^{n-\alpha-1}\frac{d^n f}{dt^n}du\right\} &= \mathcal{L}\left\{t^{n-\alpha-1}\right\}\mathcal{L}\left\{\frac{d^n f}{dt^n}\right\} \\ &= \frac{1}{n-\alpha}\left(s^{-(n-\alpha)}\Gamma(n-\alpha)\right) \\ &\quad \times\left(s^n\mathcal{L}\left\{f\right\} - \sum_{k=0}^{n-1} s^{n-k-1}\left(\frac{d^k f}{dt^k}\right)(0)\right) \\ &= s^{\alpha-n}\left[s^n\mathcal{L}\left\{f\right\} - \sum_{k=0}^{n-1} s^{n-k-1}\left(\frac{d^k f}{dt^k}\right)(0)\right].\end{aligned}$$

□

We now define the Mittag-Lefler function and calculate its Laplace transform.

Definition 2. *The one parameter Mittag-Lefler E_α function is defined by its power series.*

$$E_\alpha(t) = \sum_{k=0}^{\infty}\frac{t^k}{\Gamma(\alpha k+1)}$$

It is clear to see the definition of this function is inspired by the exponential function. Before we can calculate the Laplace transform of the Mittag-Lefler function we have to prove a simple lemma about the convergence of the series which is used in its definition.

Lemma 4. *The series*

$$\sum_{k=0}^{\infty} \frac{t^k}{\Gamma(\alpha k + 1)}$$

converges absolutely for all $t \in \mathbb{R}$.

Proof. Let $a_k = \frac{t^k}{\Gamma(\alpha k+1)}$ and see that

$$|\frac{a_{k+1}}{a_k}| = |t| \frac{\Gamma(\alpha k + 1)}{\Gamma(\alpha(k+1)+1)}$$

and that hence

$$\lim_{k \longrightarrow \infty} |\frac{a_{k+1}}{a_k}| = 0$$

for all $t \in \mathbb{R}$ so by the ratio test, the series $\sum_{k=0}^{\infty} \frac{t^k}{\Gamma(\alpha k+1)}$ converges for all $t \in \mathbb{R}$. □

Using this lemma we can then go on to state and prove the following lemma.

Lemma 5.

$$\mathcal{L}\{E_\alpha(\beta t^\alpha)\} = \frac{s^{\alpha-1}}{s^\alpha - \beta}$$

Proof. See that

$$\mathcal{L}\{E_\alpha(\beta t^\alpha)\} = \int_0^\infty e^{-st} \sum_{k=0}^{\infty} \frac{(\beta t^\alpha)^k}{\Gamma(\alpha k + 1)} dt$$

and because the series converges absolutely for all $t \in \mathbb{R}$ (lemma 4) we may interchange the integral and the sum to get

$$\begin{aligned} \int_0^\infty e^{-st} \sum_{k=0}^{\infty} \frac{(\beta t^\alpha)^k}{\Gamma(\alpha k + 1)} dt &= \sum_{k=0}^{\infty} \int_0^\infty e^{-st} \frac{(\beta t^\alpha)^k}{\Gamma(\alpha k + 1)} dt \\ &= \sum_{0}^{\infty} \frac{\beta^k}{\Gamma(\alpha k + 1)} \int_0^\infty e^{-st} t^{\alpha k} dt. \end{aligned}$$

By performing the change of variables $x = st$ we get that

$$\begin{aligned} \sum_{0}^{\infty} \frac{\beta^k}{\Gamma(\alpha k + 1)} \int_0^\infty e^{-st} t^{\alpha k} dt &= \sum_{0}^{\infty} \frac{\beta^k s^{-(k+1)}}{\Gamma(\alpha k + 1)} \underbrace{\int_0^\infty e^{-x} x^{\alpha k} dx}_{\Gamma(\alpha k+1)} \\ &= \sum_{k=0}^{\infty} \beta^k s^{-(\alpha k+1)} \\ &= \frac{s^{\alpha-1}}{s^\alpha - \beta}. \end{aligned}$$

So we have that

$$\mathcal{L}\left\{E_\alpha(\beta t^\alpha)\right\} = \frac{s^{\alpha-1}}{s^\alpha - \beta}$$

as required. □

We now have sufficient tools to attack the original problem, that is finding a solution to (1), (2).

Lemma 6. *The initial value problem defined in* (1) *and* (2)*, restated here for completeness*

$$\left({}^C\mathcal{D}_0^\alpha y\right)(t) = \beta y(t)$$

along with the initial conditions

$$y^{(k)}(0) = \begin{cases} 1 & k = 0 \\ 0 & 1 \leq k \leq \lfloor \alpha \rfloor - 1 \end{cases}$$

has solution $y(t) = E_\alpha\left(\beta t^\alpha\right)$.

Proof. Taking the Laplace transform of both sides of (1) yields

$$\mathcal{L}\left\{\left({}_0^C\mathcal{D}_t^\alpha y\right)\right\} = \beta\mathcal{L}\left\{y\right\}$$

$$s^{-(n+\alpha)}\left[s^n\mathcal{L}\left\{y\right\} - \sum_{k=0}^{n-1} s^{n-k-1}y^{(k)}(0)\right] = \beta\mathcal{L}\left\{y\right\}$$

by the result of lemma 3. Then taking into account (2) we get

$$s^{-(n+\alpha)}\left[s^n\mathcal{L}\left\{y\right\} - s^{n-1}\right] = \beta\mathcal{L}\left\{y\right\}$$

and so

$$\mathcal{L}\left\{y\right\} = \frac{s^{\alpha-1}}{s^\alpha - \beta}.$$

By using the result of lemma 5 we have that

$$y(t) = E_\alpha(\beta t^\alpha)$$

□

1.3 Solution to a multi-order initial value problem via Laplace transforms

This section follows the technique outlined in [6].

We wish to consider the following differential equation

$$\left({}_0\mathcal{D}_\Lambda^y(\,)\,t\right) + \left({}_0\mathcal{D}_\lambda^y(\,)\,t\right) = f(t) \tag{3}$$

where $0 < \lambda < \Lambda < 1$.

Firstly note that this differential equation is in terms of Riemann-Liouville derivatives. If we were to specify initial conditions we would be compelled to specify them in terms of fractional derivatives, so we leave them unspecified here to see the solution in general.

Again we will introduce a definition and prove a lemma which we will need to get a solution to 3

Definition 3 (Two Paramter Mittag-Lefler Function)**.** *We define the two paramter Mittag-Lefler function with the power series*

$$E_{\alpha,\gamma}(t) := \sum_{k=0}^{\infty} \frac{t^k}{\Gamma(\alpha k + \gamma)}.$$

Note that this is just a generalisation of the one paramter Mittag-Lefler function, in that $E_\alpha(t) = E_{\alpha,1}(t)$.

The follopwing lemma is essentially a generalisation of lemma 5.

Lemma 7. *The Laplace transform of* $t^{\alpha m+\gamma-1}E^{(m)}_{\alpha,\gamma}(t)$ *is given by*

$$\mathcal{L}\left\{t^{\alpha m+\gamma-1}E^{(m)}_{\alpha,\gamma}(\beta t^\alpha)\right\} = \frac{m! s^{\alpha-\gamma}}{(s^\alpha-\beta)^{m+1}}$$

Proof. Firstly see that

$$\begin{aligned} E^{(m)}_{\alpha,\gamma}(t) &= \sum_{k=m}^{\infty} \frac{\frac{k!}{(k-m)!}t^{k-m}}{\gamma(\alpha k+\gamma)} \\ &= \sum_{k=0}^{\infty} \frac{(k+m)!t^k}{k!\Gamma(\alpha k+\gamma)} \end{aligned}$$

so we have that

$$E^{(m)}_{\alpha,\gamma}(\beta t^\alpha) = \sum_{k=0}^{\infty} \frac{(k+m)!t^{\alpha k}\beta^k}{k!\Gamma(\alpha(k+m)+\gamma)}.$$

We can then write that

$$\begin{aligned} \mathcal{L}\left\{t^{\alpha m+\gamma-1}E^{(m)}_{\alpha,\gamma}(t)\right\} &= \int_0^\infty t^{\alpha m+\gamma-1}\sum_{k=0}^{\infty} \frac{(k+m)!t^{\alpha k}\beta^k}{k!\Gamma(\alpha(k+m)+\gamma)} \\ &= \sum_{k=0}^{\infty} \frac{\beta^k(k+m)!}{\Gamma(\alpha(k+m)+\gamma)k!} \underbrace{\int_0^\infty e^{-st}t^{\alpha(k+m)+\gamma-1}dt}_{\circledast}. \end{aligned}$$

