BEL
A .NET Computational Package Written in Zonnon
A 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`

January 16, 2010

BEL Logo

Art work was done by Blake Johnson, professor of Graphic Design at SVSU, and his students Alisha Toyzan and Brett Walsh.

Abstract

This document describes the BEL software library written in Zonnon 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 through BEL's minimalist set of numeric types. BEL's interface is cataloged in the appendices.

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].

The Microsoft .NET Framework is free to download from their website `http://msdn.microsoft.com`. Versions are available for computers that run their more recent Windows operating systems. The Mono Framework is an open source project sponsored by Novell that implements the .NET standard. It too is free to download from that project's website `http://www.mono-project.com`. Mono is available for computers that run Linux, OS-X (the Apple Mac), freeBSD and Microsoft Windows. The author compiled his Zonnon files under Mono running on Linux.

Zonnon is the most recent language in the Pascal / Modula / Oberon family of programming languages developed by Profs. Niklaus Wirth and Jürg Gutknecht from the Computer Systems Institute at ETH Zurich (Eidgenössische Technische Hochschule, Zürich, 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. The Zonnon compiler, documentation, example programs, and even BEL itself, are free and available for download from their website `http://www.zonnon.ethz.ch`.

To date, no textbook has been written in English (there are two in Russian) that describes how to program in Zonnon. In this regard, a person new to Zonnon may find it instructive to study the source code of BEL. There are, however, several important features of the Zonnon language that BEL does not use, e.g., activities and protocols.

1.1 Version

This document corresponds to software version 1.2 of BEL, December 2009, which is the first version to come with documentation.

1.2 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, please forward them to me. All feedback is welcomed.

2 Design Philosophy

The motivation behind my writing of BEL was a need to economize on the number of numeric types available in order to avoid an explosion in the number of extended higher-level types being created; in my case, scalars, vectors and tensors, which are numbers, arrays and matrices with physical units like mass, length and time (see Sec. 4.2). An examination of the sheer number of raw numeric types that are available and prebuilt into .NET, and recognizing that I would have to handle these in just a simple straightforward extension of going from mathematical fields to physical fields, led me to apply Wirth's design paradigm: **Seek simplicity!** My endeavor to seek simplicity in this jungle of numeric types motivated me to write this software.

My second conscious design decision was to use *value* types as wrappers around *ref* (or pointer) types whenever possible. There are applications where this is not desirable, e.g., nodes in data structures lend themselves nicely to dynamic types so that linkages can be created, broken and recreated as needed, without the need to move their data in memory. For most computational needs, however, static types seem to work best, so as to avoid unwanted side effects. Yet, data structures like arrays and matrices are most useful when their structures are dynamically created. BEL tries to provide the best of both worlds by appearing to the user as a collection of static types, while hiding and managing their dynamic data structures internally. The nuisance of creating dynamic types has therefore been removed from the user and relegated to the BEL system.

3 The Core

The core set of modules that define BEL include: input/output (IO), data structures, numeric types, and a collection of mathematical functions, operators and procedures.

The definition modules for various types exported by BEL are given in App. B. The software interfaces to the core modules of BEL are provided in App. C–F.

3.1 Input/Output

There are a great number of .NET modules that deal with file IO. I have economized on three types: log, data and text files. All files are written into the subdirectory `<executable>/iofiles` right below where your program's executable file resides. BEL will automatically create this directory if it does not exist.

3.1.1 Log File

Zonnon does not have a construct for throwing a runtime exception, so I created a logging capability where messages can be written that should prove useful to a programmer who is trying to debug their code. This file is titled: `<executable>/iofiles/logFile.txt`. If a log file already exists, the old file will be copied to `lastLogFile.txt`, and a new file `logFile.txt` will then be opened. If there was an existing old log file, it will be deleted before your prior log file is renamed to `lastLogFile.txt`.

This log file is automatically created when its module is first loaded into memory. There are three procedures that a programmer may use: `Message`, `WarningMessage` and `ErrorMessage`. A `Message` writes a string into the log file, while a `WarningMessage` or an `ErrorMessage` write predefined messages into the file, and a string that locates where the message was issued. These predefined messages reside in the module `Bel.IO.Log`. The difference between warning and error messages is that execution continues whenever a warning is issued, but it terminates whenever an error message is issued.

The last line in each executable file that you write should be the command that closes the log file, i.e., `Bel.IO.Log.Close`.

3.1.2 Data Files

Data files are used to write/read data to/from a file in binary format. A writer created with `Bel.IO.DataFiles.OpenWriter` takes a file name as a string and returns a .NET framework writer of type `System.IO.BinaryWriter`. The file name that you supply will be stripped of any path information and/or extension you gave it. It will be given the file extension `.dat` and placed under the directory `<executable>/iofiles` along side the log file. When you are done writing to the file, close it by calling the procedure `CloseWriter`. When it is time to read in data from a file, the corresponding `OpenReader` and `CloseReader` procedures can be called.

The .NET binary reader and writer that are supplied by these procedures are, in fact, the arguments to be used by the `Load` and `Store` methods that belong to instances of a `Bel.Object`.

3.1.3 Text Files

Text files mirror data files in all aspects except that these files are human-readable text files instead of machine-readable binary files. They are encoded in UTF8 unicode format, and given a `.txt` extension.

3.2 Objects

A `Bel.Object` is a Zonnon definition for persistent *value* objects. Any implementation of a `Bel.Object` is required to export five methods. `Initialize` is to be used to prepare an object for use. `Nullify` is to be used to set all dynamically allocated internal variables to `nil` so that they can be collected by the framework's garbage collector. `Clone` is to return a static instance of an object, i.e., all dynamically allocated internal variables are to be left as `nil`. `Load` is to be called whenever an object is to be read from a binary file. And `Store` is to be called whenever an object is to be written to a binary file.

3.3 Data Structures

The data structures implemented here came from Niklaus Wirth's classic text on the subject [2].[1] In fact, the software examples in this book have been ported over to Zonnon, and can be downloaded from the Zonnon website. I have used the engines of these algorithms for handling my data structures, as Wirth published them. Some extensions to his algorithms exist in their BEL implementations. Predominantly, each datum held within a data structure is taken to be an instance of a `Bel.Object`, making these structures very flexible and persistent.

The programmer needs to know *a priori* what data type is held within a data structure that has been stored to file before it can be read back in. In order to read data in from a binary file, a programmer must program the following two steps:

i) Specify what data type is to be read in from a file by first calling the method: `<a new data structure>.Configure(<a clone of the stored data type>)`.

ii) Read in its data using a reader attached to this file by calling the method: `<a new data structure>.Load(<reader to a binary file>)`.

Consequently, if a BEL data structure is to be persistent, then it must hold only one type or kind of data.

By supplying a clone for the type of data being held by a data structure, you enable that data structure to load itself from a file. This is not complete meta-programming, where a data structure need not know what type of data it will read-in in advance. Such a capability, although it would be nice, is seldom necessary, and is currently not supported.

It is not necessary to create or pass a node for any of the BEL data structures. All nodes are handled internally by the data structures themselves. All you have to do, as a programmer, is insert the data and, if necessary, supply its associated key. `Bel.DATA.Keys.Key` are used by lists and trees to sort the data within their structures. Each datum entry within such a structure must associate with a unique key.

3.3.1 Queues and Stacks

A *queue* is a first-in first-out (FIFO) data buffer. A *stack* is a first-in last-out (FILO) data buffer. Method `Push` is used to place a datum onto a queue or stack, while method `Pop` releases the next available datum from the structure. Method `Length` returns the number of data in the buffer.

1. When the copyright expired on Wirth's textbook "*Algorithms + Data Structures = Programs*", he rewrote the manuscript for Oberon and made it publically available, placing it on `http://www.oberon.ethz.ch`.

3.3.2 Keys

Module Bel.DATA.Keys exports type KeyType, which establishes the internal (or hidden) type of a key. Why reveal what the actual hidden type is? Well, by creating this secondary type, the interface for type Key can be made to be independent of any base system type. This allows for future growth without altering the exported interface for type Key. Say, for example, it became necessary to use System.Decimal as the raw data type for a key. This could be done easily by changing just type KeyType. The exported interface to type Key need not change. Now, certainly, the internal workings of this type would have to change, but the contract exported to the rest of the world need not. This adds robustness.

A Bel.DATA.Keys.Key is an implementation of Bel.Object. Additional methods include Get and Set, which are called by the overloaded assignment operator ":=" as a vehicle by which an assignment can be made without calling the ':=' operator being defined. This technique is used over and over again in BEL. Methods Parse and Typeset allow a key to be read/written from/to a character string. Methods Equals, LessThan and GreaterThan are called by their appropriate overloaded operators "=", "#", "<", ">=", ">" and "<=". Overloading of the assignment operator ":=" is used for type conversion, too.

3.3.3 Lists

There are many kinds of lists that can be used as data structures in programming [2]. The list available in BEL is a double-linked list whose keys are sorted in ascending order. This means that its rider can traverse in either direction without having to restart each search at its home position.

A Bel.DATA.List is an implementation of Bel.Object. Method Length returns the number of data currently held by the list. A datum, when accompanied by a new key, can be loaded into a list through its method Insert or, when accompanied by a key that already exists in the list, that datum can be loaded into the list via method Update, taking the place of the datum previously held there. To remove a datum from a list, one sends its associated key to the list via method Delete.

A BEL list has a rider to aid in its management, although it is not explicitly exported as such. Method Home sends this rider to its start position, i.e., that datum with the smallest key. Method Next moves the rider ahead by one node, if it isn't at the last node, while method Previous moves the rider back one node, if it isn't at the first node, or home. Alternatively, method Find locates that node which has the attached key. Once the rider is placed where the programmer wants it, the node's datum can be retrieved with method GetData, and its key can be extracted with method GetKey.

3.3.4 Trees

Lists are useful whenever the number of data are small, or whenever they are to be accessed sequentially, or whenever their structures are volatile in that new nodes (a datum-key pair) are constantly being created and/or destroyed. However, when a data structure gets to be large, or when its data are accessed in a random manner, or when the overall structure is mostly static, as is often the case in data bases, then a tree becomes the preferred data structure. Like lists, there are numerous types of trees that can be used as data structures. The tree implemented in BEL is a balanced, binary, Adelson-Velskii-Landis (AVL) tree [2].

A Bel.DATA.Tree is an implementation of Bel.Object. Method Entries tells how many data are currently being held by a tree, while Height specifies how many generations or levels of branching there are in a tree. Methods Insert, Update and Delete behave like they do for lists.

A BEL tree, like its kindred list, has a rider built into it to assist in its management. Method Home moves the rider to its home position which, in the case of a tree, is keyed about midway between its minimum and maximum nodes. Methods Left and Right move the rider down the tree by one node along the associated branch, if it exists, while method Previous moves the rider up the tree by one node, if it isn't already at home. Alternatively, method Find locates that node which has the attached key. Once the rider is placed where the programmer wants it, the node's datum can be retrieved with method `GetData`, and its key can be extracted with method `GetKey`.

3.4 Mathematical Fields

There are three numeric types at the heart of BEL: a number `Bel.MF.Numbers.Number`, an array `Bel.MF.Arrays.Array`, and a matrix `Bel.MF.Matrices.Matrix`, where `MF` stands for mathematical field. These three basic types are why BEL was written.

Each of these three numeric types is an implementation of `Bel.Field`, which refines `Bel.Object` by requiring five additional methods (viz., `Negative`, `Add`, `Subtract`, `Multiply` and `Divide`, which are in turn called by their associated overloaded operators) to the five methods already required by a `Bel.Object` (i.e., `Initialize`, `Nullify`, `Clone`, `Load` and `Store`).

