BEL–Math
A .NET Computational Package Written in Zonnon
Part III of the Users Guide

Alan D. Freed
Clifford H. Spicer Chair in Engineering
College of Science, Engineering & Technology
Saginaw Valley State University
202 Pioneer Hall, 7400 Bay Road
University Center, MI 48710
E-Mail: `adfreed@svsu.edu`

November 9, 2013

Abstract

This document describes some useful math modules that are part of the BEL software library written in the Zonnon programming language for the .NET and Mono frameworks. BEL provides a simple, computational, programming interface that resides on top of Zonnon. Its purpose is to assist in rapid program development, specifically in the area of computational continuum mechanics, by leveraging the wide diversity of existing .NET and Mono libraries with the simple, logical constructs of the Zonnon programming language.

1 Introduction

BEL is an acronym taken for my research laboratory: The Biological Engineering Laboratory at Saginaw Valley State University. What BEL is and what it provides are quite different from what one might expect, given this affiliation. BEL arose out of the author's desire to interface with the .NET and Mono Frameworks for the purpose of writing computational software programs in the Zonnon programming language [1, 2].

Zonnon is the most recent descendant from the family tree Euler/Algol/Pascal/Modula/Oberon of programming languages developed by Profs. Niklaus Wirth and Jürg Gutknecht from the Computer Systems Institute at ETH Zurich (Eidgenössische Technische Hochschule, Zentrum, i.e., the Swiss Federal Institute of Technology, Zurich). Zonnon was created specifically for .NET, with compiler versions available for both the .NET and Mono Frameworks. Sponsored by Xamarin, Mono is an open source implementation of Microsoft's .NET Framework based on the ECMA standards for C# and the Common Language Runtime. The Zonnon compiler targets the .NET Framework 2.0.

The Zonnon compiler, documentation, example programs, and even BEL itself, are free to download from `http://www.zonnon.ethz.ch`. The home page for Microsoft's .NET Framework is at `http://www.microsoft.com/net`. Mono's is at `http://www.mono-project.com/Main_Page`.

Licensing of the BEL library, both its software and documentation, is addressed in the *Licensing of the Software and its Documentation* part to this user guide.

1.1 Version

This document corresponds to software version 4.2 of BEL-Math, November 9, 2013. Zonnon targets the .NET Framework 2.

1.2 Required Libraries

The only external library that is required to compile BEL-Math is the one that you created when you built BEL. When compiling Bel-Math it is important to place file `interpolations.znn` first in the list of files because it exports types that are used by the other files in this library.

1.3 Known Issues

There are no know issues regarding BEL-Math. Having said this, of all the libraries in the BEL suite, this is the one most likely to be added to over time. One is always running into new math techniques, algorithms, solvers, etc. that will warrant their addition to this library.

1.4 Feedback

I consider this to be a living document. If there is some aspect that is wanting, or if you have corrections and/or additions that you believe would benefit other users of BEL-Math, please forward them to `adfreed@svsu.edu`.

2 Mathematics

BEL contains a modest set of math libraries. They are split into two topical regimes: functions and the calculus. Their interfaces are provided in App. A. The basic trigonometric, logarithmic, and hyperbolic functions reside in module Bel.Math, which is now part of the core BEL library.

The author has written a variety of other math procedures in Oberon that he would like to port over to BEL-Math in future releases.

2.1 Special Functions

Module `BelMath.SpecialFunctions` provides a variety of math functions that are more specialized than one would find in a typical math library, which are otherwise found in `Bel.Math`. The special functions in this library are:

Procedure `Erf` returns the error function of its argument. The error function $\mathrm{erf}(x)$ is defined by

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \mathrm{e}^{-y^2}\,\mathrm{d}y$$

with a graphical presentation displayed in Fig. 1.

Procedure `Erfc` returns the complimentary error function of its argument. The complementary error function $\mathrm{erfc}(x)$ obeys

$$\mathrm{erf}(x) + \mathrm{erfc}(x) = 1$$

with a graphical presentation displayed in Fig. 1.

Procedure `Gamma` returns the gamma function of its argument. The gamma function $\Gamma(x)$ is defined by

$$\Gamma(x) = \int_0^\infty y^{x-1}\mathrm{e}^{-y}\,\mathrm{d}y$$

with a graphical presentation displayed in Fig. 2. The gamma function generalizes the factorial in that $\Gamma(n+1) = n!$ whenever $n \in \mathbb{Z}$.

Procedure `Beta` returns the beta function for its two arguments. The beta function $B(m,n)$ is defined by

$$B(m,n) = \frac{\Gamma(m)\,\Gamma(n)}{\Gamma(m+n)}$$

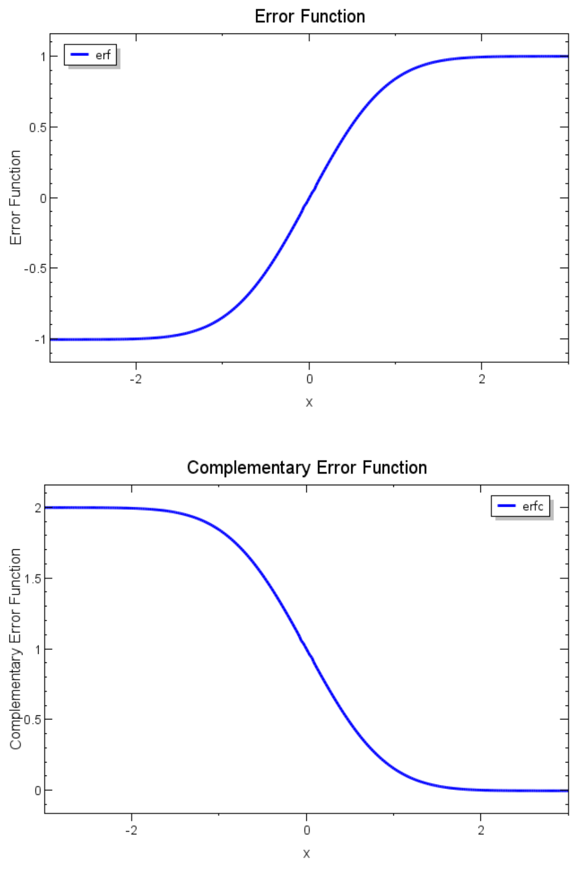

Figure 1: The error function is on top. The complementary error function is below.

which generalizes the product of two factorials.

Procedure `MittagLeffler` returns the two-parameter Mittag-Leffler function $E_{\alpha,\beta}(x)$ for parameter ranges of $0 < \alpha \leq 1$ and $\beta = 0$ or 1. The

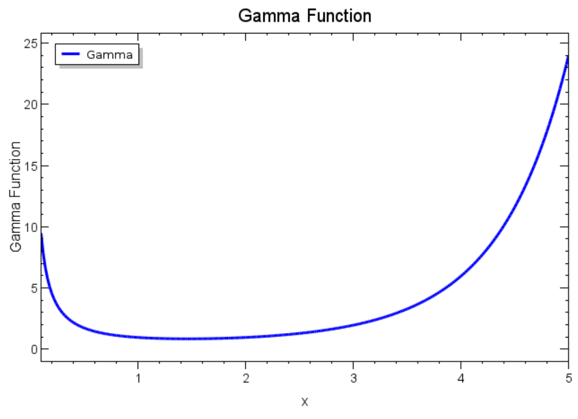

Figure 2: The gamma function.

