Users's Manual for
Basic Numerical Analysis Routines

Brent Seidel
Phoenix, AZ

July 24, 2024

Note that this is a draft version and not the final version for publication.

Contents

Chapter 1

Introduction

Back in the 1980s when I was an undergraduate, I took a numerical analysis course and quite enjoyed it. Then my first job out of college was working on a numerical analysis library for a small startup that went the way of most startups. I was recently inspired to dig out my old textbook and try implementing some of the routines. This collection includes some of those, plus others.

Note that some packages are for complex numbers and some are for real numbers. At some point, they may be combined. Most packages are generic. The packages are:

1.1 BBS.Numerical.complex

This is an object oriented collection of complex number routines. After writing this, I discovered `Ada.Numerics.Generic_Complex_Types`. So this package is deprecated in favor of the Ada package.

1.2 BBS.Numerical.derivative

This is a generic package with a real type parameter. It contains functions to compute the derivative of real valued functions with a single argument.

1.3 BBS.Numerical.functions

This is a generic package with a real type parameter. It contains some functions that are used by other packages.

1.4 BBS.Numerical.integration_real

This is a generic package with a real type parameter. It contains functions to compute integrals of real valued functions with a single argument.

1.5 BBS.Numerical.interpolation

This is a generic package with a real type parameter. It contains functions to interpolate a value between a set of data points. The functions can also extrapolate (if using a value of x outside of the range of the data points, however this is likely to be inaccurate.

1.6 BBS.Numerical.ode

This is a generic package with a real type parameter. It contains functions to solve ordinary differential equations.

1.7 BBS.Numerical.polynomial_complex

This is a generic package with a complex type parameter (from `Ada.Numerics.Generic_Complex_Types`. It contains functions for polynomials.

1.8 BBS.Numerical.polynomial_real

This is a generic package with a real type parameter. It contains functions for polynomials.

1.9 BBS.Numerical.quaternion

This is a generic package with a real type parameter. It contains functions for quaternions.

1.10 BBS.Numerical.random

This is not a generic package. It contains functions for generating pseudo-random numbers.

1.11 BBS.Numerical.regression

This is a generic package with a real type parameter. It contains functions for performing regression analysis of data.

1.12 BBS.Numerical.roots_complex

This is a generic package with a complex type parameter (from `Ada.Numerics.Generic_Complex_Types`. It contains functions for finding zeros of functions.

1.13 BBS.Numerical.roots_real

This is a generic package with a real type parameter. It contains functions for finding zeros of functions.

1.14 BBS.Numerical.statistics

This is a generic package with a real type parameter. It contains statistics related functions.

1.15 BBS.Numerical.vector

This is a generic package with a real type parameter. It contains functions for vectors.

Chapter 2

How to Obtain

This collections is currently available on GitHub at `https://github.com/BrentSeidel/Numerical`.

2.1 Dependencies

2.1.1 Ada Libraries

The following Ada libraries are used:

- `Ada.Numerics`
- `Ada.Numerics.Generic_Complex_Types`
- `Ada.Numerics.Generic_Complex_Elementary_Functions`
- `Ada.Numerics.Generic_Elementary_Functions`
- `Ada.Text_IO` (used for debugging purposes)

2.1.2 Other Libraries

This library depends on the root package `BBS` available at `https://github.com/BrentSeidel/BBS-Ada`. Internal packages used within this library are:

- `BBS.Numerical.functions`
- `BBS.Numerical.Integration_real`

Chapter 3

Usage Instructions

This is a library of routines intended to be used by some program. To use these in your program, edit your `*.gpr` file to include a line to `with` the path to `BBS_Numerical.gpr`. Then in your Ada code `with` in the package(s) you need and use the routines.

Chapter 4

API Description

4.1 BBS.Numerical.complex

This package is deprecated in favor of the Ada package `Ada.Numerics.Generic_Complex_Types`.

4.2 BBS.Numerical.derivative

This is a generic package with a real type parameter, `F`.

It defines a function type `test_func` as `access function (x : f'Base) return f'Base`.

WARNING: The calculations here may involve adding small numbers to large numbers and taking the difference of two nearly equal numbers. The world of computers is not the world of mathematics where numbers have infinite precision. If you aren't careful, you can get into a situation where $(x + h) = x$, or $f(x) = f(x \pm h)$. For example assume that the float type has 6 digits. Then, if x is $1,000,000$ and h is 1, adding x and h is a wasted operation.