3.4.1 Numbers

Like module `Bel.DATA.Keys`, which exports its hidden `KeyType`, module `Bel.MF.Numbers` also exports the hidden `NumberType` for a `Bel.MF.Numbers.Number`, which is presently a 64-bit real number. This is done so that the exported interface of type Number does not depend on any underlying system type. This will allow for any future changes in how a number gets implemented, without adversely affecting legacy code already written that relies on this interface. For example, one could change `NumberType` from a 64-bit real to two 64-bit reals used to represent a complex number without having to change the existing interface of Number at all, which is a desirable feature.

`Bel.MF.Numbers` exports six constants: `Epsilon`, often referred to as machine epsilon, which is the smallest machine number such that $1 + \mathtt{Epsilon} > 1$; `MaximumPositiveNumber` is the largest, positive, machine number *with full digit precision*; `MinimumPositiveNumber` is the smallest, positive, machine number *with full digit precision*; `NaN` is used to represent not-a-number, e.g., $0/0 = \mathtt{NaN}$; `NegativeInfinity` handles machine overflow in the negative direction; while `PositiveInfinity` handles machine overflow in the positive direction. The first three of these constants are defined different from their .NET and Mono counterparts.

The string representation of a Number can be read by the method `Parse`, while a Number can be converted into a string with either method `Typeset`, to a default representation, or via method `ToString`, which allows the programmer to select the number of significant digits to the right of the decimal point in scientific notation that are to appear in the outputted string.

Methods `Get` and `Set` are used by the overloaded assignment operator ":=" so that an assignment can be defined without calling itself. Operator ":=" also handles type conversion, allowing for a seamless interface of Number with the base types of Zonnon, which are inherited from the .NET Framework.

There are a number of methods that return a boolean value including: `Equals`, `NotEqual`, `LessThan`, `LessThanOrEqual`, `GreaterThan` and `GreaterThanOrEqual`, which are called by their associated overloaded operators "=", "#", "<", "<=", ">" and ">=", respectively. Also exported are methods for checking whether or not a number has the value of one of the various exported constants; in particular, `IsFinite`, `IsInfinite`, `IsPositiveInfinity`, `IsNegativeInfinity` and `IsNaN`.

In addition to the algebraic methods required by the `Bel.Field` definition interface, a Number provides for its `Magnitude`, or absolute value, and for its `Power` raised to another number. The former is called internally by the math function `Bel.MATH.Math.Abs`, while the latter is called internally by the overloaded operator "**".

To gain an appreciation for why type Number was created, look at the sheer number of exported overloaded operators that are needed just to handle all of the types built into the .NET Framework in App. E. It totals about eight pages!

I use instances of type Number for virtually all my programming needs now, except for when an indexer is required, which occurs in `for` loops, or when indexing an array or matrix. Here instances of type `integer` are required and used. The Zonnon language mandates that these programming structures take only integer arguments.[2]

3.4.2 Arrays

Object `Bel.MF.Arrays.Array` places a wrapper around the data type `array {math} * of Bel.MF.Numbers.Number`, whose alias is `MathArray`. The *math* modifier allows a programmer to make use of the recently added math extensions to the Zonnon language [1]. This module also exports like aliases for all of the array types that one can create when using the built-in numeric types of the .NET and Mono frameworks.

An Array is an implementation of `[]`, a feature of the Zonnon language, and of `Bel.Field`. The built-in definition `[]` allows the programmer of any user-defined type, like my Array type, to create an indexer for that type. This is accomplished through two methods that the programmer must write: `Get` and `Set` (cf. App. E). An admissible index for an Array object is any integer in interval $[1, \mathtt{Length}]$, where `Length` is an Array method that returns the number of rows in the data array held by the object. *Note: the indexer for every instance of an `Array` starts at 1, like in `Fortran`; not 0, like in Zonnon, Oberon and C.*

Method `Create` is analogous to a constructor for this type. It creates an internal data array at the length specified in its argument. There are no destructors in `.NET` languages. A garbage collector is used to reclaim unused memory. Method `Nullify` is designed to aid in the garbage collection process by setting the dynamically allocated, internal, data array to `nil`, which disconnects the pointer between an Array and its internal `MathArray`, making the internal math array eligible for garbage collection. The result of calling `Nullify` is an Array with length zero.

Arrays can be assigned element-by-element using their indexer, or in segments by calls to `GetSubArray` and `SetSubArray`, or by direct assignment through a number of methods. For getting, `GetArray` returns an `ArrayOfNumber`, while `GetMathArray` returns a `MathArray` of Number. For setting, `SetArray` is called internally by the overloaded assignment operator ":=" to allow this operation to take place without calling itself, and then there are set procedures for each of the exported array types.

There are two boolean functions that are Array methods; they are `Equals` and `IsANumber`. `Equals` compares two arrays to determine if they are equivalent or not. An Array becomes, in effect, a Number whenever it has a length of one, and `IsANumber` checks for this condition.

Method `Dot` takes the dot product between two arrays of the same length, for example `n := a.Dot(b)` is Zonnon code for

$$n = \boldsymbol{a} \cdot \boldsymbol{b} \quad \text{or} \quad n = \sum_{i=1}^{\mathtt{Length}} a_i b_i$$

for number n and the two arrays $\boldsymbol{a}$ and $\boldsymbol{b}$.

Method `Normalize` divides the elements of an array by its norm so that the norm of a normalized array is 1. The normalizing norm is returned as the argument of this procedure.

Method `Sort` rearranges its elements in ascending order, i.e., $a_1 \le a_2 \le \cdots \le a_{\mathtt{Length}}$, while method `Swap` exchanges the elements held in the array at the two specified rows.

In addition to the many methods or type-bound procedures of a `Bel.MF.Arrays.Array`, module `Bel.MF.Arrays` exports three procedures whose arguments are instances of Array. They are the `OneNorm`, the `TwoNorm` and the `InfinityNorm`. For an array $\boldsymbol{a}$, the `OneNorm` returns $\sum_{i=1}^{\mathtt{Length}} |a_i|$, the `TwoNorm` returns $\left(\sum_{i=1}^{\mathtt{Length}} (a_i)^2\right)^{1/2}$ and the `InfinityNorm` returns $\max_{i=1}^{\mathtt{Length}} |a_i|$.

2. An earlier version of this software had two number types: `Real` and `Integer`. But the inability to use `Integers` as indexers of `for` loops and arrays within Zonnon code made this design choice ineffective. At first I was disappointed, but upon reflection, the outcome led a simpler design—the single numeric type: Number.

3.4.3 Matrices

Object `Bel.MF.Matrices.Matrix` places a wrapper around the data type `array {math} *, * of Bel.MF.Numbers.Number`, whose alias is `MathMatrix`. This module also exports like names for all of the matrix types that one can create using the built-in numeric types of the .NET and Mono Frameworks. A call to method `Create` constructs a `Matrix`'s internal `MathMatrix` to the specified size, whose dimensions are supplied back to the user through methods `Rows` and `Columns`.

Matrix implements the interfaces `[]` and `Bel.Field`. In addition to the two required methods for the indexer `[]`, i.e., `Get` and `Set`, a Matrix has methods `GetRow` and `SetRow`, `GetColumn` and `SetColumn`, and `GetDiagonal` and `SetDiagonal`, which extract and assign subarrays to a matrix. It also has methods `GetSubMatrix` and `SetSubMatrix` to handle submatrix extrations and assignments. For get procedures, `GetMatrix` returns a `MatrixOfNumber`, while `GetMathMatrix` returns a `MathMatrix`. `SetMatrix` is called internally by the overloaded assignment operator ":=" to allow this operation to take place without calling itself, and then there are set procedures for each of the exported matrix types.

There are five exported boolean functions; they are: `Equals`, `IsAnArray`, `IsARowVector`, `IsAColumnVector` and `IsANumber`. `Equals` compares two matrices to determine if they are equivalent or not. A Matrix contains `Arrays` and `Numbers` as special cases, and checks for these conditions are provided through the latter four methods. `IsAnArray` will return *true* if either `IsARowVector` or `IsAColumnVector` returns *true*.

There are seven contraction methods provided by type Matrix. Method `Dot` is coded in Zonnon as `M := A.Dot(B)` for its math equivalent

$$\boldsymbol{M} = \boldsymbol{A} \cdot \boldsymbol{B} \quad \text{or} \quad M_{ij} = \sum_{k=1}^{K} A_{ik} B_{kj},$$

for any three compatible matrices $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{M}$. Method `DotTranspose` is coded in Zonnon as `M := A.DotTranspose(B)` for its math equivalent

$$\boldsymbol{M} = \boldsymbol{A} \cdot \boldsymbol{B}^T \quad \text{or} \quad M_{ij} = \sum_{k=1}^{K} A_{ik} B_{jk}.$$

Method `TransposeDot` is coded in Zonnon as `M := A.TransposeDot(B)` for its math equivalent

$$\boldsymbol{M} = \boldsymbol{A}^T \cdot \boldsymbol{B} \quad \text{or} \quad M_{ij} = \sum_{k=1}^{K} A_{ki} B_{kj}.$$

Method `Contract` is coded in Zonnon as `a := A.Contract(b)` for its math equivalent

$$\boldsymbol{a} = \boldsymbol{A} \cdot \boldsymbol{b} \quad \text{or} \quad a_i = \sum_{k=1}^{K} A_{ik} b_k,$$

for any two compatible arrays $\boldsymbol{a}$ and $\boldsymbol{b}$, and matrix $\boldsymbol{A}$. The method `TransposeContract` is coded in Zonnon as `a := A.TransposeContract(b)` for its math equivalent

$$\boldsymbol{a} = \boldsymbol{A}^T \cdot \boldsymbol{b} \quad \text{or} \quad a_i = \sum_{k=1}^{K} A_{ki} b_k.$$

Method `DoubleDot` is coded in Zonnon as `n := A.DoubleDot(B)` for its math equivalent

$$n = \boldsymbol{A} : \boldsymbol{B} \quad \text{or} \quad n = \sum_{k=1}^{K} \sum_{\ell=1}^{L} A_{k\ell} B_{\ell k},$$

for any number n. And finally, method `TransposeDoubleDot` is coded in Zonnon as `n := A.TransposeDoubleDot(B)` for its math equivalent

$$n = \boldsymbol{A}^T : \boldsymbol{B} \quad \text{or} \quad n = \sum_{k=1}^{K} \sum_{\ell=1}^{L} A_{k\ell} B_{k\ell}.$$

In the above sums, K and L associate with the appropriate dimension of an array or matrix. All of these procedures require dimensional compatibility across the summing indices.

Three methods remain. `Normalize` divides all elements of its data matrix by the matrix's norm, so that the Frobenius norm of the matrix is one.

SwapRows exchanges the specified rows in the internal matrix, while SwapColumns exchanges the specified columns.

In addition to these type-bound procedures belonging to a Bel.MF.Matrices.Matrix, module Bel.MF.Matrices exports four procedures; three are norms, and the fourth is the vector product. Procedure OneNorm ruturns the maximum vector one-norm taken over all of its columns. Procedure FrobeniusNorm returns the square root of the sum of the squares of all elements. And procedure InfinityNorm returns the maximum vector one-norm taken over all of its rows.

Procedure VectorProduct codes in Zonnon as M := VectorProduct(a, b), whose math equivalent is

$$\boldsymbol{M} = \boldsymbol{a} \otimes \boldsymbol{b} \quad \text{or} \quad M_{ij} = a_i b_j \ \ \forall \ \ i, j,$$

returning that matrix $\boldsymbol{M}$ which is the vector product of arrays $\boldsymbol{a}$ and $\boldsymbol{b}$.

3.5 Mathematics

BEL contains a modest set of math libraries. They are split into three topical regimes: functions, the calculus and matrix algebra. The author has written a variety of math procedures in Oberon that he would like to port over to BEL in future releases.