Considering just $\circledast$ and performing the substitution $x = st$ we get that

$$\begin{aligned} \circledast &= s^{-\alpha(k+m)-\gamma}\int_0^\infty e^{-x}x^{\alpha(k+m)+\gamma-1}dx \\ &= s^{-\alpha(k+m)-\gamma}\Gamma(\alpha(k+m)+\gamma) \end{aligned}$$

and so

$$\mathcal{L}\left\{t^{\alpha m+\gamma-1}E^{(m)}_{\alpha,\gamma}(t)\right\} = s^{-\alpha m-\gamma}\sum_{k=0}^{\infty}\left(\frac{\beta}{s^\alpha}\right)^k\frac{(k+m)!}{k!}.$$

Now by the derivative rule for geometric series we get

$$\begin{aligned} \sum_{k=0}^{\infty}\left(\frac{\beta}{s^\alpha}\right)^k\frac{(k+m)!}{k!} &= \frac{m!}{(1-\frac{\beta}{s^\alpha})^{m+1}} \\ &= \frac{s^{\alpha(m+1)}m!}{(s^\alpha-\beta)^{m+1}} \end{aligned}$$

and so

$$\mathcal{L}\left\{t^{\alpha m+\gamma-1}E_{\alpha,\gamma}^{(m)}(t)\right\} = \frac{m!s^{\alpha-\gamma}}{(s^\alpha-\beta)^{m+1}}.$$

□

Lemma 8. *The initial value problem, 3, restated here for completeness,*

$$\left({}_0\mathcal{D}_\Lambda^y()\,t\right) + \left({}_0\mathcal{D}_\lambda^y()\,t\right) = f(t)$$

has solution, given by

$$y(t) = Cg(t) + \int_0^t g(t-\tau)f(\tau)d\tau$$

where

$$C = \left({}_0\mathcal{D}_{\Lambda-1}^y()\right)0) + \left({}_0\mathcal{D}_{\lambda-1}^y()\right)0)$$
$$g(t) = t^{\Lambda-1}E_{\Lambda-\lambda,\Lambda}(-t^{\Lambda-\lambda}).$$

Proof. Taking the Laplace transform of both sides of 3 and using the result of lemma 2 we get that

$$\mathcal{L}\left\{\left({}_0\mathcal{D}_\Lambda^y()\,t\right)\right\} + \mathcal{L}\left\{\left({}_0\mathcal{D}_\lambda^y()\,t\right)\right\} = \mathcal{L}\left\{f(t)\right\}$$
$$s^\Lambda Y(s) + s^\lambda Y(s) - \left({}_0\mathcal{D}_{\Lambda-1}^y()\right)0) - \left({}_0\mathcal{D}_{\lambda-1}^y()\right)0) = F(s).$$

Note that

$$C = \left({}_0\mathcal{D}_{\Lambda-1}^y()\right)0) + \left({}_0\mathcal{D}_{\lambda-1}^y()\right)0)$$

is a constant so we write

$$Y(s) = \frac{C+F(s)}{s^\Lambda + s^\lambda}$$
$$= (C+F(s))\,\frac{s^{-\lambda}}{s^{\Lambda-\lambda}+1}.$$

Let

$$G(s) = \frac{s^{-\lambda}}{s^{\Lambda-\lambda}+1}$$

and by using lemma 7 with $\alpha = \Lambda - \lambda$ and $\gamma = \Lambda$ we get that

$$g(s) = t^{\Lambda-1}E_{\Lambda-\lambda,\Lambda}(-t^{\Lambda-\lambda})$$

where

$$\mathcal{L}\left\{g(t)\right\} = G(s)$$

.

Then using the Laplace convolution theorem we get that

$$y(t) = Cg(t) + \int_0^t g(t-\tau)f(\tau)d\tau$$

where

$$C = \left({}_0\mathcal{D}_{\Lambda-1}^y()\right)0) + \left({}_0\mathcal{D}_{\lambda-1}^y()\right)0)$$
$$g(t) = t^{\Lambda-1}E_{\Lambda-\lambda,\Lambda}(-t^{\Lambda-\lambda}).$$

□

1.4 Existence and uniquness of solutions to initial value problems

After looking at the solution to a couple of fractional differential equations we wish to consider the existence an uniqueness of solutions to a class fractional differential equations. This generalizes a result and technique of Tisdell [8] but a similar result for Miller-Ross sequential fractional differential equations can be found in [6].

Theorem 1 (Uniqueness)**.** *Consider the following intial value problem,*

$$\sum_{j=1}^{N} \beta_j \left({}_0^C\mathcal{D}_t^{\alpha_j} x \right)(t) = f(t, x(t)) \tag{4}$$

$$x(0) = A_0, x_1(0) = A_1, \ldots, x^{n_N}(0) = A_{n_N} \tag{5}$$

where $\alpha_1 > \alpha_2 > \ldots > \alpha_N$, $n_j = \lceil \alpha_j \rceil - 1$ *and* $\beta_1 = 1$.

Define

$$S := \{(t, p) \in \mathbb{R}^2 : t \in [0, a], p \in \mathbb{R}\}$$

Let $f : S \longrightarrow \mathbb{R}$ *be continuous. If there is a positive constant* L *such that*

$$|f(t, u) - f(t, v)| \leq L|u - v|, \text{ for all } (t, u), (t, v) \in S$$

and the constants $\{\alpha_j\}_{j=1}^N$, $\{\beta_j\}_{j=1}^N$ *are such that*

$$\sum_{j=2}^{N} |\beta_j| \, a^{\alpha_1 - \alpha_j} < 1$$

then the intial value problem defined in 4 and 5 has a unique solution.

To prove this we will need several lemmas.

Lemma 9. *The IVP defined in* (4), (5) *is equivalent to the integral equation*

$$x(t) = \sum_{k=1}^{n_1} \frac{A_k t^k}{k!} + \frac{1}{\beta_1}\Big(\frac{1}{\Gamma(\alpha_1)} \int_0^t (t-s)^{\alpha_1 - 1} f(s, x(s)) ds - \sum_{j=2}^{N} \beta_j \frac{1}{\Gamma(\alpha_1 - \alpha_j)} \int_0^t (t-s)^{\alpha_1 - \alpha_j - 1} \left(x(s) - \sum_{k=1}^{n_j} \frac{A_k s^k}{k!} \right) ds \Big)$$

Proof. Apply (I_0^α) to both sides of (4) and recognize that

$$\left(I_0^\alpha \left({}_0^C\mathcal{D}_t^\alpha x \right) \right)(t) = x(t) + \sum_{k=0}^{n} \frac{x^{(k)}(0) t^k}{k!}$$

where $n = \lceil \alpha \rceil - 1$. □

Lemma 10.

$$\left(I_0^\xi E_\alpha(\gamma t^\alpha) \right) \leq t^\xi E_\alpha(\gamma t^\alpha)$$

Proof. See that

$$\begin{aligned}\left(I_0^{\xi} E_\alpha(\gamma t^\alpha)\right) &= \frac{1}{\Gamma(\xi)} \int_0^t E_\alpha(\gamma s^\alpha)(t-s)^{\xi-1} ds \\ &= \frac{1}{\Gamma(\xi)} \int_0^t \sum_{k=0}^{\infty} \frac{\gamma^k s^{\alpha k}}{\Gamma(\alpha k+1)} (t-s)^{\xi-1} ds \\ &= \frac{1}{\Gamma(\xi)} \sum_{k=0}^{\infty} \frac{\gamma^k}{\Gamma(\alpha k+1)} \underbrace{\int_0^t s^{\alpha k}(t-s)^{\xi-1} ds}_{\circledast}.\end{aligned}$$

Letting $\tau = \frac{s}{t}$ we have that

$$\begin{aligned}\circledast &= \int_0^1 (t\tau)^{\alpha k}(t-t\tau)^{\xi-1} t d\tau \\ &= t^{\alpha k+\xi} \int_0^1 (\tau)^{\alpha k}(1-1\tau)^{\xi-1} d\tau \\ &= t^{\alpha k+\xi} B(\alpha k+1, \xi) \\ &= t^{\alpha k+\xi} \frac{\Gamma(\alpha k+1)\Gamma(\xi)}{\Gamma(\alpha k+\xi+1)}.\end{aligned}$$

This means that

$$\begin{aligned}\left(I_0^{\xi} E_\alpha(\gamma t^\alpha)\right) &= \sum_{k=0}^{\infty} \frac{\gamma^k t^{\alpha k+\xi}}{\Gamma(\alpha k+\xi+1)} \\ &= t^{\xi} \sum_{k=0}^{\infty} \frac{\gamma^k t^{\alpha k}}{\Gamma(\alpha k+\xi+1)} \\ &\leq t^{\xi} \sum_{k=0}^{\infty} \frac{\gamma^k t^{\alpha k}}{\Gamma(\alpha k+1)} \\ &= t^{\xi} E_\alpha(\gamma t^\alpha).\end{aligned}$$