The functions are:

4.2.1 Two Point Formulas

Two point formula. Use $h > 0$ for forward-difference and $h < 0$ for backward-difference. Derivative calculated at point x. While this is the basis for defining the derivative in calculus, it generally shouldn't be used.

```
function pt2(f1 : test_func; x, h : f'Base) return f'Base;
```

- `f1` - the function that you with to find the derivative of
- `x` - the point at which to calculate the derivative
- `h` - the step size

4.2.2 Three Point Formulas

Compute the derivative at x using values at x, $x+h$, and $x+2h$.

```
function pt3a(f1 : test_func; x, h : f'Base) return f'Base;
```

- `f1` - the function that you with to find the derivative of
- `x` - the point at which to calculate the derivative
- `h` - the step size

Compute the derivative at x using values at $x-h$ and $x+h$.

```
function pt3b(f1 : test_func; x, h : f'Base) return f'Base;
```

- `f1` - the function that you with to find the derivative of
- `x` - the point at which to calculate the derivative
- `h` - the step size

4.2.3 Five Point Formulas

Compute the derivative at x using values at $x-2h$, $x-h$, $x+h$, and $x+2h$.

```
function pt5a(f1 : test_func; x, h : f'Base) return f'Base;
```

- `f1` - the function that you with to find the derivative of
- `x` - the point at which to calculate the derivative
- `h` - the step size

Compute the derivative at x using values at x, $x+h$, $x+2h$, $x+3h$, and $x+4h$.

```
function pt5b(f1 : test_func; x, h : f'Base) return f'Base;
```

- `f1` - the function that you with to find the derivative of
- `x` - the point at which to calculate the derivative
- `h` - the step size

4.3 BBS.Numerical.functions

This is a generic package with a real type parameter, `F`. Its primary purpose is to contain functions that are used by other packages in this library. They may also be useful to the end user.

4.3.1 Gamma Function Related

Some of the probability functions require computing $\Gamma(\frac{n}{2})$. Since $\Gamma(\frac{1}{2}) = \sqrt{pi}$, $\Gamma(1) = 1$, and $\Gamma(a+1) = a * \Gamma(a)$, computing $\Gamma(\frac{n}{2})$, where n is a positive integer is simpler than computing the full Γ function.

Computes Γ of $2n$. Note that this will overflow for `Float` for $n > 70$.

```
function gamma2n(n : Positive) return f'Base;
```

- n - A positive integer representing twice the value for computing Γ.
- Returns the actual value.

To compute Γ of $2n$ where $n > 70$, logarithms can be used. This returns the $log_e(\Gamma(n))$.

```
function lngamma2n(n : Positive) return f'Base;
```

- n - A positive integer representing twice the value for computing Γ.
- Returns the natural log of the value.

4.3.2 Factorial Related

The factorial functions are similar to the Γ functions. Since the parameter is not divided by two in this case, overflow will occur for $n > 35$. Thus, as for Γ, there are two versions. The first returns the actual value, the second returns the natural log of the value.

```
function factorial(n : Natural) return f'Base;
```

- n - A natural integer representing.
- Returns the actual factorial value.

```
function lnfact(n : Natural) return f'Base;
```

- n - A natural integer representing.
- Returns the natural log of the factorial value.

4.3.3 Combinatorial Related

The one function computes the binomial coefficients (aka N choose K). By definition, this computes:

$$\binom{n}{k} = \frac{n!}{k!(b-k)!}$$

With some algebra, this becomes:

$$\binom{n}{k} = \prod_{i=1}^{k} \frac{n+1-i}{i}$$

which is easier to compute. The function is:

```
function nChoosek(n, k : Natural) return f'Base
    with pre => (n >= k);
```

- n - The number of items.
- k - The number of items to choose.
- Returns the binomial coefficient for n and k.
- There is a restriction that $n \geq k$

4.4 BBS.Numerical.integration_real

This is a generic package with a real type parameter, `F`.

It defines a function type `test_func` as `access function (x : f'Base) return f'Base`.

This package contains routines to perform numerical integration of a function with one parameter. Thus it computes an approximation to:

$$y = \int_a^b f(x)dx$$

4.4.1 Trapezoid Rule

This estimates the value of the integral using the composite trapezoid rule.

```
function trapezoid(test : test_func; lower, upper : f'Base; steps : Positive)
        return f'Base;
```

- *test* - The function to integrate. It must match the datatype for `test_func`.
- *lower* - The lower bounds of the integration.
- *upper* - The upper bounds of the integration.
- *steps* - The number of steps to use.

4.4.2 Simpson's Rule

This estimates the value of the integral using the composite Simpson's rule.

```
function simpson(test : test_func; lower, upper : f'Base; steps : Positive)
        return f'Base;
```

- *test* - The function to integrate. It must match the datatype for `test_func`.
- *lower* - The lower bounds of the integration.
- *upper* - The upper bounds of the integration.
- *steps* - The number of steps to use.

4.4.3 Adaptive Simpson's

This estimates the value of the integral using an adaptive version of Simpson's rule. The interval is divided into two and each side is evaluated using the composite Simpson's with 1 and 2 steps. The difference between the two evaluations is used as an error estimate. If the error is two big, the interval is divided again and the function recurses until either the recursion limit is reached or the error is within limits.

```
function adapt_simpson(test : test_func; lower, upper : f'Base;
                       tol : in out f'Base;
                       levels : Natural) return f'Base;
```

- *test* - The function to integrate. It must match the datatype for `test_func`.
- *lower* - The lower bounds of the integration.
- *upper* - The upper bounds of the integration.
- *tol* - The desired error limit. This also returns the estimated error.
- *levels* - The allowed recursion depth.

4.5 BBS.Numerical.interpolation

This is a generic package with a real type parameter, `F`. It uses Lagrange polynomials to interpolate a value given a set of points. The following data types are defined:

- `point` - A record with x and y values of type `F`.
- `points` - A currently unused array of `point`.