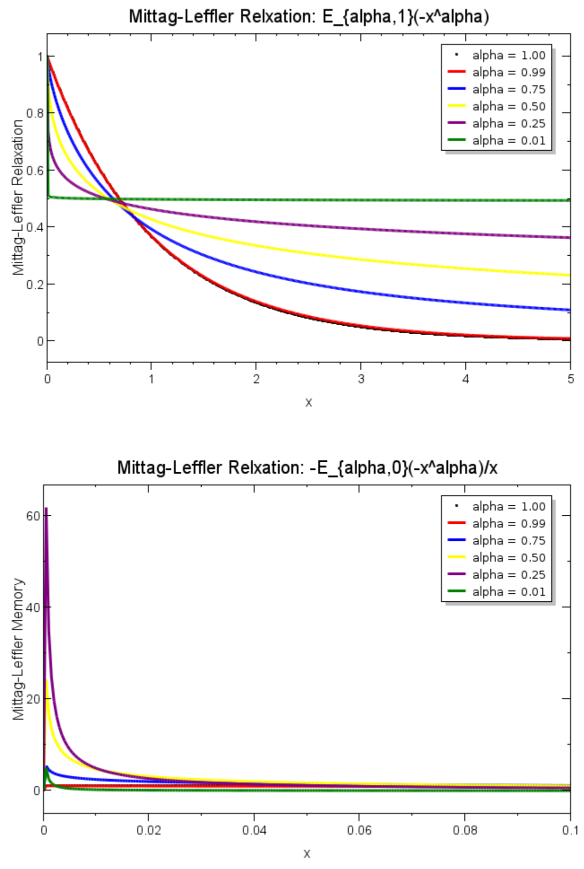

Figure 3: The Mittag-Leffler function as a relaxation function in viscoelasticity $(E_{\alpha,1}(-x^{\alpha}))$ on top, and as a memory function $(-E_{\alpha,0}(-x^{\alpha})/x)$ below.

Mittag-Leffler function is defined by

$$E_{\alpha,\beta}(x) = \sum_{0}^{\infty} \frac{x^k}{\Gamma(\alpha k + \beta)}$$

with graphs for $E_{\alpha,\beta}(x)$ of relevance to the author being shown in Fig. 3.

2.2 Interpolations

Module `BelMath.Interpolations` provides several functions for interpolating between data points.

NEVILLE's algorithm provides an efficient implementation of LAGRANGE interpolation. If vector $\mathscr{x}$ represents inputs and vector $\mathscr{y}$ its outputs, with arrays `xVec` and `yVec` holding their values, then `Neville` returns the LAGRANGE interpolation for value y at some intermediate point x residing within the range of $\mathscr{x}$.

A different approach to interpolation is that of Hermite interpolation. Procedure `Hermite` implements a cubic interpolator where `xLo` and `xHi` bound the interval of interpolation in the independent varibable x; `yLo` and `yHi` hold arrays for the dependent variables $\mathbb{y}$ at these limits, i.e., `yLo` $= \mathbb{y}(\texttt{xLo})$ and `yHi` $= \mathbb{y}(\texttt{xHi})$; and `dyLo` and `dyHi` hold the arrays for the gradient of the dependent variables $\mathrm{d}\mathbb{y}/\mathrm{d}x$ at these limits, i.e., `dyLo` $= \mathrm{d}\mathbb{y}(\texttt{xLo})/\mathrm{d}x$ and `dyHi` $= \mathrm{d}\mathbb{y}(\texttt{xHi})/\mathrm{d}x$. The location for interpolation is sent via the argument `x`, with a cubic interpolation being returned by procedure `Hermite`, viz., $\mathbb{y}(\texttt{x})$.

Two-dimensional interpolation over a rectangular grid is accommodated with `Bilinear`. Supplied values are the x and y grid coordinates, with `x1 < x2` and `y1 < y2`; their held values `valX1Y1, valX1Y2, valX2Y1` and `valX2Y2`; and the grid coordinates where the interpolation is sought: `atX` and `atY`, with `x1` $\le$ `atX` $\le$ `x2` and `y1` $\le$ `atY` $\le$ `y2`.

2.3 Distributions

`BelMath.Distributions` has, at present, three distributions in it: the chi-squared distribution, the Student's t-distribution, and the F distribution.

This module exports an enumerated type, `Certainty`, that has values: `ninety`, `ninetyFive`, `ninetySevenPointFive`, `ninetyNine` and `ninetyNinePointFive`. For any of these percentage point certainties, one can get the `StudentT` or `F` distribution values for any degree of freedom, or the `ChiSquaredLow` and `ChiSquaredHigh` distribution values up through 100 degrees of freedom.

Given that $x_1, x_2, \ldots, x_\nu$ are ν, independent, normally distributed, standardized, random variables (i.e., they have a mean of zero, and a variance of one), then $X^2 = \sum_{n=1}^{\nu} x_i^2$ is said to follow a *chi-squared* distribution in ν degrees of freedom, with the probability that $X^2 \le \chi^2$ being given by $P(\chi^2|\nu)$. `ChiSquaredLow` returns $\chi^2_\alpha(\nu)$ and `ChiSquaredHigh` returns $\chi^2_{1-\alpha}(\nu)$ for probabilities (or percentage-point certainties) of $(1-\alpha)100\%$.

Given that X is a normally distributed, standardized, random variable, and that χ^2 is a random variable following an independent, chi-squared distribution in ν degrees of freedom, then the probability $P\left(\left|X/\sqrt{\chi^2/\nu}\right| \le t\right)$ is Student t distributed. `StudentT` returns $t_\alpha(\nu)$ for probabilities (or percentage-point certainties) of $(1-\alpha)100\%$.

Given that X_1^2 and X_2^2 are two, independent, random variables distributed chi-squared in ν_1 and ν_2 degrees of freedom, respectively, then the variance ratio $F = (X_1^2/\nu_1)/(X_2^2/\nu_2)$ is distributed F with distribution function $P(F|\nu_1,\nu_2)$. `F` returns $F_\alpha(\nu_1,\nu_2)$ for probabilities (or percentage-point certainties) of $(1-\alpha)100\%$. The ordering of ν_1 and ν_2 matters. An important reflexive property of the F-distribution is that $F_{1-\alpha}(\nu_1,\nu_2) = 1/F_\alpha(\nu_2,\nu_1)$.

2.4 Random Variates

`BelMath.RandomVariates` provide random numbers that are distributed according to specified distributions. Procedure `Normal` returns a random normal variable with a mean of zero and a standard deviation of one. Procedure `Exponential` returns a random exponential variable with a mean and standard deviation of one. They implement the ziggurat method of Marsaglia & Tsang [3].

2.5 Regression

Least-squares regression is a technique that can be used to fit some basic models to experimental data, which is what the routines in module `BelMath.Regression` do. Given a vector of inputs $\mathbb{x}$ to which there corresponds a vector of outputs $\mathbb{y}$ of like dimension, procedure `Linear` fits these data to the function $\mathbb{y}_i = k_0 + k_1\mathbb{x}_i$, while procedure `Quadratic` fits these same data to the function $\mathbb{y}_i = k_0 + k_1\mathbb{x}_i + k_2\mathbb{x}_i^2$, where k_0, k_1 and k_2 are the fit parameters.