3.5.1 Series

Bel.MATH.Series exports three kinds of series that are either infinite series summed to some convergence criteria, or that are truncated series. These are: continued fractions, power series and rational power series. A continued fraction has the form

$$y = b_0 + \frac{a_1 x}{b_1 + \frac{a_2 x}{b_2 + \frac{a_3 x}{b_3 + \cdots}}}.$$

A power series has the form

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots .$$

And a rational power series has the form

$$y = \frac{a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots}{b_0 + b_1 x + b_2 x^2 + b_3 x^3 + \cdots}.$$

These kinds of series are often used to define or approximate mathematical functions in computer applications.

3.5.2 Functions

Module Bel.MATH.Functions exports two of the basic mathematical constants; they being: E $= e^1 = 2.71828182845905$ and Pi $= \pi = 3.14159265358979$, when expressed to machine accuracy. Functions also exports the more common math functions that one will likely have need of:

Random returns a random number in [0,1].

Max returns the greater of its two arguments.

Min returns the lesser of its two arguments.

Ceiling returns the smallest integer value that is greater than or equal to its argument.

Floor returns the greatest integer value that is less than or equal to its argument.

Round returns the integer value that is closest to its argument.

Abs returns the absolute value of its argument.

Sign returns one if its argument is greater than zero, minus one if its argument is less than zero, and zero if it is zero.

Sqrt returns the square root of its argument.

Pythag returns the Euclidean distance for its two arguments, i.e., $\sqrt{x^2 + y^2}$.

Log returns the base 10 logarithm of its argument.

Ln returns the natural logarithm of its argument.

Exp returns the exponential of its argument.

Sin returns the sine of its argument.

Cos returns the cosine of its argument.

Tan returns the tangent of its argument.

ArcSin returns the inverse sine of its argument.

ArcCos returns the inverse cosine of its argument.

ArcTan returns the inverse tangent of its argument.

ArcTan2 returns the quadrant-correct inverse tangent of its two arguments, i.e., $\tan^{-1}(y/x)$, where y is its first argument and x is its second.

`Sinh` returns the hyperbolic sine of its argument.

`Cosh` returns the hyperbolic cosine of its argument.

`Tanh` returns the hyperbolic tangent of its argument.

`ArcSinh` returns the inverse hyperbolic sine of its argument.

`ArcCosh` returns the inverse hyperbolic cosine of its argument.

`ArcTanh` returns the inverse hyperbolic tangent of its argument.

`Gamma` returns the gamma function of its argument.

`Beta` returns the beta function for its two arguments.

3.5.3 Interpolations

`Bel.MATH.Interpolations` provides functions to interpolate between data points.

Neville's algorithm provides an efficient implementation of Lagrange interpolation. If the $\boldsymbol{x}$'s represent inputs and the $\boldsymbol{y}$'s outputs, with arrays `xVec` and `yVec` holding their values, then `Neville` returns the Lagrange interpolation for y at some intermediate point 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` $\leq$ `atX` $\leq$ `x2` and `y1` $\leq$ `atY` $\leq$ `y2`.

3.5.4 Distributions

`Bel.MATH.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` distributions for any degree of freedom, and the `ChiSquared` distribution up through 100 degrees of freedom.

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

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 N degrees of freedom, then the probability that $P\left(\left|X/\sqrt{\chi^2/N}\right| \leq t\right)$ is distributed Student's t. `StudentT` returns $t_\rho(N)$ for probabilities (or percentage-point certainties) of ρ.

Given that X_1^2 and X_2^2 are independent random variables distributed chi-squared in N_1 and N_2 degrees of freedom, respectively, then the variance ratio $F = (X_1^2/N_1)/(X_2^2/N_2)$ is distributed F with distribution function $P(F|N_1, N_2)$. `F` returns $F_\rho(N_1, N_2)$ for probability ρ. The ordering of N_1 and N_2 matters. An important reflexive property of the F-distribution is that $F_{1-\rho}(N_1, N_2) = 1/F_\rho(N_2, N_1)$.

3.5.5 Derivatives

Module `Bel.MATH.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` 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.

3.5.6 Integrals

`Bel.MATH.Integrals` exports the enumerated type `Method` from which the programmer can choose one of five methods for numerically integrating a user-supplied function. The choices include: `trapezoidal`, `simpson`[3], `threeEights`, `romberg` and `gauss`. Procedure `Integrate` 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.

3.5.7 Runge-Kutta

`Bel.MATH.RungeKutta` implements three Runge-Kutta (RK) integrators, each with a different order of accuracy, chosen by selecting an instance of enumerated type `Order`, which includes options: `second`, `third` and `fourth`. The second-order method is based on the quadrature and weights of trapezoidal integration, the third-order method is based of the quadrature and weights of Simpson's rule, and the fourth-order method is KUTTA's most accurate integrator, his $3/8^{\text{th}}$ rule [3].

Procedure `Solve` advances the solution to a system of ordinary differential equations (ODEs) established by argument '`f`' (an instance of exported procedure type `F`) by one time step '`h`' into the future at a relative error `err` that does not exceed some specified tolerance `tol`, usually set to a value of 10^{-p} where p is the order of the method. For example, to assure a result with three significant figures of accuracy, set `tol` to 0.001. 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`. And for four or more significant figures in accuracy, set `tol` accordingly and choose the RK integrator with `ord` set to `fourth`.

The last four arguments of procedure `Solve` are **`var`** variables whose sent values and updated and returned. The first of these variables is the independent variable '`x`', whose original value is x_{n-1} and whose updated value is $x_n = x_{n-1} + h_{n-1}$. The second is an array holding the dependent variables '`y`', whose original value is y_{n-1} and whose updated value is y_n. These first two values are typically passed unaltered from step to step. The third of these varying variables holds the step size '`h`', whose sent value is h_{n-1} and whose returned value is h_n, as determined by the PI (proportional integral) controller of GUSTAFSSON, LUNDH and SÖDERLIND [4]. If the integration is to be done with fixed step size, then '`h`' needs to be reset between consecutive calls. The last of these fields provides an asymptotically correct estimate for the local truncation error, `err`, whose sent value comes from the prior step and is used by the PI controller, and whose returned value is for the current step.

The initial conditions are set in the first call to `Solve`. One must select a reasonable step size '`h`' to begin the integration with. Too large or too small and the integrator will spend time in the beginning of the solution adjusting itself to a reasonable value. Also, one needs to provide values for the independent '`x`' and dependent '`y`' variables, where $x_0 = 0$, typically, and where $y(x_0) = y_0$. One also needs to set the error `err`. It is important that one set $\text{err}_0 = 0$ so that the I controller, not the PI controller, is used here, as there is no history for the PI controller to work with.

A few additional comments are in order here. First, the exported variable `count` specifies the number of function evaluations required to obtain a solution. To determine the number of integration steps that took place, devide `count` by three times the order of the integrator. The three arises because Richardson extrapolation is used to acquire an asymptotically correct estimate for the relative truncation error over the local time step [5]. This solver will automatically divide an interval into two halves, and recursively call itself, whenever an asymptotic error estimate exceeds the error tolerance specified by the user. That is why the actual number of integration steps may well exceed the number specified through the assignment of '`h`'.

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

3.5.8 Linear Algebra

`Bel.MATH.LinearAlgebra` has, at present, just one matrix solver. It is Crout's LU decomposition employing partial pivoting, which is implemented in type `Lu`. Method `Initialize` gets `Lu` ready for the factorization of a matrix into lower- and upper-triangular matrices. Method `Factorize` does this factorization using Crout's routine. Method `Dimension` returns the dimension of the factorized matrix. Method `Rank` returns the rank of the factorized matrix. Method `Determinant` returns the determinant of the factorized matrix. Method `Inverse` returns the inverse of the factorized matrix, if it exists. And method `Solve` solves a linear system of equations whose matrix has been factorized.

In addition to object `Lu`, this module exports the procedure `RefineInverse`, which applies a single iteration of Hotelling's iterative refinement method to improve upon the accuracy of a matrix inverse.

3.5.9 Newton-Raphson

write this section...

4 Applications

Two applications are included in BEL to illustrate how it can be used as a basis for developing software for specialized problems of a computational nature. The first is a genetic algorithm (GA) for parameter estimation, and the second is a continuum analysis of membranes for a boundary-value problem that is indicative of a class of experiments performed on soft tissues.

Goldberg [6] tells his readers what genetic alogorithms are in his opening paragraph:

> "Genetic algorithms are search algorithms based on the mechanics of natural selection and natural genetics. They combine survival of the fittest among string structures with a structured yet randomized information exchange to form a search algorithm with some of the innovative flair of human search. In every generation, a new set of artificial creatures (strings) is created using bits and pieces of the fittest of the old; an occasional new part is tried for good measure. While randomized, genetic algorithms are no simple random walk. They effeciently exploit historical information to speculate on new search points with expected improved performance."

Genetic algorithms have in common a population of individuals, a means to deterimine the fitness of each individual, the pairing of individuals for reproduction according to their fitness, the cross-fertilization of genetic material from two parents to produce two offspring, and a random chance of a mutation occurring in the offsprings' genetic material. The new generation replaces the old one, and the reproduction cycle starts all over again.

Membranes span all living organisms. They are an integral building block of Mother Nature. The Biological Engineering Laboratory at Saginaw Valley State University is being set up to study the response of soft membranous tissues to imposed mechanical loadings, and to model their behaviors with contitutive equations developed using continuum mechanics principles. To accomplish this objective, scalar, vector and tensor fields are provided for two-dimensional space, all of which have the attribute of units—SI units, to be more precise. Physical fields in deforming continua coincide with special differential and integral operators, which are provided here for tensors. The kinematic and kinetic fields that associate with in-plane biaxial extensions are also provided. A simple, isotropic, tissue model is included.

4.1 Genetic Algorithms

Seven modules (presented from the bottom up) were written to implement a genetic algorithm for the purpose of obtaining parameter estimates in functions used to describe observations or experimental data. Their software interfaces are presented in App. G.

The code distributed with BEL is based on the simple genetic algorithm (SGA) written by Goldberg [6] in Pascal, with code fragments being distributed throughtout his text. What appears here is a complete rewrite of SGA in Zonnon. It turns out that GA's are a great application to showcase the capabilities of the Zonnon programming language. In this rewrite of SGA, efforts were taken to remain true to the overall spirit of Goldberg's original SGA [6], implementing refinements made later in [7], and some insights from Sivanandam & Deepa [8], the latter of which is an informative text, but it is poorly written.

4.1.1 Statistics

GA's are designed to maximize the fitness of a species. How one defines 'fitness' is left for the user to specify. Buse's [9] R^2 statistic for generalized least squares is used here for the measure of fitness.

Module `Bel.GA.Statistics` exports two probabilistic functions that are used throughout these GA modules; they are `FlipHeads` and `RandomIntergerBetween`.

`Statistics` exports the enumerated type `LeastSquares` that can take on values of `linear`, `generalized` or `nonlinear`. This type controls the method of optimization used in the analysis. These three methods are equivalent wherever there is just one response variable that is being measured and fit against.

`Statistics` also exports a procedure type, i.e., `Model`, instances of which can be submitted to the GA for parameter optimization. This function has three arguments. The first is an Array of length P that contains the parameters to be estimated. The second argument is a Matrix with C rows (the number of control variables) and O columns (the total number of experimental observations). The third is an Array of length E, which is the number of experiments being represented. This array supplies indices within the interval $[1, O]$ that align with the last entry for each experiment. Say, for example, there are three experiments to be used to fit a model's parameters to. The first has 100 data points, the second, 75, and the third, 90. Then this array would have length 3 with elements $\{100 \quad 175 \quad 265\}^T$ so $O = 265$. A call to a model returns a Matrix in R rows, which is the number of response variables, spanning over the O columns. It contains the predicted response of the model when subjected to the supplied control variables and model parameters.