□

Lemma 11.

$$(I_0^{\alpha} E_\alpha(\gamma t^\alpha)) = \frac{1}{\gamma}\left(E_\alpha(\gamma t^\alpha) - 1\right)$$

Proof. See that

$$\begin{aligned}(I_0^{\alpha} E_\alpha(\gamma t^\alpha)) &= \frac{1}{\Gamma(\alpha)} \int_0^t E_\alpha(\gamma s^\alpha)(t-s)^{\alpha-1} ds \\ &= \frac{1}{\Gamma(\alpha)} \sum_{k=0}^{\infty} \frac{\gamma^k}{\Gamma(\alpha k+1)} \underbrace{\int_0^t s^{\alpha k}(t-s)^{\alpha} ds}_{\circledast}.\end{aligned}$$

Letting $\tau = \frac{s}{t}$ we have that

$$\begin{aligned}
\circledast &= \int_0^1 (t\tau)^{\alpha k}(t - t\tau)^{\alpha-1} t d\tau \\
&= t^{\alpha(k+1)} \int_0^1 \tau^{\alpha k}(1-\tau)^{\alpha-1} d\tau \\
&= t^{\alpha(k+1)} B(\alpha k + 1, \alpha) \\
&= t^{\alpha(k+1)} \frac{\Gamma(\alpha k + 1)\Gamma(\alpha)}{\Gamma(\alpha(k+1)+1)}.
\end{aligned}$$

This then means that

$$\begin{aligned}
(I_0^\alpha E_\alpha(\gamma t^\alpha)) &= \sum_{k=0}^{\infty} \frac{\gamma^k t^{\alpha(k+1)}}{\Gamma(\alpha(k+1)+1)} \\
&= \frac{1}{\gamma} \sum_{k=1}^{\infty} \frac{\gamma^k t^{\alpha k}}{\Gamma(\alpha k + 1)} \\
&= \frac{1}{\gamma} \left(\sum_{k=0}^{\infty} \frac{\gamma^k t^{\alpha k}}{\Gamma(\alpha k + 1)} - 1 \right) \\
&= \frac{1}{\gamma} \left(E_\alpha(\gamma t^\alpha) - 1 \right).
\end{aligned}$$

□

Proof of theorem 1. To arrive at this we only have to prove that the map

$$\begin{aligned}
[Fx](t) := \sum_{k=1}^{n_1} \frac{A_k t^k}{k!} + \frac{1}{\beta_1} \Big(& \frac{1}{\Gamma(\alpha_1)} \int_0^t (t-s)^{\alpha_1 - 1} f(s, x(s)) ds \\
& - \sum_{j=2}^{N} \frac{\beta_j}{\Gamma(\alpha_1 - \alpha_j)} \int_0^t (t-s)^{\alpha_1 - \alpha_j - 1} \left(x(s) - \sum_{k=1}^{n_j} \frac{A_k s^k}{k!} \right) ds \Big)
\end{aligned}$$

is contractive in the metric space $\left(C[0,a], d_\gamma^{\alpha_1}\right)$ where

$$d_\gamma^{\alpha_1}(x,y) = \max_{t \in [0,a]} \frac{|x(t) - y(t)|}{E_{\alpha_1}(\gamma t^{\alpha_1})}.$$

To see this note that

$$\begin{aligned}
d_\gamma^{\alpha_1}(Fx, Fy) = \max_{t\in[0,a]} & \frac{1}{E_{\alpha_1}(\gamma t^{\alpha_1})} \left| \frac{1}{\beta_1} \right| \Big| \frac{1}{\Gamma(\alpha_1)} \int_0^t (t-s)^{\alpha_1 - 1} (f(s, x(s)) - f(s, y(s)) ds \\
& - \sum_{j=2}^{N} \frac{\beta_j}{\Gamma(\alpha_1 - \alpha_j)} \int_0^t (t-s)^{\alpha_1 - \alpha_j - 1} (x(s) - y(s)) ds \Big| \\
\leq \max_{t\in[0,a]} & \frac{1}{E_{\alpha_1}(\gamma t^{\alpha_1}) |\beta_1|} \Big(\frac{1}{\Gamma(\alpha_1)} \int_0^t (t-s)^{\alpha_1 - 1} |f(s, x(s)) - f(s, y(s))| ds \\
& + \sum_{j=2}^{N} \frac{|\beta_j|}{\Gamma(\alpha_1 - \alpha_j)} \int_0^t (t-s)^{\alpha_1 - \alpha_j - 1} |x(s) - y(s))| ds \Big).
\end{aligned}$$

By exploiting the Lipshitz condition we can further write that

$$\begin{aligned}
d_\gamma^{\alpha_1}(Fx,Fy) &\leq \max_{t\in[0,a]} \frac{1}{E_{\alpha_1}(\gamma t^{\alpha_1})|\beta_1|}\Big(\frac{L}{\Gamma(\alpha_1)}\int_0^t (t-s)^{\alpha_1-1}|x(s)-y(s)|ds \\
&\quad + \sum_{j=2}^{N}\frac{|\beta_j|}{\Gamma(\alpha_1-\alpha_j)}\int_0^t (t-s)^{\alpha_1-\alpha_j-1}|x(s)-y(s))|ds\Big) \\
&= \max_{t\in[0,a]} \frac{1}{E_{\alpha_1}(\gamma t^{\alpha_1})|\beta_1|}\Big(\frac{L}{\Gamma(\alpha_1)}\int_0^t (t-s)^{\alpha_1-1}\frac{|x(s)-y(s)|}{E_{\alpha_1}(\gamma s^{\alpha_1})}E_{\alpha_1}(\gamma s^{\alpha_1})ds \\
&\quad + \sum_{j=2}^{N}\frac{|\beta_j|}{\Gamma(\alpha_1-\alpha_j)}\int_0^t (t-s)^{\alpha_1-\alpha_j-1}\frac{|x(s)-y(s))|}{E_{\alpha_1}(\gamma s^{\alpha_1})}E_{\alpha_1}(\gamma s^{\alpha_1})ds\Big) \\
&\leq d_\gamma^{\alpha_1}(x,y)\max_{t\in[0,a]} \frac{1}{E_{\alpha_1}(\gamma t^{\alpha_1})|\beta_1|}\Big(\frac{L}{\Gamma(\alpha_1)}\int_0^t (t-s)^{\alpha_1-1}E_{\alpha_1}(\gamma s^{\alpha_1})ds \\
&\quad + \sum_{j=2}^{N}\frac{|\beta_j|}{\Gamma(\alpha_1-\alpha_j)}\int_0^t (t-s)^{\alpha_1-\alpha_j-1}E_{\alpha_1}(\gamma s^{\alpha_1})ds\Big) \\
&= d_\gamma^{\alpha_1}(x,y)\max_{t\in[0,a]} \frac{1}{E_{\alpha_1}(\gamma t^{\alpha_1})|\beta_1|}\Big(L\left(I_0^{\alpha_1}E_{\alpha_1}(\gamma t^{\alpha_1})\right. \\
&\quad \left. + \sum_{j=2}^{N}|\beta_j|\left(I_0^{\alpha_1-\alpha_j}E_{\alpha_1}(\gamma t^{\alpha_1})\right)\right).
\end{aligned}$$

We can now use the results of lemmas 10 and 11 to write

$$\begin{aligned}
d_\gamma^{\alpha_1}(Fx,Fy) &\leq d_\gamma^{\alpha_1}(x,y)\max_{t\in[0,a]} \frac{1}{E_{\alpha_1}(\gamma t^{\alpha_1})|\beta_1|}\Big(\frac{L}{\gamma}\left(E_{\alpha_1}(\gamma t^{\alpha_1})-1\right) \\
&\quad + \sum_{j=2}^{N}|\beta_j|t^{\alpha_1-\alpha_j}E_{\alpha_1}(\gamma t^{\alpha_1})\Big) \\
&= d_\gamma^{\alpha_1}(x,y)\max_{t\in[0,a]} \frac{1}{|\beta_1|}\Big(\frac{L}{\gamma}\left(1-\frac{1}{E_{\alpha_1}(\gamma t^{\alpha_1})}\right)+\sum_{j=2}^{N}|\beta_j|t^{\alpha_1-\alpha_j}\Big)
\end{aligned}$$

and finally we get that

$$d_\gamma^{\alpha_1}(Fx,Fy) \leq d_\gamma^{\alpha_1}(x,y)\frac{1}{|\beta_1|}\left(\frac{L}{\gamma}+\sum_{j=2}^{N}|\beta_j|a^{\alpha_1-\alpha_j}\right).$$

By choosing γ sufficiently large we get that

$$\frac{1}{|\beta_1|}\left(\frac{L}{\gamma}+\sum_{j=2}^{N}|\beta_j|a^{\alpha_1-\alpha_j}\right)<1$$

and so F is a contractive mapping and thus the IVP defined in (4), (5) has a unique solution on $[0,a]$. □

Note that although existence is resolved (by virtue of the solutions given above) for the differential equations in (1, 2) and 3, this guarentees uniqueness on some closed interval starting at 0 for both cases. Its also important to note that this result can be extended to differential equations involving Riemann-Liouville derivatives, by virtue of the correspondence between the Caputo derivative and the Riemann-Liouville derivative [6].