It is often useful to map the data into a range where the data can be fit by a linear or quadratic least-squares process. Several mappings to linear fits are provided: Procedure `Exponential` fits the data to function $\mathbb{y} = k_0\exp(k_1\mathbb{x})$, while procedure `PowerLaw` fits them to $\mathbb{y} = k_0\mathbb{x}^{k_1}$, and `StretchedExp` fits them to $\mathbb{y} = \exp(k_0\mathbb{x}^{k_1})$. The quadratic model can be applied to mapped arguments for $\mathbb{x}$ and/or $\mathbb{y}$, too, but no precanned procedures are provided for such models. The users will have to create such maps for themselves.

All regression procedures return the R^2 statistic for goodness of fit, a.k.a. the coefficient of determination. For the exponential model, the R^2 statistic is calculated in an x vs. $\log y$ plot of the data. For the power-law model, the R^2 statistic is calculated in an $\log x$ vs. $\log y$ plot of the data. And for the stretched-exponential model, the R^2 statistic is calculated in an $\log x$ vs. $\log(\log y)$ plot of the data.

2.6 Roots

`BelMath.Roots` provides algorithms for solving $f(x) = 0$, viz., to find a value for x such that function f evaluated at x gives zero. This module exports two procedure types: `F` is the type in which the function f is to be supplied and, when needed, `J` is the type that its Jacobian, i.e., $\mathrm{d}f/\mathrm{d}x$, is to be supplied in.

Newton's method, exported as the procedure `Newton`, is the predominant algorithm used to find roots of a function. Its first argument is an instance of type `F`, or the function whose roots are to be found. The second is of type `J`, or the function for acquiring the Jacobian of function `F`. The final argument of `Newton` is the argument x of function f, whose sent value is an initial guess at this root, and whose returned value is the actual root. Newton's method needs a good initial guess at where a particular root may be, otherwise it is likely to become unstable. Multiple roots are common, often residing at different locations x, so choosing an initial guess close to the desired root is an important consideration when using `Newton`.

Newton's method requires a Jacobian. If an analytic Jacobian is not available, for whatever reason, then procedure `NumericalJacobian` can be called from within the programmer's implementation of procedure type `J` to be used to estimate its value. In addition to the interface of procedure type `J`, `NumericalJacobian` requires an instance of type `F` to be passed as an argument. This procedure estimates the Jacobian using a midpoint difference. Alternatively, if the supplied Jacobian for procedure `Newton` is `nil` then `NumericalJacobian` is called internally to approximate the actual Jacobian.

3 Calculus

A variety of numerical methods have been implemented for approximating the various operators of the calculus; in particular: algorithms for approximating derivatives, integrals, ordinary differential equations (ODE), and convolution integrals are provided.

3.1 Derivatives

Module `BelMath.Derivatives` exports the enumerated type `Method` from which a programmer can choose one of four methods for numerically differentiating a user-supplied function. These four choices are: `forward`, `central`, `backward` and `richardson`. Procedure `Differentiate` estimates $\mathrm{d}y(x)/\mathrm{d}x$. It has arguments that include the user function to be differentiated '`y`' (an instance of exported procedure type `Y`), the value of the independent variable where this derivative is to be estimated '`x`', the step size that is to be used in the approximation '`h`', and the method to be used for approximation '`m`'. Only method `richardson` subdivides the interval to improve upon accuracy.

Sometimes one knows what step size h to use, but typically one just wants that step size which when used will enable the most accurate estimate possible for a given numerical method. A call to procedure `OptimalStepSize` returns such an h. This will depend upon the value of the independent variable x at which the derivative is to be gotten, which is passed as argument `atX`.

3.2 Integrals

`BelMath.Integrals` also exports an enumerated type `Method` from which the programmer can now choose one of five methods for numerically integrating a user-supplied function. The choices include: `trapezoidal`, `simpson`[1], `threeEights`,

1. Simpson's rule, as a method for approximating volumes,

`romberg` and `gauss`. Procedure `Integrate` estimates $\int_a^b f(x)\ \mathrm{d}x$. It has arguments that include the user function to be integrated '`f`' (an instance of exported procedure type `F`), the lower- and upper-limits of integration '`a`' and '`b`', and the method of approximation '`m`'. Only methods `romberg` and `gauss` subdivide the interval to improve upon accuracy.

3.3 Runge-Kutta ODE Solvers

Given some ordinary differential equation (ODE) $\mathrm{d}y/\mathrm{d}x = f(x, y)$, a Runge-Kutta (RK) integrator seeks to acquire an estimate for its solution by getting estimates $y_i = y_n + h\sum_{j=1}^{s} \mathrm{A}_{ij} F_j$ at s stage locations $x_i = x_n + \mathrm{c}_i h$ across the interval $[x_n, x_{n+1}]$. Specifically, stage derivatives F_i are evaluated at quadrature positions x_i according to the formula

$$F_i = f\left(x_n + \mathrm{c}_i h, y_n + h\textstyle\sum_{j=1}^{s}\mathrm{A}_{ij} F_j\right),$$

with its quadrature of integration being given by

$$y_{n+1} = y_n + h\sum_{i=1}^{s} \mathrm{b}_i F_i.$$

Specific Runge-Kutta integrators are typically quantified by their individual Butcher tableau [4]

$$\begin{array}{c|c} \boldsymbol{c} & \mathbf{A} \\ \hline & \boldsymbol{b}^T \end{array}.$$

A method will be explicit whenever its $\mathbf{A}$ matrix is lower triangular, as are the cases addressed below.

At the suggestion of Prof. John Butcher, Richardson extrapolation is employed to arrive at an asymptotically correct estimate of the local truncation error. By using the same integrator twice, the first time advancing a solution y_1 with a single step size of h, and the second time by advancing a solution y_2 from two successive half-steps of size $h/2$, then the truncation error is given by

$$\mathrm{err} = \frac{\|y_2 - y_1\|}{1 - 2^{-p}} + O(h^{p+2}),$$

where p is the order of the method, e.g., 4 for Kutta's original method. This error is held to a tolerance specified by the user at the time of `SetUp` by employing Söderlind's [7, 5] PI control technology to dynamically adjust the step-size for optimum performance.

Module `BelMath.RungeKutta` implements five, explicit, Runge-Kutta integrators, each with a different order of accuracy, chosen by selecting an instance of enumerated type `Order`, which includes options: `second`, `third`, `fourth`, `fifth` and `sixth`. The second-order method uses the quadrature and weights of trapezoidal integration, as specified in Table 1. The third-order method employees Simpson's quadrature and weights, as specified in Table 2. The fourth-order method is Kutta's most accurate integrator [6], which is based on Simpson's 3/8th rule, and is listed in Table 3. While the fifth- and sixth-order methods of Tables 4 & 5 were taken

0	0	0
1	1	0
	1/2	1/2

Table 1: Trapezoidal rule.