Procedure `Configure` allows the user to supply experimental data, and a numerical model whose parameters are to be estimated. This procedure has eight arguments, the first three of which contain experimental data. The first is a $C \times O$ matrix that holds the control variables to be forwarded to the user-supplied model. The second and third arguments are $R \times O$ matrices that contain the experimental input and output variables, the latter of which the optimizer will attempt to replicate with the model by systematically adjusting its parameters. These three matrices can be concatenations of several experiments, provided they all have the same sets of control and response variables. The fourth and fifth arguments contain arrays of length E. The fourth specifies the column indices where the last datum point of each experiment resides. The fifth indicates how many of the data points are to be used for optimization from each experiment. Optimization is an expensive process, and it is often useful to decimate large data sets into more managable ones. A decimation scheme similar to that of Doehring *et al.* [10] has been implemented into `Configure`, where the input/output curves are segmented into intervals of like arc length. The sixth argument selects the type of least-squares analysis that is to be used: linear, generalized or non-linear. The seventh argument supplies the numerical model to the optimizer. And the eighth argument, `solveModelWithDecimatedDataOnly`, allows the user to either solve the numerical model at all O data points, or at just the N data points that survived the decimation process. If the model is, for example, a simple function, then one can save time by solving the model at the N decimated points only. On the other hand, if the model is a differential equation, for example, it may be necessary to solve the model at all O data points to get reliable results.

In many cases, the second argument in `Con-`

`figure` will be the same as the first, but not always. In the case of a hypoelastic material model applied to a biaxial boundary condition (cf. Secs. 4.3 & 4.4), components of the deformation $\mathbf{F}$ and velocity $\mathbf{L}$ gradients are the control variables, eight in number, while the stretches λ_1 and λ_2 are the input variables, whose corresponding output variables are the engineering stresses σ_1 and σ_2, each being two in number. In other words, the second and third arguments are what one would use in an XY plot of the experimental data, while the first argument contains the control information needed to secure results from the numerical model in order to contrast theory against experiment.

Procedure `DataFitAgainst` returns the decimated experimental data set used for parameter estimation. The first and second arguments are matrices in R rows and N columns, where N is the actual number of experimental data points used during optimization, i.e., $N \leq O$. The first argument contains the experimental inputs, and is a subset of the second argument supplied to `Configure`. The second arugment contains the experimental outputs, and is a subset of the third argument supplied to `Configure`. The third argument specifies which indices in the two previous matrices are affiliated with the last entry for the associated experiments. It is the analog to the fourth argument of `Configure`, adjusted here to account for the data that have been decimated away by procedure `Configure`.

So, there are now N, experimental, data vectors $\mathscr{y}_n$, $n = 1, 2, \ldots, N$, with components $\mathscr{y}_n = \mathscr{y}(t_n) = \{y_{n1}, y_{n2}, \ldots, y_{nR}\}^T$ evaluated at instances t_n, where R is the number of responses or dependent variables that have been measured. From these data, estimates for P parameters $\boldsymbol{\theta} = \{\theta_1, \theta_2, \ldots, \theta_P\}^T$ that belong to some model $\boldsymbol{\mathcal{Y}}(t, \boldsymbol{\theta})$ are to be acquired. To accomplish this, we first create N residual vectors $\mathscr{x}_n$ with components

$$x_{rn}(\boldsymbol{\theta}) = \frac{y_{rn} - \mathcal{Y}_r(t_n, \boldsymbol{\theta})}{\bar{s}_r},$$

with $n = 1, 2, \ldots, N$ and $r = 1, 2, \ldots, R$, that are normalized by their sample standard deviates $\bar{s}_r$ obtained from

$$\bar{s}_r^2 = \frac{1}{N-1} \sum_{n=1}^{N} (y_{rn} - \bar{y}_r)^2,$$

with sample mean

$$\bar{y}_r = \frac{1}{N} \sum_{n=1}^{N} y_{rn}.$$

Normalizing $y_{rn} - \mathcal{Y}_r(t_n, \boldsymbol{\theta})$ by $\bar{s}_r$ allows response variables with different magnitudes and even with different units to be used, giving them an equal footing. Procedure `Residuals` returns the $R \times N$ matrix containing all of the $\mathscr{x}_n$, while procedure `Theory` returns a like dimensioned matrix containing all of the $\boldsymbol{\mathcal{Y}}(t_n, \boldsymbol{\theta})$, both depending upon the set of parameters $\boldsymbol{\theta}$ being sent.

An $R \times R$, bias-corrected, sample variance $\mathbb{V}$ is used whose components are given by (cf. Bard [11])

$$\mathbb{V}(\boldsymbol{\theta}) = \frac{R}{RN - P} \times \sum_{n=1}^{N} \big(\mathscr{x}_n(\boldsymbol{\theta}) - \bar{\mathscr{x}}\big) \otimes \big(\mathscr{x}_n(\boldsymbol{\theta}) - \bar{\mathscr{x}}\big).$$

The sample mean $\bar{\mathscr{x}}$ for data $\mathscr{x}_n$ has components

$$\bar{x}_r = \frac{1}{N} \sum_{n=1}^{N} x_{rn},$$

which go to zero in a neighborhood around an optimal solution $\boldsymbol{\vartheta}$, i.e., $\bar{\mathscr{x}} \to \boldsymbol{0}$ as $\boldsymbol{\theta} \to \boldsymbol{\vartheta}$, because there the model $\boldsymbol{\mathcal{Y}}(t, \boldsymbol{\vartheta})$ provides its 'best' mean description of the data.

In the statistical assessment that follows, vectors $\mathscr{x}_n$ are the data, not the $\mathscr{y}_n$.

The coefficient of determination, otherwise known as the R^2 statistic, relates the unexplained variance of a model to the total variance that is present in the data. Its formula is not unique when extended to non-linear optimization algorithms, and many methods have been proposed over the years. The R^2 statistic implemented here is due to Buse [9], who derived it for generalized least squares; specifically,

$$R^2 = 1 - \frac{1}{N} \sum_{n=1}^{N} \frac{x_n^T(\theta) \cdot \widetilde{\mathbb{V}}^{-1}(\theta) \cdot x_n(\theta)}{\frac{1}{N} \sum_{\nu=1}^{N} z_\nu^T \cdot \widetilde{\mathbb{V}}^{-1}(\theta) \cdot z_\nu},$$

with vector z_n having components

$$z_{rn} = \frac{y_{rn} - \bar{y}_r}{\bar{s}_r}.$$

Vector z_n represents a standardized random variable in the experimental response r, whose mean is 0 and whose variance is 1. Procedure `RSquared` returns the value for R^2 given a set of parameters θ.

The above formula for R^2 reduces to the well-known result from linear least squares whenever $\mathbb{V} = \sigma^2 \mathbb{I}$, where

$$\sigma^2(\theta) = \frac{1}{RN - 1} \sum_{r=1}^{R} \sum_{n=1}^{N} \big(x_{rn}(\theta) - \bar{x}_r \big)^2,$$

and to Buse's [9] result for generalized least squares whenever $\mathrm{V}_{rr} = \sigma_r^2$, $r = 1, 2, \ldots, R$, where

$$\sigma_r^2(\theta) = \frac{1}{N - 1} \sum_{n=1}^{N} \big(x_{rn}(\theta) - \bar{x}_r \big)^2.$$

Consequently, the convariance matrix $\widetilde{\mathbb{V}}$ that appears in the above formula for R^2 is selected according to

$$\widetilde{\mathbb{V}}(\theta) = \begin{cases} \sigma^2(\theta)\, \mathbb{I} & \texttt{linear} \\ [\![\sigma_r^2(\theta)\, \mathrm{I}_{rr}]\!] & \texttt{generalized} \\ \mathbb{V}(\theta) & \texttt{nonlinear} \end{cases}$$

based upon the value for `LeastSquares` asigned via procedure `Configure`. Procedure `Covariance` returns the selected $\widetilde{\mathbb{V}}$ matrix of dimension $R \times R$, given a set of parameters θ.

4.1.2 Genes

A gene is a bit in a memory bank that belongs to the overall cache of biological memory that resides within a living organism.

Because different forms of life have different types of genes, and two are coded into BEL, a Zonnon definition is provided that requires a `Bel.Gene` to implement the following six methods, cf. App. B.

`Initialize` applies even-odds probability to assign a random value for its allele (the expression of that gene). Allele are randomly assigned in the beginning, i.e., during procreation where the first colony of individuals originate from. In later colonies, the allele are assigned through chromosome splitting (occurs with high probability) with the possibility of an additional gene mutation (occurs with small probability), both of which take place at the time of reproduction.

Method `Clone` has a different outcome from that of a `Bel.Object.Clone`. `Clone` produces a duplicate or deep copy of its genetic material that, from a biological perspective, makes it the appropriate terminology to use to describe its action. In computer science, a clone would have the same type, but it would be only a shell of the original, minus any substantive data.

Method `IsEqualTo` tests two genes to determine if they are equivalent.[4]

`Parse` and `Typeset` are used to assign and extract the allele of a gene after its initial creation. This is done with strings.

Method `Mutate` performs a random mutation on a gene, switching its value at some specified probability for mutation. If a gene is mutated, it increments a counter that is used to keep track of the number of mutations that have occurred for output in the final report, cf. App. G.

Two types of genes are implemented in BEL. A `Bel.GA.Genes.Biallelic` gene has two expressive states: dominant and recessive. A dominant gene is expressed as 1, while a recessive gene is expressed as 0. A `Bel.GA.Genes.Triallelic` gene has three expressive states: dominant, dominant/recessive and recessive. A dominant gene

4. Method name `Equals` makes sense here, but it cannot be used because the argument is an `object` so there is a naming conflict with `System.Object` that the Zonnon compiler uses for its root `object` type. This conflict does not arise for a Number, for example, because there the argument of `Equals` is another Number, not an `object`.

is expressed as 1, a dominant/recessive gene is expressed as ½, and a recessive gene is expressed as 0.

4.1.3 Chromosomes

A chromosome is a bank of memory that is part of the overall cache of biological memory that resides within a living organism.

In this implementation of a GA, each chromosome is paired with a single material parameter that is being sought through optimization. If a model has five unknown parameters that are to be estimated, for example, then its implementation in BEL's GA will result in five unique chromosomes. In this respect, our GA differs from Goldberg's SGA. His has just one chromosome whose genotypes for all parameters are spliced into a single long strand of genes.

Like genes, two different types of chromosomes are coded into BEL, so a definition has been assigned that binds nine procedures to implementations of this type, cf. App. B.

Method `Initialize` receives the lower and upper bounds that associate with the parameter that this chromosome is assigned to, and the number of significant figures in accuracy that one would hope to see in the final result (I used 4 in the following example). These three factors establish the number of genes required by that chromosome, whose value is thereafter exported through method `Length`.

Method `IsEqualTo` checks to see if the strings of gene expression between the two chromosomes are the same. This does not necessarily mean that their genetic material is identical. `IsEqualTo` is called by operators "=" and "#".

`Parse` and `Typeset` are used to assign and extract the genotype (a string of allele) held by the chromosome.

Method `Mutate` forwards its call to all genes within the chromosome string, where the actual mutations take place.

`Encode` and `Decode` are analogs to `Parse` and `Typeset`, except they import and export the phenotype (i.e., the model parameter) held by the chromosome. They provide the mapping functions that bridge a model, whose parameters are to be estimated, with the GA that will estimate them.

The internal dominance map between genotype and phenotype need not be bijective. All other maps used to encode/decode are one-to-one. Gene expression is Gray encoded in our GA [12]. Gray numbers are a binary-like representation whose neighboring values differ by one bit only, which is not true of binary numbers. Gray code is mapped into binary code. Binary code is mapped into integers. And integers are then converted into their corresponding real phenotype in `Decode`. `Encode` runs these maps in reverse.