Interpolation is most accurate within the range of the provided data points. Outside of that range, it becomes extrapolation and may become increasingly inaccurate. While higher order methods tend to be more accurate, there is also a potential for oscillations. It is important to understand your data - if the data is linear or nearly so, using a 5th order method is probably a mistake.

```
function lag2(p0, p1 : point; x : f'Base) return f'Base;
```

- Interpolation with two points is linear interpolation. The data is approximated as $y = ax + b$.
- $p0$ - The first known point.
- $p1$ - The second known point.
- x - The location to interpolate.

```
function lag3(p0, p1, p2 : point; x : f'Base) return f'Base;
```

- Interpolation with three points is quadratic interpolation. The data is approximated as $y = ax^2 + bx + c$.

- $p0$ - The first known point.
- $p1$ - The second known point.
- $p2$ - The third known point.
- x - The location to interpolate.

```
function lag4(p0, p1, p2, p3 : point; x : f'Base) return f'Base;
```

- Interpolation with four points is cubic interpolation. The data is approximated as $y = ax^3 + bx^2 + cx + d$.
- $p0$ - The first known point.
- $p1$ - The second known point.
- $p2$ - The third known point.
- $p3$ - The fourth known point.
- x - The location to interpolate.

```
function lag5(p0, p1, p2, p3, p4 : point; x : f'Base) return f'Base;
```

- Interpolation with five points is quartic interpolation. The data is approximated as $y = ax^4 + bx^3 + cx^2 + dx + e$.
- $p0$ - The first known point.
- $p1$ - The second known point.
- $p2$ - The third known point.
- $p3$ - The fourth known point.
- $p4$ - The fifth known point.
- x - The location to interpolate.

4.6 BBS.Numerical.ode

This is a generic package with a real type parameter, `F`. It contains methods for numerically solving differential equations.

It defines the following datatypes:

- `test_func` as `access function (t, y : f'Base) return f'Base`
- `params` as `array (integer range <>) of f'Base`
- `sys_func` as `access function (t : f'Base; y : params) return f'Base`

- `functs` as `array (integer range <>) of sys_func`

The goal is to solve an equations:

$$\frac{dy}{dt} = f(t, y)$$

for $a \leq t \leq b$, and $y(a) = \alpha$

4.6.1 Historical

Euler's method is mostly of historic interest. It may be used if the function is expensive to evaluate and accuracy isn't very important. It most case however, there are better methods.

```
function euler(tf : test_func; start, initial, step : f'Base)
         return f'Base;
```

- *tf* - The differential equation to solve.
- *start* - The starting independent value for the step (a).
- *initial* - The initial value of the differential equation (α).
- *step* - The step size.

4.6.2 Runge-Kutta Methods

The 4th order Runge-Kutta method is the workhorse differential equation solver.

```
function rk4(tf : test_func; start, initial, step : f'Base)
         return f'Base;
```

- *tf* - The differential equation to solve.
- *start* - The starting independent value for the step (a).
- *initial* - The initial value of the differential equation (α).
- *step* - The step size.

By combining a specific 4th order with specific 5th order Runge-Kutta method, one can produce an error estimate. This is the Runge-Kutta-Fehlber method. The error estimate can be used to adjust the step size, if needed to improve accuracy.

```
function rkf(tf : test_func; start, initial, step : f'Base;
         tol : out f'Base) return f'Base;
```

- *tf* - The differential equation to solve.
- *start* - The starting independent value for the step (a).
- *initial* - The initial value of the differential equation (α).
- *step* - The step size.
- *tol* - The error estimate as an output.

4.6.3 Multistep Methods

The multistep methods depends on the values at multiple previous positions rather than just an initial value. Thus, generally another method (typically a Runge-Kutta) method is used to generate these values.

Fourth order Adams-Bashforth method. This is a four step explicit method.

```
function ab4(tf : test_func; start, step : f'Base; i0, i1, i2, i3 : f'Base)
         return f'Base;
```

- *tf* - The differential equation to solve.
- *start* - The starting independent value for the step (a).
- *step* - The step size.
- $i0$ - The initial value of the differential equation (α).
- $i1$ - The initial value of the differential equation (α_1).
- $i2$ - The initial value of the differential equation (α_2).
- $i3$ - The initial value of the differential equation (α_3).

Fourth order Adams-Moulton method. This is a three step implicit method.

```
function am4(tf : test_func; start, step : f'Base; i0, i1, i2, i3 : f'Base)
         return f'Base;
```

- *tf* - The differential equation to solve.
- *start* - The starting independent value for the step (a).
- *step* - The step size.
- $i0$ - The initial value of the differential equation (α).
- $i1$ - The initial value of the differential equation (α_1).
- $i2$ - The initial value of the differential equation (α_2).
- $i3$ - The initial value of the differential equation (α_3).

Fourth order Adams-Bashforth/Adams-Moutlton method. This uses the Adams-Bashforth method to predict the final value and then uses the Adams-Moulton method to refine that value as a corrector method.

```
function abam4(tf : test_func; start, step : f'Base; i0, i1, i2, i3 : f'Base)
         return f'Base;
```

- *tf* - The differential equation to solve.
- *start* - The starting independent value for the step (a).