0	0	0	0
1/2	1/2	0	0
1	−1	2	0
	1/6	2/3	1/6

Table 2: A third-order Runge-Kutta integrator that utilizes the weights and quadrature points of Simpson's rule, a.k.a. Kepler's cask rule.

can be traced back to the works of Johannes Kepler, about one hundred years before Thomas Simpson was born. Kepler used this algorithm to estimate the volume of wine casks.

0	0	0	0	0
1/3	1/3	0	0	0
2/3	−1/3	1	0	0
1	1	−1	1	0
	1/8	3/8	3/8	1/8

Table 3: Kutta's 3/8th rule. His most accurate Runge-Kutta method.

0	0	0	0	0	0	0	0
1/3	1/3	0	0	0	0	0	0
2/3	0	2/3	0	0	0	0	0
1/3	1/12	1/3	−1/12	0	0	0	0
5/6	25/48	−55/24	35/48	15/8	0	0	0
1/6	3/20	−11/24	−1/8	1/2	1/10	0	0
1	−261/260	33/13	43/156	−118/39	32/195	80/39	0
	13/200	0	11/40	11/40	4/25	4/25	13/200

Table 5: A sixth-order Runge-Kutta method derived by Butcher [4, pg. 194].

0	0	0	0	0	0	0
1/4	1/4	0	0	0	0	0
1/4	1/8	1/8	0	0	0	0
1/2	0	0	1/2	0	0	0
3/4	3/16	−3/8	3/8	9/16	0	0
1	−3/7	8/7	6/7	−12/7	8/7	0
	7/90	0	16/45	2/15	16/45	7/90

Table 4: A fifth-order Runge-Kutta method derived by Butcher [4, pg. 191].

from Butcher's textbook on the subject [4]. This module exports a `Bel.Entity` object called `Integrator` that preforms Runge-Kutta integration.

Module `BelMath.RungeKutta` also exports a procedure type `F`. The user must cast their system of ordinary differential equations (ODEs) in terms of this interface. Method `SetUp` of type `Integrator` is called at the start of a solution at which time one sends: the ODE to be solved '`f`' (an instance of procedure type `F`), the size of the initial time step '`h`', the error tolerance '`tol`' (usually set to a value of 10^{-p} where p is the order of the method), and finishing with '`ord`' (the order p of the method). As a rule-of-thumb, for two significant figures of accuracy, set `tol` to 0.01 and choose the RK integrator with `ord` set to `second`. For three significant figures of accuracy, set `tol` to 0.001 and choose the RK integrator with `ord` set to `third`, etc.

There are three **`var`** arguments in method `Integrate`. The first is the independent variable `x`, the second is an array holding the dependent variables `y`, and the last of these fields provides an asymptotically correct estimate for the local truncation error, `err`. Arguments `x` and `y` are input/output variables, while argument `err` is strictly an output variable. A call to this method advances the solution by a single step, whose magnitude is controlled internally by a proportional integral (PI) controller [7, 5].

The exported variable `functionEvaluations` specifies the number of function evaluations required to obtain a solution, while `integrationSteps` keeps track of the number of successfully completed integration steps. The solver will automatically restart itself with a reduced step size whenever an asymptotic error estimate exceeds the error tolerance specified by the user. When this happens, it will increment the `restarts` counter. These counters are reset by `Initialize` and `SetUp`.

3.4 General Linear ODE Solvers

Inherent Runge-Kutta stable (IRKS) methods are a sub-class to a larger class of numerical methods known as general linear (GL) methods. GL methods seek to capture the best attributes from *multi--stage* Runge-Kutta methods, like those presented in §3.3, and combine these with the best attributes from *multi-step* Adams-Bashforth/Moulton methods. The mathematics underlying these methods are sophisticated, even daunting at times, the details of which can be studied in Butcher's [4] textbook by the interested reader. Only an overview is presented herein, sufficient for a programmer to understand how these integrators work, conceptually, and how one would go about implementing them into code, if need be. Two, third-order, IRKS methods taken from the author's book [8, App. D] are presented here. IRKS methods that range from order two to order six can be found at the website `www.math.auckland.ac.nz/~hpod/atlas`. The methods from that website are not self starting.

In GL methods, r quantities $y_1^{[n-1]}$, $y_2^{[n-1]}$, $\ldots$, $y_r^{[n-1]}$, which are the history or multi-step values acquired over previous integration steps, are updated via integration from step $n-1$ to step n, thereby producing a subsequent set of new values $y_1^{[n]}$, $y_2^{[n]}$, $\ldots$, $y_r^{[n]}$, each being an array of dimension N, the number of simultaneous equations being solved. During the multi-stage integration from step $n-1$ to step n, s stage values $\mathbb{Y}_1$, $\mathbb{Y}_2$, …, $\mathbb{Y}_s$ are computed over this interval, along with their s stage derivatives $\mathbb{F}_1$, $\mathbb{F}_2$, …, $\mathbb{F}_s$. These resulting arrays are assembled into superarrays with Nordsieck scaling, i.e., the values over the past r steps going back into the history are brought forward to the current step as a truncated Taylor series expansion of the form

$$\mathbf{y}^{[n]} = \begin{Bmatrix} y_1^{[n]} \\ y_2^{[n]} \\ y_3^{[n]} \\ \vdots \\ y_r^{[n]} \end{Bmatrix} = \begin{Bmatrix} y(x_n) \\ h y'(x_n) \\ h^2 y''(x_n) \\ \vdots \\ h^{r-1} y^{(r-1)}(x_n) \end{Bmatrix},$$

$$\mathbf{Y} = \begin{Bmatrix} \mathbb{Y}_1 \\ \mathbb{Y}_2 \\ \mathbb{Y}_3 \\ \vdots \\ \mathbb{Y}_s \end{Bmatrix}, \qquad \mathbf{F} = \begin{Bmatrix} \mathbb{F}_1 \\ \mathbb{F}_2 \\ \mathbb{F}_3 \\ \vdots \\ \mathbb{F}_s \end{Bmatrix},$$

where $y' = \mathrm{d}y/\mathrm{d}x$, etc., with $\mathbf{y}^{[n]}$ being rN in size, and $\mathbf{Y}$ and $\mathbf{F}$ both being sN in size, while $h = x_n - x_{n-1}$ is the step size of integration (x being the independent variable of integration, which is usually time). For the third-order IRKS integrators presented in BEL, $r = s = 4$.

GL methods are comprised of four matrices, viz., $\mathbf{A} = [\mathrm{A}_{ij}]_{s\times s}$, $\mathbf{U} = [\mathrm{U}_{ij}]_{s\times r}$, $\mathbf{B} = [\mathrm{B}_{ij}]_{r\times s}$ and $\mathbf{V} = [\mathrm{V}_{ij}]_{r\times r}$, that appear in a special linear system of equations used to update the state; specifically,

$$\mathbb{Y}_i = h \sum_{k=1}^{s} \mathrm{A}_{ik} \mathbb{F}_k(\mathbb{Y}_k) + \sum_{k=1}^{r} \mathrm{U}_{ik} y_k^{[n-1]}, \qquad i = 1, 2, \ldots, s,$$

and

$$y_i^{[n]} = h \sum_{k=1}^{s} \mathrm{B}_{ik} \mathbb{F}_k(\mathbb{Y}_k) + \sum_{k=1}^{r} \mathrm{V}_{ik} y_k^{[n-1]}, \qquad i = 1, 2, \ldots, r,$$

where matrix $\mathbf{A}$ contains stage coefficients, akin to a Runge-Kutta method; matrix $\mathbf{B}$ contains step coefficients, akin to Adams-Bashforth/Moulton methods; while matrices $\mathbf{U}$ and $\mathbf{V}$ couple these processes together into a single integration algorithm.