A `Bel.GA.Chromosomes.Haploid` implements both the `[]` and `Bel.Chromosome` interfaces, and is comprised of a string of `Biallelic` genes, which the module exports as type `BiChromosome`. Such a chromosome might look like

$$1 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1$$

that, in this case, is a string of eight biallelic genes. The `Get` and `Set` procedures required by `[]` accept values between 1 and `Length`.

A chromosome has two types of riders that position differently. An 'index' rider points directly to a gene, and is used for mutation. Its value resides within the interval $[1, \texttt{Length}]$. A 'locus' rider points inbetween two genes, and is used for crossover (chromosome splitting). Its value belongs to the interval $[1, \texttt{Length} - 1]$. The overloaded indexer for both BEL chromosomes associates with a chromosome's 'index' rider.

The `Bel.GA.Chromosomes.Diploid` chromosome implements the same interfaces that a `Haploid` chromosome does, and exports the same set of methods. A `Diploid` chromosome is, however, very different behind the scenes. It contains two strings of `Triallelic` genes, which the module exports as type `TriChromosome`. This pairing affects the mapping between genotype and phenotype which, unlike a `Haploid` chromosome, is no longer bijective. Such a chromosome might

	0	%	1
0	0	0	1
%	0	1	1
1	1	1	1

Table 1: Hollstien triallelic dominance map.

look like

$$\begin{array}{cccccccc} 1 & \% & 0 & 1 & 0 & \% & 0 & 1 \\ 0 & 0 & 0 & 1 & \% & \% & 1 & \% \end{array}$$

that, in this case, is a pair of strings holding eight triallelic genes each.

The single-index, triallelic, dominance map of Hollstien presented in Table 1 is used to decode these chromosome pairs [6]. The allele of the two strings (labeling the rows and columns) determines the binary gene expression (listed in the matrix of boxes) exported by the decoder. This mapping between gene expression $\{0, 1\}$ and allele $\{0, \%, 1\}$ is obviously not bijective. The above example of a diploid chromosome has a gene expression of

$$1 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1 \ ,$$

which is equivalent to that of the haploid example given earlier.

In order for this dominance map between gene expression and its allele to produce 0s with a probability of $1/2$, and 1s with a probability of $1/2$, it is necessary that a `Triallelic` gene `Initialize` with a random assignment of allele that weigh 50% as 0s, 25% as %s and 25% as 1s [6], which is how this gene type is initialized.

In order for a triallelic gene to `Encode` from its binary representation of gene expression down to its allele, with a random assignment of 0s and 1s that is consistent with its initialization, then an expressed gene of binary 1 must map into triallelic pairs with equal probabilities of $1/6$ for each of its six possible pairings: (1,1), (1,0), (1,%), (%,1), (%,%) and (0,1); while an expressed gene of binary 0 must map into triallelic pairs with probabilities of: $2/3$ for the pair (0,0), and $1/6$ for the pairs (%,0) and (0,%). This is how a diploid gene gets encoded in BEL.

Module `Bel.GA.Chromosomes` also exports the procedure `Crossover`, which is called by higher-level modules to mate two individuals and return their offspring. This is where crossover (the splitting and splicing of genes at reproduction) actually occurs. The first argument establishes the probability for crossover to occur. The second pair of arguments provides the happy parents. The fourth argument is a counter that gets incremented if crossover takes place; it is a statistic provided in the final report. The final pair of arguments are the offspring of this mating. A flip of a coin determines which child gets which parent's genetic material.

`Crossover` implements a two-point crossover procedure [8]. As an example, consider two parents whose chromosomes are of length six. Via a random generator it is determined that the left splice is to take place between the second and third genes, while the right splice is to take place between the fourth and fifth genes, e.g.,

$$\begin{array}{cccccc} 1 & 0\,| & 1 & 1\,| & 0 & 0 \\ 0 & 1\,| & 0 & 1\,| & 0 & 1 \end{array} \rightarrow \begin{array}{cccccc} 1 & 0\,| & 0 & 1\,| & 0 & 0 \\ 0 & 1\,| & 1 & 1\,| & 0 & 1 \end{array} ,$$

where the parents on the left produce the children on the right.

Schemata are the building blocks of genetic alogithms; they are what make them work [6, 7]. A schema is string of genes, shorter than a chromosome; it is a subset to a chromosome. In a genetic algorithm, high-performance, low-order schemata receive exponentially increasing numbers of trials in successive generations, which is the fundamental theorem of genetic algorithms [6, 8]. This is why they outperform other stochastic methods, like Monti Carlo techniques. In schemata, some of the genes take on fixed expressions, while the others are allowed to have any expression, and are denoted with a $\star$. For example, a schema for a haploid chromosome might look like $1 \ \star \ 0$ of which instances $1 \ 0 \ 0$ and $1 \ 1 \ 0$ would both belong. In this example, the schema is of dimension three.

4.1.4 Genomes

A genome is the whole cache of biological memory, containing the evolutionary history of a living organism.

This GA differs from SGA by including the concept of a genome, taken here to be the collection of all chromosomes belonging to an individual. Recall that each model parameter associates with its own chromosome. Therefore, there is a one-to-one correlation between the array of numbers that are a model's parameters and the array of chromosomes that comprise its genome. Different models will have different genome; they are akin to being different species. The human has 23 diploid chromosomes, each comprised of two gene strings, one taken from each parent.

There is just one genome in BEL's GA, so a definition type is not needed. Module `Bel.GA.Genomes` exports two ancillary types: `ChromosomeType`, which is an enumeration type with admissible values being `haploid` and `diploid`, and `Genotype`, which is an `array * of object{Bel.Chromosome}` for which the predominant type `Genome` is a wrapper.

`Genome` implements the Zonnon indexer `[]` interface, which provides access to the various strands in a genome, with admissible values belonging to the interval [1, `Strands`], where method `Strands` returns the number of chromosomes in the genome, i.e., the number of parameters to be estimated.

Method `Clone` returns an exact duplicate of the genetic material that makes up the genome.

Method `Equals` compares two genome, and is called by the overloaded operators "=" and "#". It tests for equality of gene expression, which need not imply that the genetic material of two genome actually be identical.

Methods `Mutate`, `Encode` and `Decode` all forward their requests down to its chromosomes, where the appropriate actions are taken.

Module `Bel.GA.Genomes` exports one procedure; it is `Crossover`. It redirects this request to its individual chromosomes for action.

4.1.5 Individuals

We now move up to the level of a physical being, an entity, or an organism. The module `Bel.GA.Individuals` exports type `Lineage`, which keeps track of an individual's ancestral tree. This type makes available an individual's `birthID`, which is akin to a social security number; its `fitness`, which determines its chances for mating; and its parameters, which establish its genetics.

The main type exported by this module is an `Individual`. Because this is a reference type, it must be created with a **`new`** statement. Method `Initialize` should be called immediately after that. Two arguments are passed with this call. The first, `fixedParameters`, is an array of those model parameters that the user chooses to fix, i.e., the optimizer will not vary these. The second argument, `varyParameters`, is a boolean array with length 1 greater than the sum of all parameters, viz., those allowed to vary, and those being held fixed. Say, e.g., that your model has 5 parameters, the second and fourth of which you can get reliably from a graphical method. In this case, `varyParameters` would be boolean array of length 6 with values $\{F, T, F, T, F, T\}^T$, with the [0] element not being used, and therefore set to *`false`*.

Each individual has a unique phenotype that associates with a unique set of model parameters that are stored in its `lineage` as `parameters`. This array of the model's parameters contains both those that are fixed and those that have been allowed to vary.

Two individuals can be discerned as being clones via method `Equals`, which is called by the overloaded operators "=" and "#".

An individual's genome can be retrieved from `Get`.

All requests to `Mutate` are forwarded down to its genome to handle.

An individual can come into being via three different routes. `Procreate` is used to create the first colony of individuals. It requires arguments that specify the lower and upper bounds defining the interval that each parameter is allowed to vary over, the number of significant figures in accuracy hoped for in the final solution, and what type of chromosome is to be used.

The second way an individual can be created is by an `Alien` migrating into the population. An alien individual has the same arguments that a procreated individual does, plus it specifies what its phenotype is (i.e., its model parameters); whereas, a procreated individual has its phenotype assigned by chance. Aliens are useful when you want to seed a new colony with an individual you deem to have good fitness in an effort to help reduce the required number of generations needed to achieve convergence.

The third and final way to create an individual is to `Conceive` it through the mating of two distinct individuals, i.e., its parents. This is the means by which all individuals belonging to generations two and up are created, except for immigrants. A conceived individual has the same arguments that a procreated individual does, plus it specifies its genotype.

4.1.6 Colonies

The highest level in the hierarchy of our GA (there are five levels) is the population of individuals that makes up a colony. Pairs of individuals from an existing colony are selected for mating through tournament play, whose offspring will comprise the next generation for the colony. The contestants for tournament play are selected at random from the existing population of individuals, with the most fit contestant selected being allowed to mate. The number of contestants s in tournament play is set internally at [7]

$$s = \lceil 1/(1 - p_c) \rceil,$$

where p_c is the probability of crossover.

Some GAs allow their population size to vary from one generation to the next. Here the colony size n is held fixed over the generations, assigned internally according to the formula [7]

$$n = \lceil \chi^k (k \ln \chi + \ln \ell) \rceil,$$

where χ is the cardinality of the gene alphabet being used (2 for haploid, 3 for diploid), k is the bounding dimension for which all schemata up to that length are to be successfully sampled, recombined and resampled, while ℓ is the number of genes in the genome. The user affects this value through his/her choices for the variables `dimensionOfSchemata` and `numberOfSignificantFigures`, the former of which is assigned to k, and the latter through its influence on the number of genes ℓ required by the encoder/decoder.

The BEL GA manages its population somewhat different from that of the SGA. Here the *elite*, or most fit, individual is retained from the prior generation, which is why the population size will always be odd. Furthermore, if two individuals have equal gene expression, then the younger is forced to migrate out of the colony, being replaced with a procreated immigrant. Immigrants can also arise as the conquests of tournament play at an incident level specified by the user. The number of immigrants is a statistic documented in the final report. These population management processes help to maintain diversity, and to also ensure convergence [13]. It is a truly remarkable result that the number of generations needed for convergence t_c can be estimated *a priori* via the formula [7]

$$t_c = \lceil \sqrt{\ell} \, \ln n / \ln s \rceil,$$

where ℓ, n and s are as defined above.

Module `Bel.GA.Colonies` exports two types. The predominant type `Colony` places a wrapper around `Inhabitants = array * of Bel.GA.Individuals.Individual`.

To `Initialize` a `Colony` requires a whole slew of arguments. The first eight are the same as those required by `Statistics.Configure`. After that there are: an array of possibly zero length containing parameters whose values the optimizer cannot adjust, a set of alien parameters introduced to seed the optimizer, lower and upper bounds to establish intervals over which an optimal set of parameters is to be found. The last three of these arrays pertain to only those parameters whose values the optimizer is allowed to adjust. This is followed by a boolean array that tells the optimizer how to splice the fixed and variable parameters into a single strand that can be passed onto the numerical model, which is followed by the dimension of schemata that are to be captured, the number of significant figures in accuracy that are desired in the solution, the percentage-point statistic for certainty to be used for acquiring confidence regions and intervals in the parameters, the probabilities for crossover, mutation and immigration, and what type of chromosome is to be used.

Method `Propagate` advances the colony to its next generation.

Method `Parameters` returns the phenotype from the elite individual in the colony for the most current generation.