- *step* - The step size.
- $i0$ - The initial value of the differential equation (α).
- $i1$ - The initial value of the differential equation (α_1).
- $i2$ - The initial value of the differential equation (α_2).
- $i3$ - The initial value of the differential equation (α_3).

4.6.4 Systems of Differential Equations

The previous methods solve a single first order differential equation. If you have a higher order equation or a system of coupled equations, then you need a method that works on systems of differential equations. An nth order system of first order equations has the form:

$$\begin{aligned}\frac{dy_1}{dt} &= f_1(t, y_1, ..., y_n)\\ \frac{dy_2}{dt} &= f_2(t, y_1, ..., y_n)\\ &\vdots\\ \frac{dy_n}{dt} &= f_n(t, y_1, ..., y_n)\end{aligned}$$

with $a \leq t \leq b$, and initial conditions:

$$\begin{aligned}y_1(a) &= \alpha_1\\ y_2(a) &= \alpha_2\\ &\vdots\\ y_n(a) &= \alpha_n\end{aligned}$$

To solve higher order differential equations, they need to be transformed into a system of first order equations.

```
function rk4s(tf : functs; start : f'Base; initial : params; step : f'Base)
         return params
      with pre => (tf'First = initial'First) and (tf'Last = initial'Last);
```

- tf - An array containing the system of differential equations to solve ($f_1 ... f_n$).
- *start* - The starting independent value for the step (a).
- *initial* - An array containing the initial values for the system of differential equation ($\alpha_1 ... \alpha_n$).
- *step* - The step size.
- There is a restriction that the array bounds for tf and *initial* must be the same.
- The returned array has the same bounds as tf and *initial*.

4.7 BBS.Numerical.polynomial_complex

This is a generic package with a complex package parameter, `cmplx`. This parameter has datatypes `cmplx.Complex` and `cmplx.Real`.

The package defines the datatype `poly` as `array (Natural range <>) of cmplx.Complex`, with a constraint that `poly'First` is equal to 0. Polynomials are implemented as an array of coefficients where the index in the array represents the exponent of the independent variable. The index range starts from 0 and extends to the order of the polynomial. This package contains operations on polynomials with complex number coefficients and independent variable.

4.7.1 Basic Operations

The basic binary operations on polynomials, addition ('+'), subtraction ('-'), and multiplication ('*') are implemented and assigned to the operators. Unary negation ('-') is implemented and assigned to the '-' operator. Multiplication of a polynomial by a scalar is implemented and assigned to the '*' operator for both scalar first and scalar last.

Division is based on the `polydiv()` implementation of synthetic division in [3]. It divides two polynomials and returns a remainder and quotient.

$$\frac{u}{v} = q, r$$

```
procedure divide(u, v : poly; q : out poly; r : out poly)
      with pre => (q'Last >= (u'Last - v'Last)) and
                  (r'Last >= (v'Last - 1)) and
                  (u'Last >= v'Last);
```

- u - The numerator for division.
- v - The denominator for division.
- q - The quotient result of the division.
- r - The remainder from the division.
- This is a procedure, thus nothing is returned.

4.7.2 Other Operations

Evaluate a polynomial at a specific value of the independent variable.

```
function evaluate(p : poly; x : cmplx.Complex) return cmplx.Complex;
```

- p - The polynomial to evaluate.
- x - The independent variable.
- Returns the value of the polynomial evaluated at x.

Eliminate any leading coefficients with a zero value. For example $0x^3+x^2+3x+4$ gets trimmed to x^2+3x+4.

```
function trim(p : poly) return poly
      with post => (trim'Result'Last <= p'Last);
```

- p - The polynomial to trim.
- Returns the trimmed polynomial of order equal to or less than the original polynomial.

Determine the order of a polynomial. That is the maximum index with a non-zero value.

```
function order(p : poly) return Natural;
```

- p - The polynomial to determine the order of.
- Returns the order of the polynomial.

4.7.3 Utility

For development or debugging purposes a simple print polynomial procedure is provided.

```
procedure print(p : poly; fore, aft, exp : Natural);
```

- p - The polynomial to print.
- fore, aft, and exp - values passed in for floating point formatting.
- This is a procedure so no value is returned. A representation of the polynomial is displayed on the default output.

4.7.4 Calculus

Integrals and derivatives of polynomials are fairly easy to generate. So, functions are provided to take a polynomial and return the integral or the derivative.

$$\int ax^n dx = a\frac{x^{n+1}}{n+1} + C$$

The resulting polynomial for integration has order one greater than the original polynomial.

```
function integrate(p : poly; c : cmplx.Complex) return poly
      with post => (integrate'Result'Last = (p'Last + 1));
```

```
integrate(p :  poly; c :  cmplx.Complex) return poly
```

- p - The polynomial to integrate.
- c - The constant term.
- Returns the integral of p.

$$\frac{d}{dx}x^n = nx^{n-1}$$

The resulting polynomial for differentiation has order one less than the original polynomial.

```
function derivative(p : poly) return poly
      with post => (((derivative'Result'Last = (p'Last - 1)) and (p'Last > 0)) or
                    ((derivative'Result'Last = 0) and (p'Last = 0)));
```

- p - The polynomial to differentiate.
- Returns the derivative of the original polynomial.