The stage derivatives $\mathbb{F}_k(x + c_k h, \{\mathbb{Y}_i\})$, with $i, k = 1, 2, \ldots, s$, depend upon the set of stage values $\{\mathbb{Y}_i\}$. These derivatives are computed at quadrature points c_k along the interval of integration $[x, x+h] \equiv [x_{n-1}, x_n]$. The sum on index k over the Runge-Kutta–like A_{ik} coefficients terminates at $i-1$ for explicit methods, at i for diagonally implicit methods (like the methods presented below), or at s for fully implicit methods, which are neither discussed nor implemented herein.

3.4.1 Estimate Error

The Butcher matrix for a GL method is a partitioned $(s + r) \times (s + r)$ matrix of the form

$$\left[\begin{array}{c|c} \mathbf{A} & \mathbf{U} \\ \hline \mathbf{B} & \mathbf{V} \end{array}\right].$$

Property F, a.k.a. a first-same-as-last method in the RK literature, is a desirable property for an IRKS method to have in that it enables one to compute an asymptotically accurate estimate for the local truncation error. It is readily apperent by looking at a method's Butcher matrix if it can or cannot possess this property. If it can, then the last row in the $[\mathbf{A}|\mathbf{U}]$ matrix will be identical to the first row in the $[\mathbf{B}|\mathbf{V}]$ matrix, and the second row in the $[\mathbf{B}|\mathbf{V}]$ matrix will be all zeros, except for the last column in matrix $\mathbf{B}$ where there will be a one.

$$\left[\begin{array}{cccc|cccc}
0 & 0 & 0 & 0 & 1 & 0.25 & 0.0625 & 0.015625 \\
0.845232 & 0 & 0 & 0 & 1 & -0.345232 & -0.172616 & -0.033481 \\
-0.0449691 & 0.882754 & 0 & 0 & 1 & -0.087785 & -0.29777 & -0.231759 \\
1.03061 & 0.483638 & 0.157846 & 0 & 1 & -0.672094 & -0.235712 & 0.177668 \\
\hline
1.03061 & 0.483638 & 0.157846 & 0 & 1 & -0.672094 & -0.235712 & 0.177668 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
-0.712022 & -0.63459 & -0.0665772 & 1.06962 & 0 & 0.343573 & -0.0487644 & 0.512949 \\
-0.707945 & -0.57053 & -0.210071 & 0.622123 & 0 & 0.866424 & -0.00463586 & 0.0487644
\end{array}\right]$$

Table 6: Partitioned matrix for an explicit IRKS method of third order with property F, where $\mathbb{c} = \{1/4, 1/2, 3/4, 1\}^T$, with error-estimation coefficients of $\phi_0 = 98.267807$ and $\phi = \{329.071228, -397.606843, 201.071228, -34.267807\}^T$ [8].

$$\left[\begin{array}{cccc|cccc}
0.225 & 0 & 0 & 0 & 1 & 0.25 & -0.05 & -0.0265625 \\
0.211287 & 0.225 & 0 & 0 & 1 & 0.063713 & -0.0806435 & -0.0833663 \\
0.946338 & -0.342943 & 0.225 & 0 & 1 & -0.0783954 & 0.0947737 & 0.121956 \\
0.52149 & -0.662474 & 0.490476 & 0.225 & 1 & 0.425507 & 0.216014 & -0.103603 \\
\hline
0.52149 & -0.662474 & 0.490476 & 0.225 & 1 & 0.425507 & 0.216014 & -0.103603 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
-0.0423385 & 0.695379 & -0.784079 & 1.0116 & 0 & -0.880558 & -0.521284 & 0.774748 \\
0.077564 & 0.246379 & -0.321806 & 0.274145 & 0 & -0.276282 & -0.350743 & 0.521284
\end{array}\right]$$

Table 7: The partitioned matrix for an L-stable, implicit, IRKS method of third order that possesses property F, where $\mathbb{c} = \left\{1/4, 1/2, 3/4, 1\right\}^T$, and whose coefficients for error estimation are $\phi_0 = 43.700369$ and $\phi = \{110.801474, -70.202212, -17.198526, 20.299631\}^T$ [8].

To obtain an estimate of solution error for an IRKS method, one first constructs the difference vector

$$\Delta^{[n]} = h\left(\phi_0 y'^{[n-1]} - \sum_{i=1}^{s} \phi_i \mathbb{F}_i\right),$$

from which one can then compute a normalized estimate for the error via

$$\text{error}_n = \frac{\|\Delta^{[n]}\|}{\max\left(1, \|y^{[n-1]}\|\right)},$$

where ϕ_0 and $\phi = \{\phi_1, \phi_2, \dots, \phi_s\}^T$ are parameters of the integrator. Note that $y^{[n-1]}$ and $h y'^{[n-1]}$ are stored as the first two subarrays in the super-array $\mathbf{y}^{[n-1]}$.

3.4.2 Step-Size Control

The PI controller of Söderlind [5, 7] has been adopted to automatically control the step-size h in an optimal manner, maintaining a local truncation error of 0.0001. Specifically, if $\epsilon_n < 1.2\,\epsilon_{\max}$ then

$$\frac{h_{n+1}}{h_n} = \begin{cases} \left(\frac{\epsilon_{\max}}{\epsilon_n}\right)^{0.15} \left(\frac{\epsilon_{n-1}}{\epsilon_{\max}}\right)^{0.1} & \text{if } \epsilon_{n-1} < \frac{6}{5}\epsilon_{\max}, \\ \left(\frac{\epsilon_{\max}}{\epsilon_n}\right)^{0.25} & \text{otherwise,} \end{cases}$$

with $\epsilon_{\max}$ set at 0.0001. If $\epsilon_n > 1.2\,\epsilon_{\max}$ then

$$h_n = h_n \left(\frac{\epsilon_{\max}}{\epsilon_n}\right)^{0.33}$$

and the integration over the step is redone until an acceptable error is obtained. To avoid oscillatory behavior, step-size resizing is restricted so that $0.1 \le h_{n+1}/h_n \le 5$, i.e., anti windup.