To establish confidence regions and intervals, let θ be an estimate of the best parameters ϑ obtained by maximizing the R^2 statistic, with parameters ϑ being taken from the most fit individual in the final population. A confidence region associated with a probability of certainty ρ that surrounds parameters ϑ can be described by the inequality [14]

$$\sum_{n=1}^{N} \frac{x_n^T(\theta) \cdot \widehat{\mathbb{V}}^{-1} \cdot x_n(\theta)}{\sum_{\nu=1}^{N} x_\nu^T(\vartheta) \cdot \widehat{\mathbb{V}}^{-1} \cdot x_\nu(\vartheta)} - 1 \le \frac{P}{N-P}\, F_\rho(P, N-P),$$

wherein

$$\widehat{\mathbb{V}} = \begin{cases} \sigma^2(\vartheta)\, \mathbb{I} & \texttt{linear} \\ [\![\sigma_r^2(\vartheta)\, \mathrm{I}_{rr}]\!] & \texttt{generalized} \\ \mathbb{V}(\vartheta) & \texttt{nonlinear} \end{cases}$$

is taken to be the 'actual' covariance matrix. The above expression gives the correct confidence region for linear models, but it only approximates the actual region for non-linear models, whose exact boundary is much more costly to compute [15]. Nevertheless, this formula adequately serves our needs.

The remaining methods are used to write a report to file. They provide a condensed version for the history of a colony, along with a number of useful statistics, and the optimum parameters found in that search. `ReportHeader` is to be called before the first generation is brought into existence. `ReportBody` can be called after the arrival of any new generation. And `ReportFooter` is to be called after the final generation has been born. Appendix A contains an example of such a report.

4.1.7 Genetic Algorithm Driver

Module `Bel.GA.GeneticAlgorithm` exports a single procedure, viz., `Optimize`. It is the driver of the BEL GA. There are twenty one arguments that must be sent to this driver. They are now discussed in some detail, as this is the interface that most programmers will use in order to obtain an optimal set of estimated parameters.

The first argument is a string that contains the name of a file where the report is to be written. It will be placed in the `iofiles` directory beneath the directory where your executable code resides. The file will be given a `.txt` extension.

The second argument is a matrix that contains the controlled (or independent) variables belonging to the experimental data sets that are being used to fit the model against. These data drive the user-defined model whose parameters are being estimated.

The third argument is a matrix that contains the input variables belonging to the experimental data sets that are being used to fit the model against. This matrix may or may not be the same as that of the second argument. This third argument supplies the 'X' values for XY plots of theory vs. experiment.

The fourth argument is a matrix that contains the output (or dependent) variables belonging to the experimental data sets that are being used to fit the model against. The GA seeks an optimal fit of these data by adjusting the model's parameters. This matrix contains the 'Y' values for the experimental XY curves to be plotted.

The fifth argument is an array whose length equals the number of individual experiments that make up the experimental data set. Say, for example, there are two experiments in the data set. The first has 93 data points, and the second has 57. In this case, the array sent would have elements $\{93 \quad 150\}^T$. This information is passed onto the numerical model where it may be needed, for example, to reset the initial conditions of an integrator.

The sixth argument is another array with the same dimension as the previous array. This array specifies how many data points are to be used in each experiment by the optimizer. All other data are decimated. The chosen data are selected to be roughly equidistant from one to the next along their experimental XY curves.

The seventh argument selects the method of least-squares analysis that is to be imposed. Acceptable values belong to the enumeration type `LeastSquares` exported from module `Bel.GA.Statistics`, which include `linear`, `generalized` and `nonlinear`. They are equivalent whenever there is only one response variable; otherwise, they are distinct. If the response variables act independent of one-another, and if the noise between experiment and theory is the same over all of the responses, then linear least squares can be used. If the response variables act independent of one-another, but the noise between experiment and theory differs between the responses, then generalized least squares should be used. And if any two response varibles are not independent, then non-linear least squares should be used.

The eighth argument is a user-defined instance of a procedure type that is exported as `Bel.GA.Statistics.Model`. Here is where the programmer must provide their model whose parameters are to be estimated. This procedure needs to evaluate the model at all N experimental data points, and return its results. The model itself can be a function, a differential equation, an integral equation, or whatever. But whatever it is, it can only interact with the GA through this particular interface.

The nineth argument allows the user to select between having the model solved at all data points supplied to the GA via arguments 1–3, or to only be solved at those data points that survived decimation, where the optimizer will get its theoretical inputs. Turning on this option can dramatically reduce the runtime of an optimization. An example of when one might choose to only evaluate the model at the decimated data sites would be if the model were a simple function. An example of when one might not choose to do so would be when the full data set would be needed to give a dense enough history that numerical integration of a differential equation, say, could take place with reasonable accuracy.

The tenth argument lists those model parameters whose values are held fixed, i.e., that the optimizer cannot adjust. This can be an array of length zero, in which case all parameters are adjustable. The next three arguments pertain only to those parameters that can be adjusted.

The eleventh argument is a set of alien parameters for seeding the GA. These may be your best guess at what you expect they could be, or they may be parameters from a previous run that terminated too soon, for whatever reason, or they may be an optimum set obtained from a prior experimental data set. Whatever the situation may be, this alien becomes your Adam and Eve.

The twelveth argument is the minimum or lower bound for the parameters. No parameter is allowed below this boundary.

The thirteenth argument is the maximum or upper bound for the parameters. No parameter is allowed above this boundary. Between these lower and upper bounds lie intervals in which the optimum set of parameters is thought to reside. The tighter these bounds can be set, the better the chance of the GA converging onto the global solution in a reasonable amount of time.

The fourteenth argument provides a boolean array that tells the optimizer how to splice together the fixed and adjustable parameters into a single array of parameters that can then be passed onto the numerical model. The zeroth element of this boolean array is not used. If, e.g., the first element of the array were *`true`*, then the first element in the composite array would come from the first element in the array of varied parameters. If the second element were *`false`*, then the second element of the composite array would come from the first element in the array of fixed parameters. If the third element were *`true`*, the the third element of the composite array would come from the second element in the array of varied parameters, and so on.

The fifteenth argument establishes the maximum length of schemata, or building blocks, that you want the optimizer to be able to handle in its search for a global minimum. Typical values lie between 4 and 8.

The sixteenth argument specifies the number of significant figures that you would like the optimizer to converge to. This has an impact on the lengths that chromosomes take on, and therefore, on the size of the population, too. I usually use 3 to 6.

The seventeenth argument establishes the certainty of the percentage-point statistic that you want to use to get confidence intervals. I usually choose 95%, but 60%, 80%, 90%, 95%, 97.5%, 99% and 99.5% certainties are available to select from. The higher the level of certainty, the wider the confidence intervals will be.

The eighteenth argument specifies the probability for crossover. In other words, it sets the odds for the splitting of a genome, thereby taking part from both parents and passing that onto the children. Otherwise, the chromosomes from the parents are copied to their children unaltered, but which child gets which parent's genetic material is determined by an even-odds flip of a coin. Typical values for the probability of crossover range between 0.5 and 0.8.

The nineteenth argument specifies the probability for gene mutation. A mutated gene swaps the value of its allele, e.g., $0 \rightarrow 1$ or $1 \rightarrow 0$ in a mutated biallelic gene belonging to a haploid chromosome. This probability is small, usually on the order of 0.01 or less. This means that if a genome were to have 100 genes, then that genome would have an even chance of having a single gene mutate.

The twentith argument specifies the probability of an immigrant entering into the gene pool. This immigrant replaces the winner from a round of tournament play as a parent for mating. This helps to maintain diversity in the overall gene pool, and should occur at a fairly low probability. Immigrants are procreated.

The twenty first and final argument selects the type of chromosome to be used, which can either be a haploid (single strand) or a diploid (double strand) chromosome. Goldberg [6] reports there to be no advantage given to the more complex diploid option whenever the model is static; however, when the model is dynamic, then the diploid chromosome can outperform the haploid chromosome, because the dominant/recessive gene 'remembers' further into the past.

x	-3	-2	-1	0	1	2	3
y	-4.0	-1.9	-0.55	0.05	0.65	1.9	4.3

Table 2: Input/output pairs for test case.

4.1.8 An Example

Appendix A provides an example report for the following test problem, which is included as file `testGeneticAlorithm.znn` in the test directory of the BEL software.

The experimental data that were supplied are listed in Table 2, where vector x contains the controlled or independent variables, and vector y contains the response or dependent variables.

The model was the polynomial

$$y = \theta_1 + \theta_2 x + \theta_3 x^3,$$

where θ_1, θ_2 and θ_3 are the model parameters to be fit, whose values, 95% confidence intervals, and R^2 values are recorded in App. A. From the perspective of parameter optimization, this is a linear model in parameters $\boldsymbol{\theta} = \{\theta_1, \theta_2, \theta_3\}^T$.

The optimizer was configured with parameter bounds of

$$\boldsymbol{\theta}_{min} = \left\{ \begin{array}{c} -0.1 \\ -0.1 \\ 0 \end{array} \right\} \quad \boldsymbol{\theta}_{max} = \left\{ \begin{array}{c} 0.1 \\ 1.0 \\ 0.25 \end{array} \right\}$$

and seeded with the alien parameters

$$\boldsymbol{\theta}_{alien} = \left\{ \begin{array}{c} 0 \\ 0.45 \\ 0.125 \end{array} \right\}.$$

The `dimensionOfSchemata` was set at 6, the `numberOfSignificantFigures` was set at 4, the `probabilityOfCrossover` was set at 0.75, implying there is a probability of splitting a chromosome of $0.25 = 0.75/3$ (there are 3 parameters

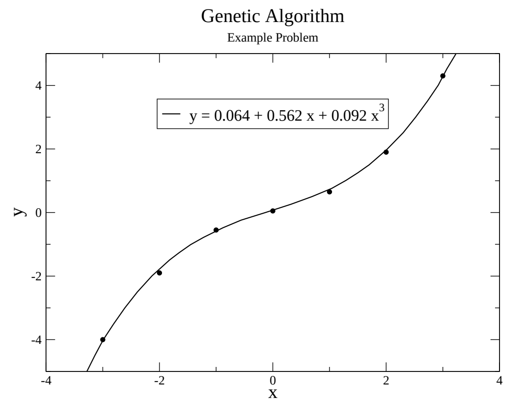

Figure 1: GA fit to raw data, $R^2 = 0.999$.

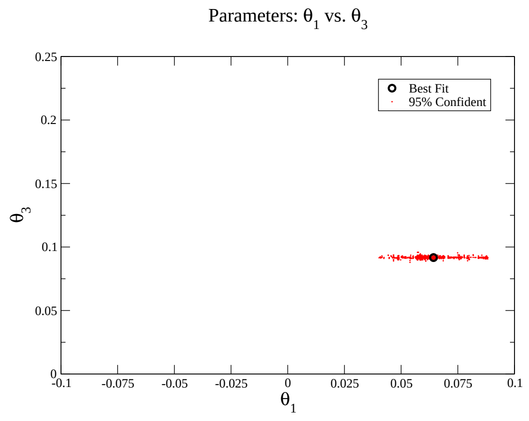

Figure 3: Cloud plot for θ_1 versus θ_3.

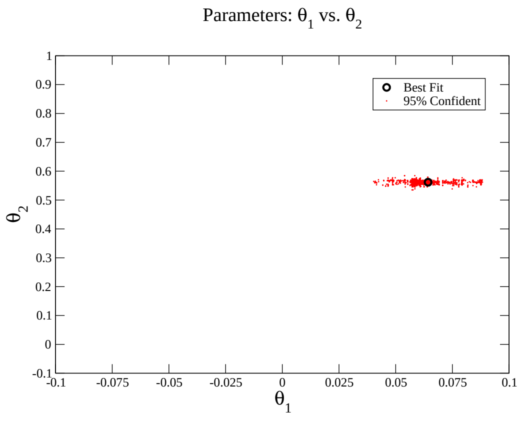

Figure 2: Cloud plot for θ_1 versus θ_2.