4.8 BBS.Numerical.polynomial_real

This is a generic package with a real type parameter, `F`.

The package defines the datatype `poly` as `array (Natural range <>) of f'Base`, with a constraint that `poly'First` is equal to 0. Polynomials are implemented as an array of coefficients where the index in the array represents the exponent of the independent variable. The index range starts from 0 and extends to the order of the polynomial. This package contains operations on polynomials with real number coefficients and independent variable.

4.8.1 Basic Operations

The basic binary operations on polynomials, addition ('+'), subtraction ('-'), and multiplication ('*') are implemented and assigned to the operators. Unary negation ('-') is implemented and assigned to the '-' operator. Multiplication of a polynomial by a scalar is implemented and assigned to the '*' operator for both scalar first and scalar last.

Division is based on the `poludiv()` implementation of synthetic division in [3]. It divides two polynomials and returns a remainder and quotient.

$$\frac{u}{v} = q, r$$

```
procedure divide(u, v : poly; q : out poly; r : out poly)
      with pre => (q'Last >= (u'Last - v'Last)) and
                  (r'Last >= (v'Last - 1)) and
                  (u'Last >= v'Last);
```

- u - The numerator for division.
- v - The denominator for division.
- q - The quotient result of the division.
- r - The remainder from the division.
- This is a procedure, thus nothing is returned.

4.8.2 Other Operations

Evaluate a polynomial at a specific value of the independent variable.

```
function evaluate(p : poly; x : f'Base) return f'Base;
```

- p - The polynomial to evaluate.
- x - The independent variable.
- Returns the value of the polynomial evaluated at x.

Eliminate any leading coefficients with a zero value. For example $0x^3+x^2+3x+4$ gets trimmed to x^2+3x+4.

```
function trim(p : poly) return poly
      with post => (trim'Result'Last <= p'Last);
```

- p - The polynomial to trim.
- Returns the trimmed polynomial of order equal to or less than the original polynomial.

Determine the order of a polynomial. That is the maximum index with a non-zero value.

```
function order(p : poly) return Natural;
```

- p - The polynomial to determine the order of.
- Returns the order of the polynomial.

4.8.3 Utility

For development or debugging purposes a simple print polynomial procedure is provided.

```
procedure print(p : poly; fore, aft, exp : Natural);
```

- p - The polynomial to print.
- fore, aft, and exp - values passed in for floating point formatting.
- This is a procedure so no value is returned. A representation of the polynomial is displayed on the default output.

4.8.4 Calculus

Integrals and derivatives of polynomials are fairly easy to generate. So, functions are provided to take a polynomial and return the integral or the derivative.

$$\int ax^n dx = a\frac{x^{n+1}}{n+1} + C$$

The resulting polynomial for integration has order one greater than the original polynomial.

```
function integrate(p : poly; c : f'Base) return poly
      with post => (integrate'Result'Last = (p'Last + 1));
```

- p - The polynomial to integrate.
- c - The constant term.
- Returns the integral of p.

$$\frac{d}{dx}x^n = nx^{n-1}$$

The resulting polynomial for differentiation has order one less than the original polynomial.

```
function derivative(p : poly) return poly
      with post => (((derivative'Result'Last = (p'Last - 1)) and (p'Last > 0)) or
                    ((derivative'Result'Last = 0) and (p'Last = 0)));
```

- p - The polynomial to differentiate.
- Returns the derivative of the original polynomial.

4.9 BBS.Numerical.quaternion

This is a generic package with a real type parameter, `F`.

Quaternions are an extension of complex numbers into three dimensions. A quaternion q is defined as follows:

$$q = a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$$

Where

$$\mathbf{i}^2 + \mathbf{j}^2 + \mathbf{k}^2 = \mathbf{ijk} = -1$$

This package defines the datatype `quaternion` as a tagged record with fields r, i, j, and k, each of type `F'Base`. This means that it is treated as an object and some of the functions can be called in object oriented fashion.

4.9.1 Basic Operations

The basic binary operations on polynomials, addition ('+'), subtraction ('-'), multiplication ('*'), and division ('/') are implemented and assigned to the operators. Multiplication of a polynomial by a scalar is implemented and assigned to the '*' operator for both scalar first and scalar last. Note that with quaternions, multiplication is not necessarily commutative. That is, in general $q_1 q_2 \neq q_2 q_1$.

4.9.2 Other Operations

Compute the magnitude of a quaternion (similar to absolute value) using $|q| = \sqrt{a^2 + b^2 + c^2 + d^2}$. Note that overflow is possible if any of a, b, c, d are greater than the square root of the maximum value of `F'Base`.

```
function magnitude(self : in quaternion) return f'Base;
```

- *self* - The quaternion to have its magnitude returned.
- Returns the magnitude of *self*.