2 Numerical Methods for Fractional Differential Equations

In this section we describe numerical methods which can be used for solving fractional differential equations. We begin by looking at two ways for approximating a fractional derivative and then look at a specific scheme outlined by Diethelm [3].

2.1 Approximations of Fractional Derivatives and Integrals

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

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

For a very large class of functions this coincides with the Riemann-Liouville fractional derivative [6] which we again recall here

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

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

2.1.1 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

$$\widehat{\left({}_a^{GL}\mathcal{D}_t^\alpha f\right)}(t) = \frac{1}{h^\alpha}\sum_{j=0}^{n}(-1)^j\binom{\alpha}{j}f(t-jh). \tag{6}$$

It can be shown [6] that this yields a first order approximation of the Riemann-Liouville fractional derivative.

2.1.2 Quadrature Approximation

A completely different approach is to use a quadrature rule to evaluate 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 off a quadrature method. [3]

2.1.3 An Initial Value Problem

Lets consider the initial value problem

$$\left({}_0^C\mathcal{D}_x^\alpha y\right) = f(x,y) \tag{7}$$

$$y^{(k)}(0) = y_0^{(k)} \tag{8}$$

for $0 \leq k < \lfloor \alpha \rfloor$. Note that this initial value problem is stated in terms of Caputo derivatives. The motivation for such initial 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 $y(x), x \in [0, x_0]$ [2]. This is in contrast to ordinary differential equations and this fact is what makes fractional differential equations considerably more complex to solve. Even multi-step 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]

2.1.4 The Adams Moulton Bashforth Scheme

We briefly outline a method explained and analyzed in detail in [4]. As shown in section 1.4 the initial value problem (7), (8) is equivalent to the Volterra equation.

$$y(x) = \sum_{k=0}^{\lceil \alpha \rceil - 1} y_0^{(k)} \frac{x^k}{k!} + \frac{1}{\Gamma(\alpha)} \int_0^x (x-t)^{\alpha-1} f(t, y(t)) dt \tag{9}$$

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 grid-points x_k. As outlined in [4] we can approximate the integral in (9) as

$$\int_0^{x_{k+1}} (x_{k+1} - t)^{\alpha-1} \tilde{g}_{k+1}(t) dt = \sum_{j=0}^{k+1} a_{j,k+1} g(x_j) \tag{10}$$

where

$$a_{j,k+1} = \frac{h^\alpha}{\alpha(\alpha+1)} \times \begin{cases} (k^{\alpha+1} - (k-\alpha)(k+1)^\alpha) & \text{if } j = 0 \\ ((k-j+2)^{\alpha+1} + (k-j)^{\alpha+1} - 2(k-j+1)^{\alpha+1}) & \text{if } 1 \leq j \leq k \\ 1 & \text{if } j = k+1. \end{cases} \tag{11}$$

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

By separating the final term in the sum we can write

$$y_{k+1} = \sum_{j=0}^{\lceil\alpha\rceil-1} y_0^{(j)} \frac{x_{k+1}^j}{j!} + \frac{1}{\Gamma(\alpha)} \left(\sum_{j=0}^{k} a_{j,k+1} f(x_j, y_j) + a_{k+1,k+1} f(x_{k+1}, y_{k+1}^P) \right) \tag{12}$$

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 [4].

This predicted value is calculated by taking a rectangle approximation to the integral in (9), to get

$$\int_0^{x_{k+1}} (x_{k+1} - t)^{\alpha-1} g(t) dt \approx \sum_{j=0}^{k} b_{j,k+1} g(x_j) \tag{13}$$

where

$$b_{j,k+1} = \frac{h^\alpha}{\alpha} \left((k+1-j)^\alpha - (k-j)^\alpha \right). \tag{14}$$

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

$$y_{k+1}^P = \sum_{j=0}^{\lceil\alpha\rceil-1} \frac{x_{k+1}^j}{j!} y_0^{(j)} + \frac{1}{\Gamma(\alpha)} \sum_{j=0}^{k} b_{j,k+1} f(x_j, y_j). \tag{15}$$

This outlines a fractional Adams Moulton Bashforth scheme. Of particular note are the sums which arise in (10) and (13). 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 [4]. 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 $O(N^2)$ [3]. Section 2.1.5 will outline a task based parallel approach and section 2.1.7 will outline a massively parallel approach to solving (7), (8) with NVIDIA's CUDA.

2.1.5 Parallelizing: A Task Based Approach

Diethelm's paper [3] outlines a parallel method of numerically approximating the solution to (7), (8) by using a thread based parallel implementation of the Adams Moulton Bashforth method outlined in 2.1.4. We base our approach in this section broadly on those of [3] 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 grid-points $x_0, \ldots, x_{N-1}$ 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 immediately 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 1. We have broken the vector (array) $\mathbf{y}$ up into $K = \lceil \frac{N}{p} \rceil$ blocks of p variables.

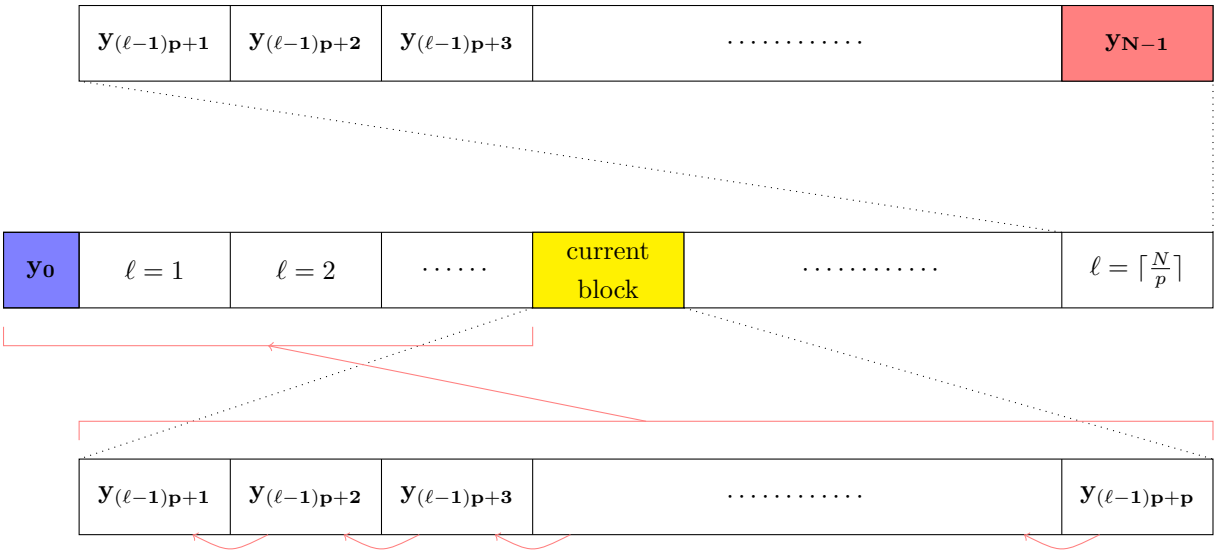


Figure 1: Computation diagram for the values in the vector (array) $\mathbf{y}$.

The center 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 emphasize 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 [3] we rewrite the equations (12) and (15) as

$$y_{j+1}^P = I_{j+1} + h^\alpha H_{j,\ell}^P + h^\alpha L_{j,\ell}^P \tag{16}$$

and

$$y_{j+1} = I_{j+1} + h^\alpha H_{j,\ell} + h^\alpha L_{j,\ell} \tag{17}$$

where

$$I_{j+1} := \sum_{k=0}^{\lceil\alpha\rceil-1} \frac{x_{j+1}^k}{k!} y_0^{(k)} \tag{18}$$

$$H_{j,\ell}^P := \sum_{k=0}^{(\ell-1)p} b_{j-k} f(x_k, y_k) \tag{19}$$

$$L_{j,\ell}^P := \sum_{k=(\ell-1)p+1}^{j} b_{j-k} f(x_k, y_k) \tag{20}$$

$$H_{j,\ell} := c_j f(x_0, y_0) + \sum_{k=1}^{(\ell-1)p} a_{j-k} f(x_k, y_k) \tag{21}$$

$$L_{j,\ell} := \sum_{k=(\ell-1)p+1}^{j} aj - kf(x_k, y_k) + \frac{f(x_{j+1}, y_{j+1}^P)}{\Gamma(\alpha + 2.)} \tag{22}$$

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 proceeds 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 reasonably 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 respectively. 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.