Before the step-size h can be controlled, it is necessary to obtain a reasonable estimate for the initial step-size, viz., h_0. The software does this by first computing

$$h_0 = \frac{\|y_0\|}{\|f(x_0, y_0)\|} \quad \text{so} \quad \epsilon_m \le h_0 \le 0.001$$

where machine epsilon ϵ_m is taken to be a lower bound on step-size and 0.001 an upper bound. A forward Euler integration then gives an estimate for $y_1 = y_0 + h_0 f_0$ associated with $f_0 = f(x_0, y_0)$. This allows one to estimate $f_1 = f(x_1, y_1)$ wherein $x_1 = x_0 + h_0$. Trapezoidal integration improves the estimate for $y_1 = y_0 + \frac{1}{2}h_0(f_0 + f_1)$ that, in turn, allows the rates to be improved upon by recalculating function $f_1 = f(x_1, y_1)$, thereby supplying an improved trapezoidal estimate for y_1. With these fields in hand, one can acquire a reasonable estimate for the initial step-size via

$$h_0 = 2\left|\frac{\|y_0\| - \|y_1\|}{\|f_0\| + \|f_1\|}\right|$$

so constrained as to lie within $\epsilon_m \le h_0 \le 0.001$. This comes from the second-order approximation $\|y_1\| \approx \|y_0\| + h\|f_0\| + \frac{1}{2}h^2\|f_0'\|$ where $\|f_0'\| \approx (\|f_1\| - \|f_0\|)/h$. A factor of safety is imposed in that $h_0/10$ is actually used for the first step, from which the step-size controller quickly recovers. This helps to ensure that a reasonable first step-size is adopted, as there is no ability to establish an error estimate over this step using the startup integrator of Table 8. Whatever error one starts with carries through the solution.

3.4.3 IRKS Methods

Tables 6 & 7 provide the coefficients for explicit and implicit IRKS methods with property F that have been implemented into software. These two methods are not self-starting, so a startup method must be called upon to take the first step of integration. An admissible startup method for the integrators in Tables 6 & 7 is the integrator listed in Table 8.

Explicit integrators, like the one in Table 6, are applicable for non-stiff problems, while implicit

$$\left[\begin{array}{cccc|cccc}
\frac{1}{4} & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
\frac{1}{4} & \frac{1}{4} & 0 & 0 & 1 & 0 & 0 & 0 \\
-\frac{5}{4} & \frac{7}{4} & \frac{1}{4} & 0 & 1 & 0 & 0 & 0 \\
\frac{5}{12} & \frac{5}{12} & -\frac{1}{12} & \frac{1}{4} & 1 & 0 & 0 & 0 \\
\hline
\frac{5}{12} & \frac{5}{12} & -\frac{1}{12} & \frac{1}{4} & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\frac{29}{10} & -\frac{67}{10} & \frac{7}{10} & \frac{31}{10} & 0 & 0 & 0 & 0 \\
\frac{56}{5} & -\frac{88}{5} & \frac{8}{5} & \frac{24}{5} & 0 & 0 & 0 & 0
\end{array}\right]$$

Table 8: Partitioned matrix for a viable startup method for the IRKS methods given in Tables 6 & 7, where $c = \{1/4,\ 1/2,\ 3/4,\ 1\}^T$. Created for this author [8] by Prof. John Butcher.

integrators, like the one in Table 7, are required whenever a system of ODEs exhibits stiffness, as they often do. To implement an implicit integrator requires the use of Newton's method for finding roots, thereby making these methods more expensive at run time. Nevertheless, if a system of ODEs is found to be stiff, then their integration with an implicit method will be less costly than trying to solve it, forcibly, with an explicit method, because the step size required by an explicit method needed to maintain an accurate output in the presence of stiffness can become extremely small. Oftentimes stability cannot be achieved under these conditions.

3.4.4 Software

Module `BelMath.Irks` implements the explicit and implicit IRKS methods just discussed, both of which are thrid-order accurate. Unlike the Runge-Kutta integrator of module `RungeKutta`, which is a ***value*** object, here instances of `Explicit` and `Implicit` are ***ref*** objects, and therefore must be created with the **`new`** command.

Two procedure types are exported from this module. An instance of procedure type `F` provides the ODE that is to be integrated, i.e., $f(x, y) = \mathrm{d}y/\mathrm{d}x$. An instance of procedure type `J` provides its Jacobian, viz., $\mathbb{J}(x, y) = \partial f(x, y)/\partial y$. The first argument of both `F` and `J` is a ***real***, while their second argument is an **array** {math} * of ***real***. If an analytic Jacobian is not available, one can construct a numerical approximation for it by writing a wrapper around `NumericalJacobian`, which is a procedure exported from module `BelMath.Roots`.

Two integrators are supplied. Type `Explicit` provides an explicit method, while type `Implicit` provides an implicit method. They both export counters `functionEvaluations`, `integrationFalseStarts`, `integrationRestarts` and `integrationSteps`, with the `Implicit` integrator also providing a counter for `jacobianEvaluations`. Also exported as a read-only value is the independent variable `x`. These types have common type-bound procedures, with a slight difference occurring in method `Start`, where the ODE to be solved is passed as variable `f` of type `F` and, in the case of the implicit integrator, its Jacobian is also passed as variable `j` of type `J`. The other arguments of procedure `Start` include: the initial value `x0` for the independent variable x_0, and the initial value `y0` for the dependent variable $\mathscr{y}_0$, i.e., the initial condition. Returned is the optimal step-size `h` for the next integration step.

Wherever the system of ODEs is discontinuous, i.e., at any x where $\mathscr{f}(x_-, \mathscr{y}) \neq \mathscr{y}(x_+, \mathscr{y})$, the integrator must be restarted at x. This is done by calling procedure `Restart` which returns the optimal step-size h for the ensuing integration step.

Integration is normally carried out by procedure `Integrate` where the step-size is both an input and output variable. The user can adjust h for the next integration step, say to hit a target value of x. Normally, it is left alone and returned in the next call to `Integrate`.

The final procedure that is in common with the `Explicit` and `Implicit` integrators is method `GetSolution` that uses the embedded Taylor series in the Nordsieck scaling, thereby insuring a third-order accurate estimate to the solution over the entire range of integration. The first argument `atX` specifies where the solution is sought. The second argument `y` returns this solution, with the final argument specifying the local truncation error pertaining to the interval from which the solution was extracted. Module `BelMath.Irks` uses a `Bel.List` to manage this data base.

3.5 Convolution Integrals

Consider a convolution integral of the form

$$\int_0^x \kappa(x-y)\, \mathscr{f}(y, x)\, \mathrm{d}y, \quad x \in [0, X],$$

where the kernel κ and forcing $\mathscr{f}$ functions are known. The kernel is a real valued function, while the forcing function is taken to be an array. The K-BKZ viscoelastic models discussed in the author's book *Soft Solids* are described by Volterra integral equations that contain such convolution integrals.

The algorithm implemented by type `Integral` in the module `BelMath.ConvolutionIntegrals` provides a numerical approximation for this convolution integral based upon Algorithms 3–6 from Diethelm & Freed [9], and has been republished as App. E in the author's book [8]. The interested reader is referred to [9] for substantive details.[2]

The Diethelm-Freed integrator is more than just formulæ; it is a collection of four algorithms. The first algorithm manages assignment of an evolving set of quadrature nodes defined over a changing interval of integration. The second algorithm manages the history variables, keeping them in conformation with the nodes of quadrature. The third algorithm handles calls made to the kernel function. While the forth algorithm performs a numerical approximation of this convolution integral using a Laplace quadrature scheme with end-point correction, resulting in a method that is fifth-order accurate. This quadrature scheme allows the kernel function to be weakly singular, viz., $\kappa(0) = \infty$ is permitted.