Scale a quaternion to have a magnitude of 1 (a unit vector). Since the magnitude is computed to be used as a scale factor, the same comments about overflow apply here.

```
function normalize(self : in quaternion) return quaternion;
```

- *self* - The quaternion to be scaled.
- Returns a unit quaternion with the same direction as *self*.

4.10 BBS.Numerical.random

This is not a generic package.

The Ada standard doesn't specify the random number generator to be used. This package provides two different methods of generating a sequence of pseudo-random numbers. This package defines an abstract tagged type, `RNG`, which is used to specify the operations that all random number generators need to provide. It also defines the type `double` as `digits 12`. This type is used to convert the `uint32` output of the random number generator into a real number in the range 0.0 to 1.0.

I wrote implementations of these in Java around 2010. The following routines are based on the Java routines translated into Ada.

4.10.1 Linear Congruent Generator

The linear congruent generator is based on algorithms in [3]. It provides a datatype `LCG` which defines an object containing the parameters and state for the generator. This allows for multiple random number streams with the same or different parameters.

The linear congruent generator is based on the formula:

$$R_{n+1} = (aR_n + c) \mod modulus$$

where R_0 is the starting point or seed of the generator. This generator is fairly fast and has minimal state information. With a proper choice of parameters, it can produce decent results. With an improper choice of parameters, the results are not very random.

Initialize the `LCG` object to some default values.

```
overriding
procedure init(self : in out LCG);
```

- *self* - The LCG object to initialize.

Initialize the `LCG` object with specified parameters. Since the linear-congruent generator is commonly used, parameters can be chosen to emulate the sequence from many other systems.

```
procedure init(self : in out LCG; seed, modulus, a, c : uint32);
```

- *self* - The LCG object to initialize.
- *seed* - The initial value for the generator.
- *modulus* - The modulus for the generator.
- *a* - The *a* coefficient in the generator formula.
- *c* - The *c* coefficient in the generator formula.

The initial, (*seed*) value can be set in a generator without changing the other values.

```
procedure setSeed(self : in out LCG; seed : uint32);
```

- *self* - The LCG object to adjust.
- *seed* - The new seed value.

Return the next random number in raw format as a `uint32`.

```
overriding
function getNext(self : in out LCG) return uint32;
```

- *self* - The LCG object to get the next random number from.
- Returns the next `uint32` in the sequence.

Return the next random number as a floating point in the range 0.0 to 1.0. This can then be cast to the desired real number type.

```
overriding
function getNextF(self : in out LCG) return double;
```

- *self* - The LCG object to get the next random number from.
- Returns the next real number in the sequence.

4.10.2 Mersenne Twister Generator

The mersenne twister generator is based on algorithms in [2]. It provides a datatype `MT` which defines an object containing the parameters and state for the generator. This allows for multiple random number streams with the same or different parameters. While this generator is fairly fast, it requires about 624 `uint32` values of state information, thus considerably more memory than `LCG`.

Initialize the `MT` object to some default values.

```
overriding
procedure init(self : in out MT);
```

- *self* - The MT object to initialize.

Initialize the `MT` object with specified seed.

```
procedure init(self : in out MT; seed : uint32);
```

- *self* - The LCG object to initialize.
- *seed* - The initial value for the generator.

Return the next random number in raw format as a `uint32`.

```
overriding
function getNext(self : in out MT) return uint32;
```

- *self* - The MT object to get the next random number from.
- Returns the next `uint32` in the sequence.

Return the next random number as a floating point in the range 0.0 to 1.0. This can then be cast to the desired real number type.

```
overriding
function getNextF(self : in out MT) return double;
```

- *self* - The MT object to get the next random number from.
- Returns the next real number in the sequence.

4.11 BBS.Numerical.regression

This is a generic package with a real type parameter, `F`. It defines the following datatypes:

- `point` which is a record of two fields: x and y, both of which are `F'Base`.
- `data_array` as an array of `point`.
- `simple_linreg_result` is a record containing the results of simple linear regression. The fields are:

- a - The Y intercept
- b - The slope
- SSe - The sum of square errors
- var - The sum of square errors normalized by the degrees of freedom $(n-2)$.
- $Bvar$ - The variance of the slope.
- cor - The correlation coefficient (R) (note that correlation does not imply causation).

4.11.1 Simple Linear Regression

Given a set of (x, y) points where x is assumed to have no error, and the errors on y are assumed to have a mean of zero, normal distribution, and constant variance, find the line, $y = ax + b + \epsilon$, where ϵ is the error, that best fits the data.

```
function simple_linear(d : data_array) return simple_linreg_result;
```

- d - An array of the data points to analyze.
- Returns a record of type `simple_linreg_result` containing the parameters for the line as well as estimates of how good the fit is.

4.12 BBS.Numerical.roots_complex

This is a generic package with a complex package parameter, `cmplx`. This has datatypes `cmplx.Complex` and `cmplx.Real`. It contains methods for finding roots of equations with complex variables.