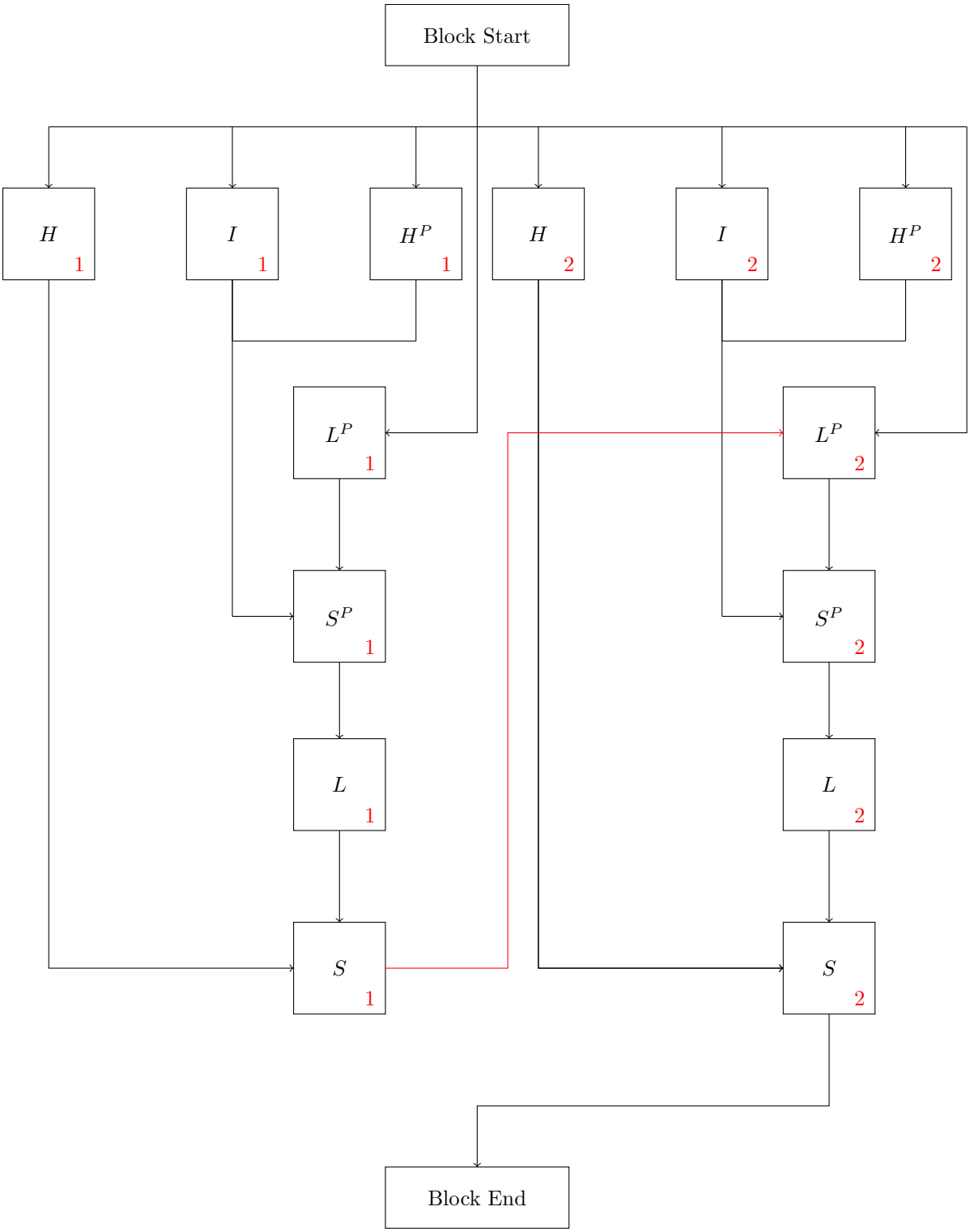


Figure 2: Task flow diagram for each block in the case when $p = 2$

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

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

For the theoretical analysis of the scheme we introduce Amdahl's Law or Amdahl's principle.

Principle 1 (Amdahl)**.** *Let P be the proportion of a computation which can be parallelized in a given progress. Then the speedup given by Q cores is*

$$S(Q) = \frac{1}{(1-P) + \frac{P}{Q}}$$

This law was first introduced in a 1967 paper by computer architect Amdahl. We introduce a new notion of Amdahl efficiency to characterize the how *good* a parallel algorithm is in comparison with it serial counterpart in the limit of large input.

To formalize this notion we introduce the following definition.

Definition 4 (Amdahl Efficiency)**.** *Let $S(N, Q)$ represent the parallel speedup for a problem with input size N and executed with Q cores. Then we define the Amdahl efficiency of parallel scheme as*

$$E(Q) = \lim_{N \longrightarrow \infty} \frac{S(N, Q)}{Q}.$$

Further we say that a parallel scheme is Amdahl efficient with respect to its serial counterpart if $E(p) = 1$.

Proposition 1. *The parallel Adams Moulton Bashforth scheme is Amdahl efficient with respect to its serial counterpart.*

Proof. We present the paralellizable fraction of computation in each block of variables in the following table. The order column represents the number of *multiply-add* operations per task.

Task	Computational Complexity	Number of Variables per Block	Total Complexity
H	$O(\ell p)$	p	$O(\ell p^2)$
H_p	$O(\ell p)$	p	$O(\ell p^2)$
I	$O(\alpha)$	p	$O(p)$
L	$O(p)$	p	$O(p^2)$
L_p	$O(p)$	p	$O(p^2)$

In each block H, H_p and I can be computed in parallel across all variables in the block. For the purposes of simplicity we assume that the tasks L and L_p must all be computed in serial across all variables in a block. This means that in the ℓ-th block the non-paralellizable fraction of computation is

$$\overline{P}_{block}(\ell) \sim \frac{2p^2}{2p^2(\ell+1)} \sim \frac{1}{\ell} \tag{23}$$

The amount of computation in the ℓ-th block is $O(\ell p^2)$ and there are $K \approx \frac{N}{p}$ blocks in the total computation. This means that the total amount of computation is

$$\sum_{\ell=1}^{K} \ell p^2 \sim \frac{K^2 p^2}{2} \tag{24}$$

and by taking a weighted average sum, the total fraction of non-paralellizable computation is

$$\overline{P}_{total} \sim \sum_{\ell=1}^{K} \frac{2\ell p^2}{K^2 \ell p^2} = \frac{2}{K} = \frac{2p}{N}. \tag{25}$$

Noticing the fact that the computation in the task I is insignificant compared to H and H_p we would suppose using $Q = 2p$ cores would be in some sense optimal. Applying Amdahl's principle with the result in 25 and with $Q = 2p$ we get

$$\begin{aligned} S(N, Q) &= S(N, 2p) \\ &\sim \frac{1}{(\frac{2p}{N}) + \frac{1-\frac{2p}{N}}{2p}} \end{aligned}$$

then letting $N \longrightarrow \infty$ we get

$$E(p) = \lim_{N \longrightarrow \infty} \left(\frac{1}{(\frac{2p}{N}) + \frac{1-\frac{2p}{N}}{2p}} \right) / p = 1.$$

□

What this result means is that if we have $2p$ free cores this parallel method should take $\frac{1}{2p}$ the amount of time to run compared to it's serial counterpart given a sufficiently large number of grid points, N. Due to the fact that this a task based scheme this is also dependent on an efficient scheduler.

We now compare this with practical experimentation. This method was developed in C# with MONO (see Appendix A for C# code) and run on a machine with an Intel ®Core ™ i7-4930K CPU @ 3.40GHz. This particular CPU has 6 physical cores with 12 logical cores provided by hyper-threading.

We solved the following initial value problem

$$\left({}_0^C \mathcal{D}_x^{\frac{1}{2}} y \right)(x) = -y(x) \qquad\qquad y(0) = 1 \tag{26}$$

with $N = 2000$ through to $N = 10000$ time steps over the interval $[0, 10]$ with p ranging from 1 though to 12. Each solution was computed 5 times and the average runtime was computed, yielding the *performance surface* shown in figure 3. [2]

[2]See Appendix B for a full table of runtimes.