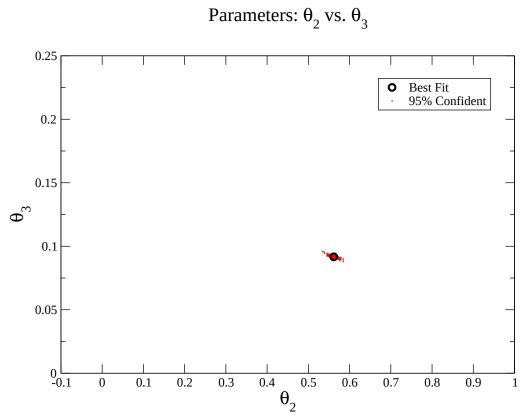

Figure 4: Cloud plot for θ_2 versus θ_3.

being determined, so there are 3 chromosomes in the genome), the `probabilityOfMutation` was set at 0.01, and the `probabilityOfImmigration` was set at 0.005. The dimensions of this problem are: $C = 1$, $N = 7$, $O = 7$, $P = 3$ and $R = 1$. The acquired fit and its parameters are displayed in Fig. 1.

The example run of App. A required 30 generations to reach theoretical convergence, given a population size of 507. Reported at specified generations are best, mean and worst values for the R^2 statistic. This is followed by a section that displays some of the vital statistics of the run, viz., the actaul and probable numbers of crossovers, mutations and immigrants, the parameter estimates with, in this case, their 95% confidence intervals, and the Gray codes held by these parameters at selected generations.

Parameter cloud plots for individuals whose parameters fall within the 95% confidence region are shown in Figs. 2–4. The ranges of these plots extend over the ranges given by the upper and lower bounds assigned to the parameters. Of the 15,181 individuals created to solve this problem, 9,398 fell within the 95% confidence region, i.e., 62% of all individuals were ‘viable’, which is a strong testiment to the efficacy of genetic algorithms in solving optimization problems.

4.2 Physical Fields

Physical fields are mathematical fields with units attached to them. In the set of modules contained in BEL, physical fields are provided for doing membrane analysis, so vectors and tensors associate with two-dimensional space. The acronym PF in the module names stands for physical field. Their software interfaces are provided in App. H.

4.2.1 Units

Module Bel.PF.Units exports type Si, which is an implementation for SI units. Also exported are numerous predefined constants that the programmer can use straightaway; they are: Acceleration, Ampere, Candela, Dimensionless, Force, Hertz, Joule, Kelvin, Kilogram, Meter, Mole, Momentum, Newton, Pascal, Power, Rate, Second, Strain, StrainRate, Stress, StressRate, Time, Velocity, Watt and Work.

From the perspective of the programmer, these predefined units are all that you will probably use most from this module. The other features exported by it are used internally by the various physical fields. If the units you happen to require are not predefined, then the easiest way to create them is to Parse them in, cf. source code for examples. *One point worth mentioning is that the units in your string need to appear in alphabetical order in both the numerator and the denominator to be able to be parsed in correctly.*

4.2.2 Scalars

A Bel.PF.Scalars.Scalar is the conglomeration of a Bel.MF.Numbers.Number and a Bel.PF.Units.Si. A Scalar has the same methods as a Number, except for the arguments are now Scalars instead of Numbers, plus it has four addition methods. Method GetUnits returns the Si units held by the scalar. Method SetUnits assigns an Si unit to the scalar. Method IsVoid returns *true* if the scalar is dimensionless; otherwise, it returns *false*. And method Reciprocal returns one divided by the scalar that it is bound to.

4.2.3 Scalar Functions

The math module Bel.PF.Functions exports the same math functions that the module Bel.MATH.Functions exports. The only difference is that here the functions accept and return instances of Bel.PF.Scalars.Scalar instead of instances of Bel.MF.Numbers.Number. Many, but not all, functions only accept dimensionless scalar arguments.

4.2.4 2D Vectors

Bel.PF.Vectors2 exports type Vector, which combines a Bel.MF.Arrays.Array of length two with a Bel.PF.Units.Si. A Vector has fewer methods than an Array. Retained are the methods: Get, Set, GetArray, SetArray, Equals and Dot. New are the methods: Parse, Typeset, GetUnits, SetUnits and IsVoid, whose behaviors are as one would expect. Module Vectors2 also exports the procedure Norm, which is the 2-norm of an array, and procedure UnitVector, which returns a normalized dimensionless vector.

4.2.5 2D Tensors

Bel.PF.Tensors2 exports type Tensor, which combines a 2×2 Bel.MF.Matrices.Matrix with a Bel.PF.Units.Si, and it supplies the dimensionless unit tensor I as a constant tensor field. Like a vector, a tensor has a reduced set of methods because of its smaller fixed size. Retained are the methods: Get, Set, GetMatrix, SetMatrix, Equals, Dot, DotTranspose, TransposeDot, Contract, TransposeContract, DoubleDot and TransposeDoubleDot. New are the methods: Parse, Typeset, TypesetRow, GetUnits, SetUnits, GetArray, SetArray, IsVoid, Transpose and Inverse.

Methods Parse, Typeset, GetArray and SetArray all apply to an array representation of the matrix components. Specifically, the components of some tensor

$$\mathbf{T} = \begin{bmatrix} \mathrm{T}_{11} & \mathrm{T}_{12} \\ \mathrm{T}_{21} & \mathrm{T}_{22} \end{bmatrix}$$

are represented as the array

$$\mathbf{T} = \{ \mathrm{T}_{11} \quad \mathrm{T}_{12} \quad \mathrm{T}_{21} \quad \mathrm{T}_{22} \}^T$$

in these methods. This idea is sometimes referred to as Voigt notation, except here there is no predisposition for matrix symmetry, as is usually the case with Voigt notations.

Module `Bel.PF.Tensors2` also exports a number of procedures that are of interest; they are `Norm`, which returns the Frobenius matrix norm, `FirstInvariant` and `Trace`, which are synonyms for the same operation, `SecondInvariant` and `Determinant`, which are equivalent operations for 2×2 matrices, `SymmetricPart`, which is $\frac{1}{2}(\mathbf{T} + \mathbf{T}^T)$, `SkewPart`, which is $\frac{1}{2}(\mathbf{T} - \mathbf{T}^T)$, and `Eigenvalues` and `SpectralDecomposition`. Given a tensor $\mathbf{T}$, `SpectralDecomposition` returns tensors $\mathbf{\Lambda}$ and $\mathbf{Q}$ such that $\mathbf{T} = \mathbf{Q} \cdot \mathbf{\Lambda} \cdot \mathbf{Q}^T$, where $\mathbf{\Lambda}$ is a diagonal tensor containing the eigenvalues of $\mathbf{T}$, and $\mathbf{Q} = [\{\boldsymbol{e}_1\}\{\boldsymbol{e}_2\}]$ is an orthogonal tensor containing the eigenvectors $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ of $\mathbf{T}$.

4.2.6 2D Fourth-Rank Tensors

`Bel.PF.QuadTensors2` exports six constant tensors that are instances of type Tensor, whose elements are a $2 \times 2 \times 2 \times 2$ matrix. These six constant fields are defined accordingly [16]

$$\begin{aligned}
\texttt{I} &= I_{ijk\ell} = \mathrm{I}_{ik}\mathrm{I}_{j\ell}, \\
\texttt{IBar} &= \bar{I}_{ijk\ell} = \mathrm{I}_{jk}\mathrm{I}_{i\ell}, \\
\texttt{S} &= S_{ijk\ell} = \tfrac{1}{2}\left(\mathrm{I}_{ik}\mathrm{I}_{j\ell} + \mathrm{I}_{jk}\mathrm{I}_{i\ell}\right), \\
\texttt{W} &= W_{ijk\ell} = \tfrac{1}{2}\left(\mathrm{I}_{ik}\mathrm{I}_{j\ell} - \mathrm{I}_{jk}\mathrm{I}_{i\ell}\right), \\
\texttt{One} &= \mathbb{1}_{ijk\ell} = \mathrm{I}_{ij}\mathrm{I}_{k\ell}, \\
\texttt{P} &= P_{ijk\ell} = \mathrm{I}_{ik}\mathrm{I}_{j\ell} - \tfrac{1}{3}\left(\mathrm{I}_{ij}\mathrm{I}_{k\ell}\right),
\end{aligned}$$

where $\mathbf{T} = \boldsymbol{I} : \mathbf{T}$, $\frac{1}{2}\left(\mathbf{T} + \mathbf{T}^T\right) = \boldsymbol{S} : \mathbf{T}$, $\mathbf{T}^T = \bar{\boldsymbol{I}} : \mathbf{T}$, $\frac{1}{2}\left(\mathbf{T} - \mathbf{T}^T\right) = \boldsymbol{W} : \mathbf{T}$, $\mathrm{tr}(\mathbf{T})\mathbf{I} = \mathbb{1} : \mathbf{T}$, and $\mathbf{T} - \frac{1}{3}\mathrm{tr}(\mathbf{T})\mathbf{I} = \boldsymbol{P} : \mathbf{T}$.

The methods exported by fourth-rank tensor fields are: `Get`, `Set`, `TypesetRow`, `GetUnits`, `SetUnits`, `GetMatrix`, `SetMatrix`, `IsVoid` and `Equals`. These behave as one would expect.

`TypesetRow`, `GetMatrix` and `SetMatrix` all refer to the elements of this $2 \times 2 \times 2 \times 2$ matrix in a condensed 4×4 format, whose components are assigned as follows

$$\boldsymbol{T} = \begin{bmatrix} T_{1111} & T_{1112} & T_{1121} & T_{1122} \\ T_{1211} & T_{1212} & T_{1221} & T_{1222} \\ T_{2111} & T_{2112} & T_{2121} & T_{2122} \\ T_{2211} & T_{2212} & T_{2221} & T_{2222} \end{bmatrix}.$$

It turns out that in this format the matrix contraction operators of `Dot`, `DotTranspose`, `TransposeDot`, `Contract` and `TransposeContract` apply for certain double contractions that commonly appear in continuum mechanics applications.

Method `Dot` is coded in Zonnon as `C := A.Dot(B)` for its math equivalent

$$\boldsymbol{C} = \boldsymbol{A} : \boldsymbol{B} \quad \text{or} \quad C_{ijk\ell} = A_{ijmn}B_{mnk\ell}$$

for fourth-rank tensors $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{C}$. Method `DotTranspose` is coded in Zonnon as `C := A.DotTranspose(B)` for its math equivalent

$$\boldsymbol{C} = \boldsymbol{A} : \boldsymbol{B}^T \quad \text{or} \quad C_{ijk\ell} = A_{ijmn}B_{k\ell mn}.$$

Method `TransposeDot` is coded in Zonnon as `C := A.TransposeDot(B)` for its math equivalent

$$\boldsymbol{C} = \boldsymbol{A}^T : \boldsymbol{B} \quad \text{or} \quad C_{ijk\ell} = A_{mnij}B_{mnk\ell}.$$

Method `Contract` is coded in Zonnon as `C := A.Contract(B)` for its math equivalent

$$\mathbf{C} = \boldsymbol{A} : \mathbf{B} \quad \text{or} \quad \mathrm{C}_{ij} = A_{ijk\ell}\mathrm{B}_{k\ell},$$

where $\mathbf{B}$ and $\mathbf{C}$ are now second-rank tensors. And method `TransposeContract` is coded in Zonnon as `C := A.TransposeContract(B)` for its math equivalent

$$\mathbf{C} = \boldsymbol{A}^T : \mathbf{B} \quad \text{or} \quad \mathrm{C}_{ij} = A_{k\ell ij}\mathrm{B}_{k\ell}.$$

Einstein's summation index notation has been used in the above formulæ.