It defines the following datatypes:

- `test_func` as `access function (x : cmplx.complex) return cmplx.complex`
- `errors` as an enumeration of `(none, bad_args, no_solution)`

Solve for (possibly) complex roots using Mueller's method. Mueller's method starts with three initial approximations to the root and solves the parabola through these three points to produce the next approximation.

```
function mueller(test : test_func; x0, x2 : in out cmplx.complex;
        limit : in out Positive; err : out errors) return cmplx.complex;
```

- *test* - The function to find a root of.
- $x0$ and $x2$ - Starting points for the root finding
- *limit* - The number of iterations.
- *err* - An error code.
- Returns an estimate of a root of the function.

4.13 BBS.Numerical.roots_real

This is a generic package with a real type parameter, `F`. It contains methods for finding roots of equations with complex variables.

It defines the following datatypes:

- `test_func` as `access function (x : f'Base) return f'Base`
- `errors` as an enumeration of `(none, bad_args, no_solution)`

4.13.1 Linear Methods

If there is an odd number of roots between the lower and upper limits, the bisection algorithm will always converge. Each iteration simply halves the interval between the limits, keeping a root between them. The result is the midpoint of the final interval after the specified number of iterations.

The lower and upper values are updated during the iterations to provide an interval containing the root.

```
function bisection(test : test_func; lower, upper : in out f'Base;
         limit : Positive; err : out errors) return f'Base;
```

- *test* - The function to find a root of.
- *lower* - The lower bound of the interval for the root.
- *upper* - The upper bound of the interval for the root.
- *limit* - The number of iterations.
- *err* - An error code.
- Returns an estimate of a root of the function.

If there is an odd number of roots between the lower and upper limits, the secant method will always converge. Each stage of the iteration identifies a point where a line between the lower and upper values crosses the axis. This point is used instead of the midpoint in the bisection algorithm.

Depending on the function, this can converge to a root much faster than the bisection algorithm. On the other had, it can also converge much slower.

The lower and upper values are updated during the iterations to provide an interval containing the root. If the root is exact, the lower and upper values are equal to the returned value.

This method will fail if during the process, the function values at the upper and lower bounds are ever equal. This will cause a divide by zero error.

```
function seacant(test : test_func; lower, upper : in out f'Base;
         limit : Positive; err : out errors) return f'Base;
```

- *test* - The function to find a root of.
- *lower* - The lower bound of the interval for the root.
- *upper* - The upper bound of the interval for the root.

- *limit* - The number of iterations.
- *err* - An error code.
- Returns an estimate of a root of the function.

4.13.2 Mueller's Method

Mueller's method uses three points to model a quadratic curve and uses that to find a candidate root. Unlike the bisection method, Mueller's method does not require a root to be located within the three points. This can potentially be used to find complex roots, however this implementation does not.

This method will fail if the function value at the three points is equal.

In this implementation, the user provides two points and the third is generated as the average of these two points. This keeps the call the same as the bisection and secant functions.

Note that for this algorithm, the x0 and x2 values are not necessarily meaningful as upper and lower bounds for the root, except that they are both set equal to the return value if the root is exact.

Note that the success may be sensitive to the choice of x0 and x2. If you know that a root exists and get a no_solution error, try different values.

```
function mueller(test : test_func; x0, x2 : in out f'Base;
         limit : in out Positive; err : out errors) return f'Base;
```

- *test* - The function to find a root of.
- $x0$ and $x2$ - Starting points for the root finding
- *limit* - The number of iterations.
- *err* - An error code.
- Returns an estimate of a root of the function.

4.14 BBS.Numerical.statistics

This is a generic package with a real type parameter, `F`. It contains a number of statistics related routines. It defines the datatype `data_array` as `array (Integer range <>) of F'Base`.

4.14.1 Data Statistics

These routines compute statistics on the values in a `data_array`.

Compute the mean of a set of values. For n values, computes:

$$m = \frac{1}{n}\sum_{i=1}^{n} d_i$$

```
function mean(d : data_array) return F'Base;
```

- *d* - The array containing the data.
- Returns the mean value of the data.

Compute the limits of a set of data.

```
procedure limits(d : data_array; min : out F'Base; max : out F'Base);
```

- *d* - The array containing the data.
- *min* - Output as the minimum value in the array.
- *max* - Output as the maximum value in the array.
- This is a procedure and does not return a value.

Compute the variance of a set of data. Since computing the variance also requires computing the mean, this procedure returns both values.

```
procedure variance(d : data_array; var : out F'Base; mean : out F'Base);
```

- *d* - The array containing the data.
- *var* - Output as the variance of the data in the array.
- *mean* - Output as the mean of the data in the array.
- This is a procedure and does not return a value.

4.14.2 Probability Distributions

Note on naming conventions for probability distributions. For continuous distribution functions, there are generally two functions. The one with the "_pdf" suffix is the probability density function. The one with the "_cdf" suffix is the cumulative distribution function. For discrete functions, the "_pmf" suffix is for the probability mass function.

Normal Distribution

Compute the standard normal distribution where the PDF is given by $x = e^{-p^2}$.

```
function normal_pdf(p : F'Base) return F'Base;
```

- *p* - The value.
- Returns the probability density at point *p*.

Compute the normal distribution with a specified mean and standard deviation (σ).

```
function normal_pdf(p, mean, sigma : F'Base) return F'Base;
```

- *p* - The value.