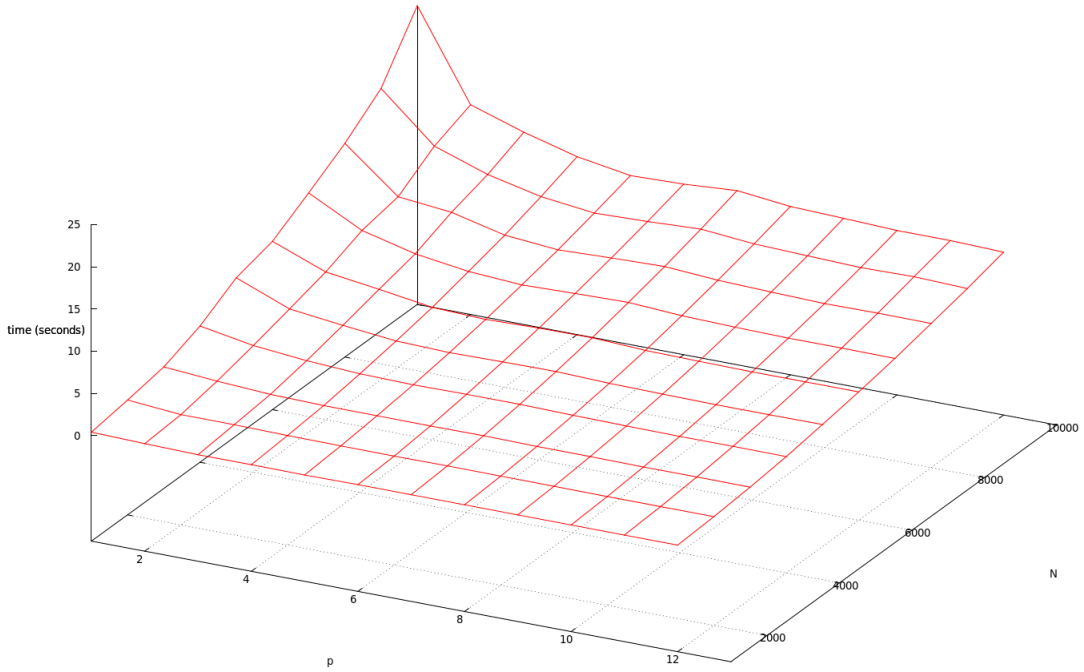


Figure 3: Average runtime of the scheme for different values of p and N.

Figure 3 clearly shows the $O(N^2)$ complexity of the algorithm, especially in the case where $p = 1$. We present some of the results from the $N = 10000$ computations in a figure 2.1.6 to illustrate the speedup.

p	Runtime (s)	Speedup (wrt p = 1)
1	22.89	1
2	12.35	1.85
3	10.28	2.22
4	8.58	2.66
5	7.53	3.04
6	7.70	2.97

From a theoretical perspective we would expect speedups of 2, 3 and 4 for the cases $p = 2, 3, 4$ respectively, however, this is not achieved. There are a number of possible reasons for this. Firstly the MONO task scheduler (based of the .Net scheduler) is not performance optimized [1].

There is the possibility that we haven't chosen a large enough value of N to get the full relative speedup. This is supported by the fact that for the $N = 8000$ case the speedup from $p = 1$ to $p = 2$ was 1.53 (less than that in the $N = 10000$ case).

Also for cases where $p > 3$ we have 6 computationally heavy tasks (H and H_p) being completed per block and are therefore reaching the limit of the number of physical cores on the system. Obviously we cannot expect an improvement once p gets too large as the number of tasks to be executed in parallel exceeds the number of available cores. We see the full effect of this in the $p = 6$ case where the runtime was actually longer than that for $p = 5$.

2.1.7 Parallelizing: A CUDA Approach

One of the main benefits of the scheme outlined in section 2.1.5, especially noted in it's form presented by Diethelm in [3] is the ability for this scheme to be implemented across multiple machines with a small amount of message passing between executing threads (or in this case tasks). The main reason for wanting to minimize message passing is that it can significantly impact performance to the point where most of the time is actually spent communicating updates between execution nodes. This is given significant consideration in [3]. In this section we abandon these measures in favor of massive performance gains when executing in a massively parallel manner on a GPU.

Modern graphics cards have the advantage of being able to efficiently perform massively parallel computations, especially calculations involving floating point arithmetic [5]. For example the NVIDIA GeForce GTX 780 has 2304 cores clocked at 863MHz. We use one of these graphics cards in an effort to dramatically speed up the fractional Adams Moulton Bashforth Method using NVIDIA's CUDA tool-chain.

Instead of using blocks of variables as was done in [3] and reimplemented by us in section 2.1.5 we return to the original specification of the scheme in section 2.1.4 and seek out other opportunities of parallelism.

The essential problem with the scheme is that the number of terms in the sums defined in (15) and (12) grows linearly with each time step taken. It is clear that the actual terms in each sum are independent and so breaking the sum up into smaller parts to be calculated and then summed together is a reasonable approach. This is so long as communication between cores doing different parts of the sum is sufficiently fast and global memory access is sufficiently fast. In the case of a GPU both of these criteria are satisfied and so this approach is reasonable.[3]

We aim to use as much of the GPU as possible to perform the calculations and so unlike the scheme outlined in section 2.1.5 we will automatically scale the number of threads to use as many cores as possible. We define one *performance tuning* parameter $opsPerThread$ which define the number of *multiply-add* operations per thread. Figure 4 shows a *performance surface* showing runtime against number of time steps N and $opsPerThread$ [4]. The actual initial value problem being solved is exactly the same as in section 2.1.5.

[3] It is easy to see that this scheme would be Amdahl efficient in the sense of definition 4.

[4] See appendix B for a full table of runtimes

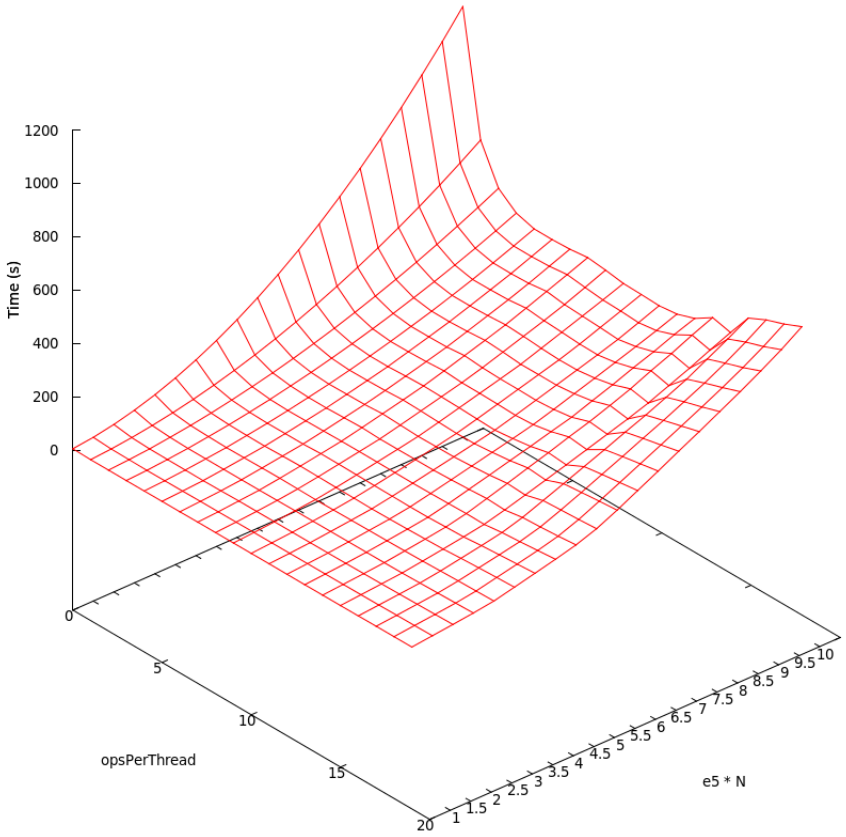


Figure 4: Runtime for the scheme for various values of opsPerThread and N.

Firstly we note the size of the problems being solved here. The last point on the N axis is 10^6 time-steps, 100 times larger than was computed in the CPU based C# approach. Given the quadratic computational complexity of this algorithm that means this CUDA implementation is solving problems which would take roughly 2.5 days to compute using the previous parallel scheme, in less than 6 minutes.

As can be seen from this chart, as N increases it makes sense to perform more operations per thread. This arises from the fact that there is an overhead in dealing with each thread and this overhead becomes significant as N gets large. This effect is also exacerbated by the fact that for small values of N the overhead of managing more threads on the GPU is offset by the increased parallelism that a smaller number of operations per thread affords. Once, however, every core is being utilized it makes sense to start to increase the number of operations per thread as N is increased.

The speedup from running these computation on CUDA is enormous. It is arguable that any advantage achieved through reducing the number of messages passed between multiple threads and nodes by using the scheme outlined in [3] and in section 2.1.5, is completely outweighed by the sheer speed of computation on a GPU.

