Basic Numerical Analysis Routines

Brent Seidel
Phoenix, AZ

July 14, 2024

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.

```
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 $x + 2h$.

```
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 $x + h$.
`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 $x + 2h$.
`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 $x + 4h$.
`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 $n > 70$.
`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 $log_e(\Gamma(n))$.
`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.
`factorial(n : Natural) return f'Base`

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

`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:
`nChoosek(n, k : Natural) return f'Base`

- n - The number of items.
- k - The number of items to choose.
- 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.

```
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.

```
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.

```
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.

`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.

`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.

`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.

`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.

`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.

```
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.

```
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.

```
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.

```
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.

```
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`

- 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 has datatypes `cmplx.Complex` and `cmplx.Real`.

4.8 BBS.Numerical.polynomial_real

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

4.9 BBS.Numerical.quaternion

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

4.10 BBS.Numerical.random

This is not a generic package.

4.11 BBS.Numerical.regression

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

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`.

4.13 BBS.Numerical.roots_real

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

4.14 BBS.Numerical.statistics

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

4.15 BBS.Numerical.vector

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