- *mean* - The mean of the distribution.
- *sigma* - The standard deviation of the distribution.
- Returns the probability density at point p.

Compute the normal cumulative distribution for the standard normal function. This is computed by integrating the standard normal function from a to b.

$$p = \int_a^b e^{-x^2} dx$$

```
function normal_cdf(a, b : F'Base; steps : Positive) return F'Base;
```

- a - Starting point for integration.
- b - Ending point for integration.
- *steps* - Number of steps for Simpson's rule integration.
- Returns the cumulative probability between a and b.

χ^2 Distribution

Compute the χ^2 distribution where the PDF is given by:

$$p = \frac{1}{2^{\frac{k}{2}}\Gamma(\frac{k}{2})} x^{\frac{k}{2}-1} e^{-\frac{x}{2}}$$

```
function chi2_pdf(x : f'Base; k : Positive) return f'Base;
```

- x - The value.
- k - The number of degrees of freedom. Must be greater than zero.
- Returns the probability density at point x.

Compute the cumulative χ^2 between a and b.

```
function chi2_cdf(a, b : F'Base; k, steps : Positive) return F'Base;
```

- a - Starting point for integration.
- b - Ending point for integration.
- *steps* - Recursion depth for adaptive Simpson's rule integration.
- Returns the cumulative probability between a and b.

Student's T Distribution

Compute the student's t distribution where the PDF is given by:

$$p = \frac{\Gamma(\frac{\nu+1}{2})}{\sqrt{\pi\nu}\Gamma(\frac{\nu}{2})}(1+\frac{t^2}{\nu})^{-\frac{\nu+1}{2}}$$

```
function studentT_pdf(t : f'Base; nu : Positive) return f'Base;
```

- t - The value.
- nu - Shape parameter (ν).
- Returns the probability density at point t.

Compute the cumulative student t between a and b.

```
function studentT_cdf(a, b : F'Base; nu, steps : Positive) return F'Base;
```

- a - Starting point for integration.
- b - Ending point for integration.
- *steps* - Recursion depth for adaptive Simpson's rule integration.
- Returns the cumulative probability between a and b.

Exponential Distribution

Compute the exponential distribution where the PDF is given by $\lambda e^{-\lambda x}$.

```
function exp_pdf(x, lambda : f'Base) return f'Base;
```

- x - The value.
- *lambda* - The shape parameter (λ)
- Returns the probability at point x.

Compute the cumulative exponential between 0 and x. This is given by $1 - e^{-\lambda x}$

```
function exp_cdf(x, lambda : f'Base) return f'Base;
```

- x - The value.
- *lambda* - The shape parameter (λ)
- Returns the cumulative probability from 0 to x.

Poisson Distribution

Since the Poisson Distribution is a discrete distribution rather than a cumulative distribution, we compute a probability mass function. This is given by:

$$p(k) = \frac{\lambda^k e^{-\lambda}}{k!}$$

```
function poisson_pmf(k : Natural; lambda : Positive) return f'Base;
```

- k - The value.
- *lambda* - The shape parameter (λ)
- Returns the probability of k.

4.14.3 Statistical Tests

Student's T Tests

Use the one sample t test to compare the mean of a sample to a specified mean. Given the sample mean, $\bar{x}$, the sample standard deviation, s, and the sample size, n, compute the student's t statistic for the difference between $\bar{x}$ and the specified mean, μ_0.

```
function studentT_one(x_bar, mu0, s, n : f'Base) return f'Base;
```

- x_bar - The sample mean ($\bar{x}$).
- $mu0$ - The specified mean (μ_0).
- s - The sample standard deviation (s).
- n - The number of samples (n).
- Returns the t statistic for the difference between the $\bar{x}$ and μ_0.

4.15 BBS.Numerical.vector

This is a generic package with a real type parameter, `F`. It contains some vector related routines. It defines the datatype `vect` as `array (integer range <>) of f'Base`

4.15.1 Basic Operations

Addition ('+') and subtraction ('-') of vectors is defined and assigned to the operators. These operations require both vectors to have the same index ranges and return a vector with the same index range.

Multiplication ('*') of vectors is defined and assigned to the operator. It requires both vectors to have the same index and returns the scaler dot product of the vectors.

Multiplication ('*') of a vector by a scaler is defined and assigned to the operator for both scaler first and scaler last. It returns a vector with the same index range as the source vector.

4.15.2 Other Operations

Compute the magnitude of a vector by taking the square root of the sum of the squares of the elements. Note that overflow is possible if any of the elements are greater than the square root of the maximum value of `F'Base`.

```
function magnitude(self : in vect) return f'Base;
```

- *self* - The vector to take the magnitude of.
- Returns the magnitude of the vector.

Scale the elements in a vector so that the vector has a magnitude of one. The same overflow warning for the `magnitude` function also apply here.

```
function normalize(self : in vect) return vect;
```

- *self* - The vector to normalize.
- Returns the normalized vector.

Bibliography

[1] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. Prindle, Weber & Schmidt, 3rd edition, 1985.

[2] Makoto Matsumoto and Takuji Mishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, January 1998.

[3] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1992.