3 Numerical Evaluation of Special Functions

In this section we cover the numerical evaluation of the Mittag-Leffler function and the Wright function which as seen in other sections of this thesis, are functions which commonly arise when dealing with fractional differential equations.

Even if an analytic solution to an initial value problem, or a boundary value problem is known, actually evaluating values of the functions involved in the function is not trivial. Despite the fact that both the Mittag-Leffler function and the Wright function are defined in terms of series, these series do not converge at a satisfactory rate. [NEED REF] Also these series involve gamma functions which in themselves have a number of numerical difficulties, at least in so much as it is not quite as clear how to evaluate a gamma function as it is a factorial. We discuss the evaluation of the gamma function in section 3.2.

3.1 Mittag-Leffler Function

3.1.1 Integral Representation of $E_{\alpha,\beta}(z)$

We firstly need to develop a contour integral representation of $\Gamma(z)$ and of $\frac{1}{\Gamma(z)}$. The gamma function is defined as

$$\Gamma(z) = \int_0^\infty e^t t^{z-1} dt \tag{27}$$

where integration is done with a real valued t. The idea here is to *lift* this integral into complex plane an then integrate along the contour shown in figure 5. We formalize this idea in the following lemma.

Lemma 12. *A contour integral representation of* $\Gamma(z)$ *is given by*

$$\Gamma(z) = \frac{1}{e^{2\pi i z} - 1} \int_C e^{-t} d^{z-1} dt \tag{28}$$

where C *is the contour depicted in figure 5.*

This proof follows that of Podlubny [6].

Proof. Consider the contour integral

$$\int_C e^{(z-1)\log(t) - t} dt.$$

Due to the fact that the integrand involves a log function it is multivalued unless we specify a branch cut. We will adopt a branch cut along the non-negative real axis. As long as ε is chosen sufficiently small ($\varepsilon < 1$) then Cauchy's theorem guarentees that this contour integral has the same value for all choices of $\varepsilon > 0$ because the integrand has only one singularity at $t = 0$.

Now consider the contour in three pieces, C_ε, the circle of radius ε centered at 0, (ε, ∞), the bottom half of the contour and (∞, ε), the top half of the contour.

As ε can be made arbitarily small we take both straight parts of the contour to be real but on the lower half of the contour we must replace $\log(t)$ with $\log(t) + 2\pi i$.

This results in

$$\begin{aligned}\int_C e^{(z-1)\log(t)-t}dt &= \int_\infty^\varepsilon e^{(z-1)\log(t)-t}dt + \int_{C_\varepsilon} e^{(z-1)log(t)-t}dt + \int_{\varepsilon,\infty} e^{(z-1)\log(t)-t+2\pi i}dt \\ &= \int_\infty^\varepsilon e^{-t}t^{z-1}dt + \int_{C_\varepsilon} e^{-t}t^{z-1}dt + e^{2(z-1)\pi i}\int_\varepsilon^\infty e^{-t}t^{z-1}dt\end{aligned}$$

It can be shown by simple application of the ML lemma, as in [6], that the integral along C_ε goes to zero as $\varepsilon \longrightarrow 0$.

Taking $\varepsilon \longrightarrow 0$ in the other two integrals and rearanging gives us the result

$$\int_0^\infty e^{-t}t^{z-1}dt = \frac{1}{e^{2\pi iz}-1}\int_c e^{-t}t^{x-1}dt \tag{29}$$

□

We would like to exploit this result to give a simple contour integral representation of $\frac{1}{\Gamma(z)}$.

Lemma 13. *A contour integral representation of $\frac{1}{\Gamma(z)}$ is given by*

$$\frac{1}{\Gamma(z)} = \frac{1}{2\pi i}\int_{Ha} e^\tau \tau^{-z}d\tau$$

where the contour Ha is shown in figure 6.

This proof follows that of Podlubny [6]

Proof. By taking the result of 12 and substituting $z-1$ in (29) we get that

$$\int_C e^{-t}t^{-z}dt = (e^{-2\pi iz}-1)\Gamma(1-z).$$

Performing the substitution $\tau = -t$ rotates the contour integral clockwise 180 degrees and results in

$$\int_C e^{-t}t^{-z}dt = -e^{-z\pi i}\int_{Ha} e^\tau \tau^{-z}d\tau$$

and so

$$\begin{aligned}\Gamma(1-z) &= \frac{1}{e^{z\pi i}-e^{-z\pi i}}\int_{Ha} e^\tau \tau^{-z}d\tau \\ &= \frac{1}{2i\sin(\pi z)}\int_{Ha} e^\tau \tau^{-z}d\tau.\end{aligned}$$

Now taking into account the well known Euler reflection formula, $\Gamma(z)\Gamma(1-z) = \frac{\pi}{\sin(\pi z)}$, we get

$$\frac{1}{\Gamma(z)} = \frac{1}{2\pi i}\int_{Ha} e^\tau \tau^{-z}d\tau.$$

□

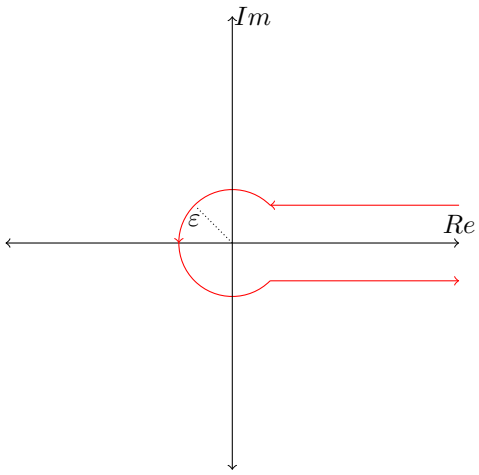


Figure 5: The Hankel contour C

□

We now wish to give a pair of contour integral representations of the function $\frac{1}{\Gamma(z)}$ which we will then use in a pair of contour integrals representing $E_{\alpha,\beta}(z)$.

Lemma 14. *The function* $\frac{1}{\Gamma(z)}$ *can be represented by the following contour integrals*

$$\frac{1}{\Gamma(z)} = \frac{1}{2\pi\alpha i}\int_{\gamma(\varepsilon,\mu)} e^{\zeta^{1/\alpha}}\zeta^{(1-z-\alpha)/\alpha}d\zeta \tag{30}$$

where

$$\left(\alpha < 2, \frac{\pi\alpha}{2} < \mu < \min\{\pi, \pi\alpha\}\right) \tag{31}$$

or where $Re(z) > 0$

$$\frac{1}{\Gamma(z)} = \frac{1}{4\pi i}\int_{\gamma(\varepsilon,\pi)} e^{\zeta^{1/2}}\zeta^{-(z+1)/2}d\zeta \tag{32}$$

where the contours $\gamma(\varepsilon,\varphi)$ *are depicted in figure 8.*

This proof follows that of Podlubny [6].

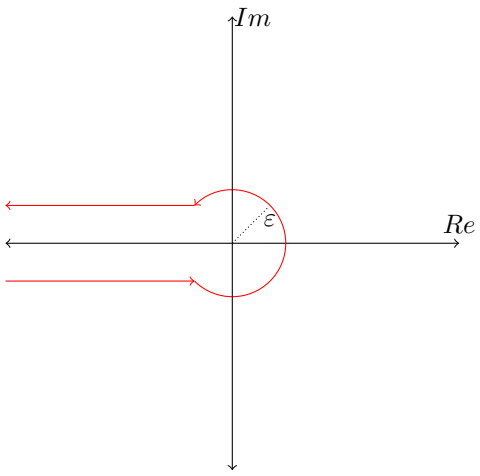


Figure 6: The Hankel contour Ha

Proof. □

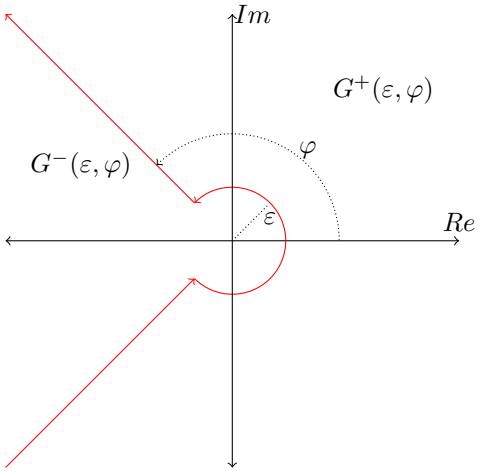
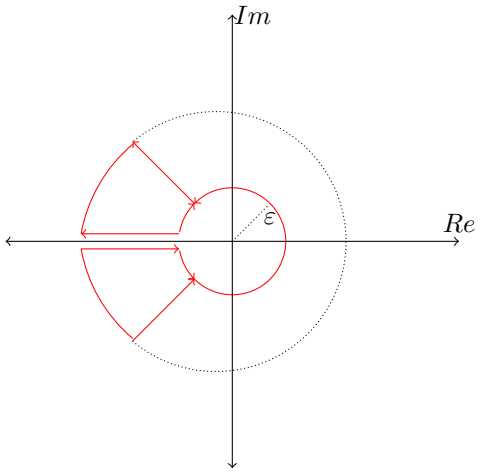


Figure 7: The Hankel countour $\gamma(\varepsilon, \varphi)$

3.2 Additional Notes on the Numerical Evaluation of the Gamma Function



These notes are included because the numerical evaluation of the gamma function is important regardless of whether it is actually used in the above methods for evaluation of the Mittag-Leffler and Wright functions. For example the Lanczos approximation is used in the scheme outlined in [INSERT CC REF] and implemented in code available in Appendix A.