For K-BKZ viscoelastic models, the forcing function $\boldsymbol{\mathcal{E}}$ (strain) is a function of another function $\mathbf{F}$ (the deformation gradient) that in turn depends

2. There is a typo at the end of Alg. 4 in [9]. It reads:
```
FOR i := 1 TO L_n DO
    I := I + P_n[i].
```
It should read:
```
FOR i := 1 TO L_{n-1} DO
    I := I + P_{n-1}[i].
```
This is corrected in [8, App. E].

on the independent variable of integration, which is time t in this case, and as such, can be rewritten as $\boldsymbol{\mathcal{E}}(t', t) = \widehat{\boldsymbol{\mathcal{E}}}(\mathbf{F}(0, t'), \mathbf{F}(0, t))$, or more simply, $\boldsymbol{\mathcal{E}}(t', t) = \widehat{\boldsymbol{\mathcal{E}}}(\mathbf{F}(t'), \mathbf{F}(t))$, where $\mathbf{F}(t')$ implies $\mathbf{F}(0, t')$, etc. When expressed in the format of the above convolution integral, K-BKZ models look like

$$\int_0^t \kappa(t - t')\, \mathscr{f}\big(g(t'), g(t)\big)\, \mathrm{d}t', \quad t \in [0, T],$$

where the forcing function $\mathscr{f}$ now depends upon a subordinate function g that is evaluated at two states, viz., t' and t. What this integrator stores in terms of history variables is $g(t_j)$, not $\mathscr{f}(t_i, t_j)$, where $i \leq j$, which can be done much more economically. The algorithm is structured so that as one moves further along the solution path one can delete some of the earlier $g(t_i)$, since all future samplings will only occur at points other than t_i. Actually, this was a fundamental design criterion for the algorithm developed in [9]. In the more general setting of the first expression given for the convolution integral, one has

$$\mathscr{f}(y, x) = \mathscr{f}\big(g(y), g(x)\big),$$

where the expression on the right is how the forcing function is implemented in BEL.

This numerical integrator is built upon three premises. The first is that some characteristic time $\tau > 0$ quantifies the decay behavior of kernel κ. This parameter is chosen at the beginning of the process, and is kept fixed throughout the integration. For viscoelastic models, one would likely assign τ as being the model's characteristic relaxation time.

The second premise resides with the user selecting the number of integration steps that are to be applied over the time interval $[0, \tau]$. This value is assigned to variable $H > 4$, with the step-size of integration h then being set at $h = \tau/H$, where h remains fixed over the entire interval of integration $[0, T]$. H must be greater than 4 because of the quadrature rule selected for integration.

The third premise is the existence of a quality parameter in the form of an odd cardinal number $Q \in \{3, 5, 7, 9, \ldots\}$ that is also kept fixed throughout the process. A small value for Q will lead to a fast but not so accurate implementation of the algorithm, while large choices for Q improve the accuracy, but run slower.

What Q controls is the maximum size of each subinterval of integration. Each subinterval can contain H, $2H$, ..., or QH nodes, with the subinterval closest to the upper limit containing a possible 1 to $H-1$ extra nodes, as deemed necessary to span the interval of integration $[0, T]$. The step size is fixed over each subinterval, but increases with a logarithmic gait from one subinterval to the next. Bearing in mind that the kernel κ monotonically decays, this suggests the use of a step size of h on the first subinterval, a step size of Qh on the second subinterval, a step size of Q^2h on the third subinterval, etc. This logarithmic memory concept was introduced by Ford & Simpson [10].

It is the garbage collector devised by Diethelm in [9] for managing the history variables that, when used in accordance with a Ford-Simpson logarithmic gait between quadrature nodes, requires the step size of integration h to be uniform over the whole of the interval of integration $[0, T]$.

3.5.1 Software

Module `BelMath.ConvolutionIntegrals` exports two procedure types. Instances of `Kernel` are to describe $\kappa(x-y)$, while instances of `ForcingFn` are to describe $\mathscr{f}(g(y), g(x))$. Type `Integral` exports three numbers: the next integration step will have count `n`, the total number of integration steps required is stored in `N`, while `xN` designates the value of the dependent variable x at the end of the next step `n`.

Method `SetUp` contains seven arguments. The first two arguments, `qualityPara` and `stepsPerCharTime`, are integers that hold the values for: quality parameter Q, and the number of integration steps H to be used to traverse an interval of width T. The third and fourth arguments are ***`real`***s. `xMax` is the upper limit of integration X at the final step of integration, while `charTime` is the characteristic time T associated with the specified kernel. The fifth argument, `gAt0`, is where one passes the initial value for the subordinate forcing function, viz.,

$g(0)$, to the integrator, which is an **array** {math} *** of *real***. The last two arguments contain instances of types `Kernel` and `ForcingFn`, denoted as `kernelFn` and `forcingFn`.

Method `Integrate` performs the necessary integration to advance a solution from step $n-1$ to step n. The user supplies $g(x_n)$ as argument `gAtN`, which is the second argument of f over the integration, with the first arguments coming from the memory manager of the integrator. `Integrate` returns the value of the convolution integral whose upper limit is x_n. `N` successive calls to this procedure are needed to solve a convolution integral over some range $[0, X]$.

4 History of Changes

BEL version releases wherein no change took place in BelMath are omitted. Version numbering corresponds with the version releases of BEL.

4.1 Changes to Version 4.2

An automatic time-step controller was added to the IRKS integrators. This required introducing a data base for managing the history of integration, which necessitated some changes in the interface, too.

4.2 Changes to Version 4.1

Exception handler added to `BelMath.Roots.Newton` in an effort to make it more robust. The module `BelMath.SpecialFunctions` was created, moving `Erf`, `Erfc`, `Gamma`, and `Beta` from `Bel.Math` and adding to it the function `MittagLeffler`.

4.3 Changes to Version 4.0

Modules `BelMath.Series` and `BelMath.Functions` were moved to the core library as modules `Bel.Series` and `Bel.Math`, respectively. Module `BelMath.LinearAlgebra` has been depreciated because of the ability to solve linear systems of equations directly with calls like `x := A\b`, which solves $\mathbb{A}x = b$ for vector x by using the {*math*} extension for arrays.

All of the modules in this library, including their interfaces, were altered such that numeric types `BelTypes.Numbers.Number`, `BelTypes.Arrays.Array` and `BelTypes.Matrices.Matrix` were changed to ***real***, **array**{math} *** of *real*** and **array**{math} ***,* of *real***, respectively. These simplifications were an indirect consequence of a recent compiler enhancement (not under the jurisdiction of the language definition).

4.4 Changes to Version 3.3

Modules `Irks` and `ConvolutionIntegrals` were added into the distribution, actually they were moved from a different library in the restructuring of software in this release.

Several bugs were fixed.

4.5 Changes to Version 2.3

The modules for an implicitly stable Runge-Kutta method, and for solving convolution integrals have been moved to the application library BEL-SS that accompanies the author's textbook *Soft Solids*.

Procedure `LnDeterminant` was added to module `Bel.LinearAlgebra`.

Acknowledgments

The author is indebted to Prof. John Butcher from The University of Auckland for suggestions that he made which lead to an improved implementation of the explicit Runge-Kutta methods presented in BEL for solving non-stiff ODEs, and for his patient coaching that enabled this author to implement his explicit and implicit IRKS methods as alternate means for solving ODEs. This required that he create a starter method for me, which is presented in Table 8.

References

[1] Gutknecht, J., 2009, *Zonnon Language Report*, ETH Zürich, Switzerland, 4th ed., available online at www.zonnon.ethz.ch.

[2] Gutknecht, J., Mitin, R., and Gonova, N., 2010, *Zonnon Tutorial*, ETH Zurich, Switzerland, available online at www.zonnon.ethz.ch.

[3] Marsaglia, G. and Tsang, W. W., 1984, "A fast, easily implemented method for sampling from decreasing or symmetric unimodal density functions," SIAM Journal on Scientific and Statistical Computing, **5** (2), pp. 349–359.

[4] Butcher, J. C., 2008, *Numerical Methods for Ordinary Differential Equations*, 2nd ed., Wiley, Chichester.

[5] Söderlind, G., 2002, "Automatic control and adaptive time-stepping," Numerical Algorithms, **31**, pp. 281–310.

[6] Kutta, W., 1901, “Beitrag zur näherungsweisen Integration totaler Differentialgleichungen,” Zeitschrift für angewandte Mathematik und Physik, **46**, pp. 435–453.

[7] Gustafsson, K., Lundh, M., and Söderlind, G., 1988, “A PI stepsize control for the numerical solution of ordinary differential equations,” BIT, **28** (2), pp. 270–287.

[8] Freed, A. D., 2013, *Soft Solids: A primer to the theoretical mechanics of materials*, Birkhäuser (in press), Boston.

[9] Diethelm, K. and Freed, A. D., 2006, “An efficient algorithm for the evaluation of convolution integrals,” Computers and Mathematics with Applications, **51**, pp. 51–72.

[10] Ford, N. J. and Simpson, A. C., 2001, “The numerical solution of fractional differential equations: Speed versus accuracy,” Numerical Algorithms, **26**, pp. 333–346.

A Math Modules

```
module BelMath.SpecialFunctions;
   procedure Erf (x : real) : real;
   procedure Erfc (x : real) : real;
   procedure Gamma (x : real) : real;
   procedure Beta (x, y : real) : real;
   procedure MittagLeffler (α : real; β : integer; x : real) : real;
end SpecialFunctions.
```

```
module BelMath.Interpolations;
   import
      Bel.Types as T;
   procedure Neville (xVec, yVec : T.RealVector; x : real) : real;
   procedure Hermite (xLo, xHi : real; yLo, yHi, dyLo, dyHi : T.RealVector;
                     x : real) : T.RealVector;
   procedure Bilinear (x1, x2, y1, y2, valX1Y1, valX1Y2, valX2Y1, valX2Y2,
                     atX, atY : real) : real;
end Interpolations.
```

```
module BelMath.Distributions;
   type
      Certainty = (ninety, ninetyFive, ninetySevenPointFive,
                     ninetyNine, ninetyNinePointFive);
   procedure ChiSquaredLow (percentagePoint : Certainty;
                     degreesOfFreedom : integer) : real;
   procedure ChiSquaredHigh (percentagePoint : Certainty;
                     degreesOfFreedom : integer) : real;
   procedure StudentT (percentagePoint : Certainty;
                     degreesOfFreedom : integer) : real;
   procedure F (percentagePoint : Certainty;
                     degreesOfFreedom1, degreesOfFreedom2 : integer) : real;
end Distributions.
```

```
module BelMath.RandomVariates;
   procedure Normal () : real;
   procedure Exponential () : real;
end RandomVariates.
```

```
module BelMath.Regression;
   import
      Bel.Types as T;
   procedure Linear (x, y : T.RealVector; var k0, k1, rSquared : real);
   procedure Exponential (x, y : T.RealVector; var k0, k1, rSquared : real);
   procedure PowerLaw (x, y : T.RealVector; var k0, k1, rSquared : real);
   procedure StretchedExp (x, y : T.RealVector; var k0, k1, rSquared : real);
   procedure Quadratic (x, y : T.RealVector; var k0, k1, k2, rSquared : real);
end Regression.

module BelMath.Roots;
   import
      Bel.Types as T;
   type
      F = procedure (T.RealVector) : T.RealVector;
      J = procedure (T.RealVector) : T.RealMatrix;
   procedure Newton (f : F; j : J; var x : T.RealVector);
   procedure NumericalJacobian (f : F; x : T.RealVector) : T.RealMatrix;
end Roots.

module BelMath.Derivatives;
   type
      Method = (forward, central, backward, richardson);
      Y = procedure (real) : real;
   procedure Differentiate (y : Y; x, h : real; m : Method) : real;
   procedure OptimalStepSize (atX : real) : real;
end Derivatives.

module BelMath.Integrals;
   type
      Method = (trapezoidal, simpson, threeEights, romberg, gauss);
      F = procedure (real) : real;
   procedure Integrate (f : F; a, b : real; m : Method) : real;
end Integrals.
```

```
module BelMath.RungeKutta;
   import
      Bel.Entity as Entity,
      Bel.Types as T;
   type
      Order = (second, third, fourth, fifth, sixth);
      F = procedure (real; T.RealVector) : T.RealVector;
   type {value} Integrator = object implements Entity
      var {immutable}
         functionEvaluations, integrationSteps, restarts : integer;
      procedure SetUp (f : F; h, tol : real; ord : Order);
      procedure Integrate (var x : real; var y : T.RealVector; var err : real);
   end Integrator;
end RungeKutta.

module BelMath.Irks;
   import
      Bel.Entity as Entity,
      Bel.Types as T;
   type
      F = procedure (real; T.RealVector) : T.RealVector;
      J = procedure (real; T.RealVector) : T.RealMatrix;
   type {ref} Explicit = object implements Entity
      var {immutable}
         functionEvaluations, integrationFalseStarts,
            integrationRestarts, integrationSteps : integer;
         x : real;
      procedure Start (f : F; x0 : real; y0 : T.RealVector; var h : real);
      procedure Restart (var h : real);
      procedure Integrate (var h : real);
      procedure GetSolution (atX : real; var y : T.RealVector; var error : real);
   end Explicit;
   type {ref} Implicit = object implements Entity
      var {immutable}
         functionEvaluations, integrationFalseStarts, integrationRestarts,
            integrationSteps, jacobianEvaluations : integer;
         x : real;
      procedure Start (f : F; j : J; x0 : real; y0 : T.RealVector; var h : real);
      procedure Restart (var h : real);
      procedure Integrate (var h : real);
      procedure GetSolution (atX : real; var y : T.RealVector; var error : real);
   end Implicit;
end Irks.

module BelMath.ConvolutionIntegrals;
   import
      Bel.Entity as Entity,
      Bel.Types as T;
   type
      Kernel = procedure (real) : real;
      ForcingFn = procedure (T.RealVector; T.RealVector) : T.RealVector;
   type {value} Integral = object implements Entity
      var {immutable}
         n, N : integer;
         xN : real;
      procedure SetUp (qualityPara, stepsPerCharTime : integer;
                       xMax, charTime : real; gAt0 : T.RealVector;
                       kernelFn : Kernel; forcingFn : ForcingFn);
      procedure Integrate (gAtN : T.RealVector) : T.RealVector;
   end Integral;
end ConvolutionIntegrals.
```