3.2.1 Stirling's Approximation

3.2.2 Lanczos Approximation

3.2.3 Spouge Approximation

The Spouge approximation, like teh Lanczos approximation

4 Appendix A: Adams-Moulton-Bashforth FDE Solver Code

4.1 C# Implementation of AMB FDE Solver

4.1.1 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;
                }


        }
}
```

4.1.2 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;
                }
        }
}
```

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

4.2.1 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;
}
```

5 Appendix B

5.1 Runtimes for C# implementation described in section 2.1.4

N \ p	1	2	3	4	5	5	7	8	9	10	11	12
1000	0.4070868	0.2362202	0.1191832	0.1293034	0.0938632	0.1450464	0.1427508	0.1377472	0.133987	0.125388	0.1216968	0.1200818
2000	1.0937832	0.558851	0.556602	0.46559	0.430715	0.403747	0.4425184	0.4152856	0.4051934	0.3774052	0.3794404	0.3809334
3000	1.9052022	1.4103498	1.0329988	0.895754	0.883083	0.8648174	0.8972584	0.820875	0.813499	0.7662662	0.7725402	0.7604456
4000	3.6316786	2.5111496	1.9091866	1.5481538	1.4340788	1.4993042	1.5020828	1.3928906	1.3221538	1.27441	1.2869578	1.263289
5000	6.2298358	3.7344554	2.9771892	2.3199244	2.1266484	2.220891	2.2675012	2.068062	1.980582	1.9098458	1.921069	1.9492734
6000	7.4398042	4.9997078	4.135724	3.1844926	2.9543306	3.1121518	3.1542058	2.8458522	2.7509526	2.6820768	2.6935614	2.6925528
7000	10.055285	6.8022498	5.2268638	4.3256554	3.9238724	4.0993682	4.2330766	3.845559	3.6778138	3.5685162	3.5791386	3.5445644
8000	12.8123234	7.6822148	7.023647	5.5118882	4.962013	5.1901768	5.3650102	4.9373976	4.7257116	4.549266	4.6082242	4.571809
9000	16.2015754	10.5661146	8.3860596	6.959551	6.2022426	6.3978686	6.6804982	6.1248574	5.9008362	5.6928986	5.8067794	5.600407
10000	22.8870988	12.3470092	10.2837944	8.575945	7.5265192	7.7042254	8.1302114	7.4418008	7.2273678	6.9739174	6.9767076	6.7921516

5

5.2 Runtimes for CUDA implementation described in section 2.1.7

N \ opsPerThread	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
50,000	4.87108	3.32297	3.37962	3.46946	3.64845	3.87464	4.13553	4.43697	4.57786	4.71845	4.85645	4.99309	5.12561	5.2565	5.38823	5.58507	5.87955	6.1915	6.5155	6.85466
100,000	13.7332	9.2564	8.29588	7.51481	7.89867	8.36102	8.77676	9.19788	9.52294	9.85837	10.1958	10.5321	10.8622	11.1902	11.5218	11.9984	12.7003	13.4505	14.2386	15.069
150,000	27.9077	17.7804	14.7961	14.2085	13.8015	13.4937	14.4337	15.507	15.7728	15.9378	16.0039	16.1438	16.6539	17.1985	17.7416	18.5025	19.609	20.7922	22.0386	23.3795
200,000	47.0168	28.939	23.8399	21.6257	21.6106	21.677	21.8518	22.1316	22.628	23.2031	23.6663	24.039	24.3042	24.5237	24.7081	25.3161	26.8756	28.4815	30.2146	32.0608
250,000	71.0519	42.6913	34.5362	31.5671	30.0385	30.939	31.7693	32.404	31.6023	30.755	31.4151	32.2341	32.9353	33.7153	34.6383	35.7562	38.0052	39.8973	41.2286	42.9506
300,000	100.033	59.0803	46.8046	42.238	41.1102	40.6073	42.502	44.0988	43.1857	42.2975	41.477	40.7721	41.6668	43.0575	44.9286	46.8037	51.3096	54.6343	57.8365	60.6987
350,000	133.784	77.9304	61.372	55.2624	53.0512	53.5426	53.5646	56.3827	56.1513	55.4469	54.2999	53.8824	53.206	53.5035	56.1716	58.2979	64.7001	69.3878	74.0164	78.8297
400,000	172.509	99.4371	77.6313	69.2178	66.592	67.1919	68.302	69.2099	69.3106	69.2914	68.8276	68.4649	69.3828	70.6512	72.3131	71.232	79.6719	84.8636	90.7919	97.2622
450,000	215.96	123.333	95.4081	85.2786	81.7588	81.2605	83.8495	86.0194	82.8356	83.161	83.6226	83.8407	86.6134	90.099	94.5162	91.1048	102.314	105.757	110.677	116.999
500,000	264.41	149.922	115.271	102.267	97.4631	98.4406	99.3867	103.704	100.777	97.5023	98.3143	99.3144	103.798	109.351	117.381	113.081	131.434	135.395	139.066	143.586
550,000	317.531	178.85	136.995	121.337	115.664	116.319	117.559	121.546	119.587	116.542	113.796	115.009	121.197	129.498	141.149	135.374	160.621	167.084	173.385	179.086
600,000	375.574	210.454	160.125	141.451	134.63	134.439	137.721	139.959	138.566	136.459	133.901	131.22	138.704	148.854	164.37	158.017	190.778	200.624	209.304	218.444
650,000	438.333	244.445	185.481	163.521	155.259	155.915	157.971	162.504	157.517	156.625	155.25	152.742	157.018	169.943	189.083	181.18	220.261	233.223	245.357	257.474
700,000	506.486	281.27	212.58	186.674	177.392	177.773	178.82	185.737	179.677	176.794	176.563	175.212	179.988	189.953	212.321	204.599	251.396	267.425	281.956	297.777
750,000	578.928	320.209	241.267	211.658	200.474	200.262	203.419	208.98	204.186	197.723	198.007	197.835	204.805	217.196	239.112	228.087	281.324	300.706	318.765	337.085
800,000	656.638	362.042	271.795	237.863	225.392	225.654	228.412	233.173	229.111	223.378	219.488	220.567	229.216	243.557	267.325	253.588	314.309	336.544	356.23	378.126
850,000	738.779	406.038	304.418	265.773	251.271	251.736	253.47	261.266	253.93	249.597	244.374	243.305	254.169	272.859	301.262	284.64	348.23	372.77	394.878	418.79
900,000	826.216	452.979	338.473	295.148	279.101	278.412	281.179	290.1	279.968	275.995	272.18	267.148	279.3	299.379	331.805	315.486	389.705	413.51	436.016	462.42
950,000	918.002	501.929	374.475	326.098	308.433	308.072	311.171	318.883	310.039	302.488	300.025	295.68	305.017	329.237	366.771	349.544	430.481	458.217	480.702	508.34
1,000,000	1015.2	553.967	412.362	358.399	338.406	337.998	340.872	348.722	340.862	330.045	328.021	325.068	332.821	355.923	398.319	381.349	475.024	507.402	530.949	560.252

6

[5]The times listed are in seconds and are calculated as the average over 5 runs.
[6]The times listed are in seconds and are for one run.

References

[1] Microsoft Copropration. Task schedulers. `http://msdn.microsoft.com/en-us/library/dd997402%28v=vs.110%29.aspx`. Accessed: 2014-08-01.

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

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

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

[5] M. Harris. Mapping computational concepts to gpus. In M. Pharr, R. Fernando, and T. Sweeney, editors, *GPU Gems 2*. Addison-Wesley Professional, 2005.

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

[7] S.G. Samko, A.A. Kilbas, and O.I. Marichev. *Fractional Integrals and Derivatives*. Breach Science Publishers, 1993.

[8] C.C. Tisdell. On the application of sequential and fixed-point methods to fractional differential equations of arbitrary order. *Journal of Integral Equations*, 24, 2012.