Module `Bel.PF.QuadTensors2` also exports five procedures; they are: `Norm`, which returns the Frobenius norm of its matrix, `TensorProduct`, `SymTensorProduct`, `ODotProduct` and `SymODotProduct`. Method `TensorProduct` is coded in BEL as `T := TensorProduct(A, B)` for its math equivalent

$$\boldsymbol{T} = \mathbf{A} \otimes \mathbf{B} \quad \text{or} \quad T_{ijk\ell} = \mathrm{A}_{ij}\mathrm{B}_{k\ell},$$

where $\boldsymbol{T}$ is a fourth-rank tensor, while $\mathbf{A}$ and $\mathbf{B}$ are second-rank tensors. Method `SymTensorProduct` is coded in Zonnon as `T := SymTensorProduct(A, B)` for its math equivalent

$$\boldsymbol{T} = \tfrac{1}{2}\left(\mathbf{A} \otimes \mathbf{B} + \mathbf{B} \otimes \mathbf{A}\right)$$

or

$$\mathcal{T}_{ijk\ell} = \tfrac{1}{2}\left(\mathrm{A}_{ij}\mathrm{B}_{k\ell} + \mathrm{B}_{ij}\mathrm{A}_{k\ell}\right).$$

Method `ODotProduct` is coded in Zonnon as `T := ODotProduct(A, B)` for its math equivalent

$$\boldsymbol{\mathcal{T}} = \mathbf{A} \odot \mathbf{B}$$

or

$$\mathcal{T}_{ijk\ell} = \tfrac{1}{2}\left(\mathrm{A}_{ik}\mathrm{B}_{j\ell} + \mathrm{A}_{jk}\mathrm{B}_{i\ell}\right).$$

Method `SymODotProduct` is coded in Zonnon as `T := SymODotProduct(A, B)` for its math equivalent

$$\boldsymbol{\mathcal{T}} = \tfrac{1}{2}\left(\mathbf{A} \odot \mathbf{B} + \mathbf{B} \odot \mathbf{A}\right)$$

or

$$\mathcal{T}_{ijk\ell} = \tfrac{1}{4}\big(\mathrm{A}_{ik}\mathrm{B}_{j\ell} + \mathrm{A}_{jk}\mathrm{B}_{i\ell} + \mathrm{B}_{ik}\mathrm{A}_{j\ell} + \mathrm{B}_{jk}\mathrm{A}_{i\ell}\big).$$

These operators are common when constructing tangent moduli for many material models [16].

4.3 Membrane Kinematics & Kinetics

The physical fields that are exported as procedures from modules `Bel.BI.Kinematics` and `Bel.BI.Kinetics` have been used by the author [17] in his modeling of membranous tissues subject to in-plane biaxial deformations. These fundamental fields can be used in almost any constitutive equation that one might select for describing the mechanical response of tissues or other materials.

4.3.1 Kinematics

Module `Bel.BI.Kinematics` provides the various kinematic fields that describe an isochoric, in-plane, biaxial deformation of a membrane whose deformation gradient has components

$$\mathrm{F}_{ij} = \frac{\partial \mathrm{x}_i}{\partial \mathrm{X}_j} = \begin{bmatrix} \lambda_1 & \gamma_1 \\ \gamma_2 & \lambda_2 \end{bmatrix},$$

where the λ_i are the stretches, and the γ_i are the shears. Here the motion of a mass point originally located at position $\boldsymbol{X} = \{\, \mathrm{X}_1 \;\; \mathrm{X}_2 \,\}^T$ moves through space and time according to some continuous function of $\boldsymbol{x} = \boldsymbol{\chi}(\boldsymbol{X}, t) = \{\, \mathrm{x}_1 \;\; \mathrm{x}_2 \,\}^T$ (cf. textbook of Holzapfel [16]). The gradient of such a motion, given above, is the tensor field $\mathbf{F}$ known as the deformation gradient.

Assuming that one knows what the deformation gradient $\mathbf{F}$ is, then procedure `FInverse` returns $\mathbf{F}^{-1}$, which always exists (a consequence from the conservation of mass).

The left- and right-deformation tensors $\mathbf{B} = \mathbf{F} \cdot \mathbf{F}^T$ and $\mathbf{C} = \mathbf{F}^T \cdot \mathbf{F}$, respectively, are exported as procedures `B` and `C`, and their inverses by procedures `BInverse` and `CInverse`.

Procedure `R` returns the orthogonal rotation tensor $\mathbf{R}$ that arises from the polar decomposition of $\mathbf{F}$, i.e.,

$$\mathbf{F} = \mathbf{R} \cdot \mathbf{U} = \mathbf{V} \cdot \mathbf{R}.$$

where $\mathbf{V}$ and $\mathbf{U}$ are the left- and right-stretch tensors, respectively, and are given by procedures `V` and `U`. These are symmetric, positive-definite, tensor fields, so their inverses exist and are returned by procedures `VInverse` and `UInverse`.

Given these kinematic fields, it is possible to construct most strain tensors: $\frac{1}{2}(\mathbf{C} - \mathbf{I})$ is the Green-Lagrange strain tensor, $\frac{1}{2}(\mathbf{B} - \mathbf{I})$ is the Signorini strain tensor, $\frac{1}{2}(\mathbf{I} - \mathbf{B}^{-1})$ is the Euler-Almansi strain tensor, $\mathbf{U} - \mathbf{I}$ is the Biot strain tensor, and $\mathbf{V} - \mathbf{I}$ is the Bell-Ericksen strain tensor. Even Hencky strain $\ln \mathbf{U}$ can be computed with the aid of `SpectralDecomposition(U, Λ, Q)`, because $\ln \mathbf{U} = \mathbf{Q} \ln(\boldsymbol{\Lambda}) \mathbf{Q}^T$, wherein $\ln \boldsymbol{\Lambda}$ is given by the logarithm of its diagonal elements.

For rate formulations, the velocity gradient $\mathbf{L}$ replaces the deformation gradient $\mathbf{F}$ as the fundamental kinematic field. It has components $\mathrm{L}_{ij} = \partial \mathrm{v}_i / \partial \mathrm{x}_j$, where $\boldsymbol{v} = \dot{\boldsymbol{x}}$ is the velocity of particle $\boldsymbol{x}$. Procedure `L` returns the velocity gradient. The stretching $\mathbf{D}$ and spin $\mathbf{W}$ tensors provide the symmetric and skew-symmetric parts of the velocity gradient, respectively, which procedures `D` and `W` supply.

To integrate fields, we will need the incremental deformation gradient $\hat{\mathbf{F}}$. Other algorithms will require the incremental rotation $\hat{\mathbf{R}}$ that arises from the polar decomposition of $\hat{\mathbf{F}}$. These fields are supplied by procedures `HatF` and `HatR`.

Strain, as it has been derived [18] and used [17, 19] by this author, is established through an integral equation

$$\mathbf{E} = \mathbf{F} \cdot \int_0^t \left(\mathbf{F}^{-1} \cdot \mathbf{D} \cdot \mathbf{F}^{-T}\right)(\tau)\, \mathrm{d}\tau \cdot \mathbf{F}^T,$$

that is integrated in procedure `E` from time t_n over an increment in time `dTime` to time t_{n+1}. The time rate-of-change, i.e., the material derivative of strain $\mathbf{E}$, as described by

$$\dot{\mathbf{E}} = \mathbf{D} + \mathbf{L} \cdot \mathbf{E} + \mathbf{E} \cdot \mathbf{L}^T,$$

is returned by procedure `DE`. Here the stretching tensor $\mathbf{D}$ is synonymous with the Lie derivative for strain taken with respect to its particle velocity, i.e., $\mathcal{L}_{\boldsymbol{v}}(\mathbf{E}) = \mathbf{D}$, which, in effect, is how this author defines strain, viz., via its rate, as it appears in the physical laws for continua.

Procedures `HatF, HatR, E` and `DE` are acquired using algorithms found in Ref. [19].

4.3.2 Kinetics

Module `Bel.BI.Kinetics` provides mappings between three different descriptions for stress. They all require knowledge of the deformation gradient $\mathbf{F}$ and the stress field being mapped from. These are assumed to be known, provided by some boundary-value problem of interest.

Because the deformation is considered to be isochoric (volume preserving), which is a good assumption for soft biological tissues, there is no distinction between the symmetric stress tensors of Cauchy and Kirchhoff, denoted here as $\mathbf{T}$. This is often referred to as the true stress, and is the stress measure used in Eulerian analysis.

The other two stress tensors arise in Lagrangian formulations. The first Piola-Kirchhoff stress, also known as the Lagrangian or engineering stress, is not symmetric, and is given by $\mathbf{P} = \mathbf{T}\mathbf{F}^{-T}$. The second Piola-Kirchhoff stress, which is symmetric, is the more commonly used of the two, and is given by $\mathbf{S} = \mathbf{F}^{-1}\mathbf{T}\mathbf{F}^{-T}$, i.e., it is the Kirchhoff stress pulled back from the current configuration to reside within the reference configuration.

Procedures `PtoS` and `PtoT` map $\mathbf{P}$ into either $\mathbf{S}$ or $\mathbf{T}$, while procedures `StoP` and `StoT` map $\mathbf{S}$ into either $\mathbf{P}$ or $\mathbf{T}$, and procedures `TtoP` and `TtoS` map $\mathbf{T}$ into either $\mathbf{P}$ or $\mathbf{S}$, as their names so indicate.

Procedure `PtoTraction` maps the first Piola-Kirchhoff stress $\mathbf{P}$ into the traction vector $\boldsymbol{t}$ whereby

$$\mathrm{t}_1 = \mathrm{P}_{11} + \mathrm{P}_{12},$$
$$\mathrm{t}_2 = \mathrm{P}_{21} + \mathrm{P}_{22},$$

with t_1 and t_2 being the components of traction. The traction vector $\boldsymbol{t}$ has experimental components described by

$$\boldsymbol{t} = \left\{ \begin{array}{c} \mathrm{t}_1 \\ \mathrm{t}_2 \end{array} \right\} = \left\{ \begin{array}{c} \mathrm{f}_1/w\ell_2 \\ \mathrm{f}_2/w\ell_1 \end{array} \right\},$$

where f_1 and f_2 are the applied forces, ℓ_1 and ℓ_2 are the initial gage lengths that align with the 1 and 2 directions, respectively, and w is the width. Procedure `TractionToP` does the inverse mapping [17, 19], after which $\mathbf{T}$ and $\mathbf{S}$ can be gotten from procedures `PtoT` and `PtoS`.

4.4 Tissue Mechanics

There are numerous mathematical models for biological tissues that have been proposed over the years. An isochoric, isotropic, hypoelastic model is included in BEL, as it is the simplest model in its class.

4.4.1 Isotropic Hypoelasticity

Hypoelasticity is a continuum theory for finite elasticity that expresses stress rate as a function of strain rate, instead of the more common hyperelastic theory that expresses stress as a function of strain. The author [18, 20] has developed a hypoelastic theory that he considers to be suitable for modeling the passive response of isotropic soft tissues.

Module `Bel.TM.Hypoelastic.Isotropic` implements the 2D membrane version of this isotropic, hypoelastic, constitutive theory, whose general structure is given by the integral equation [19]

$$\mathbf{T} + \wp\,\mathbf{I} = \mathbf{F} \int_{t_0}^{t} \mathbf{F}^{-1} \frac{\partial \Psi(\mathbf{T},\mathbf{D})}{\partial\,\mathbf{D}} \mathbf{F}^{-T} \mathrm{d}\tau\ \mathbf{F}^{T},$$

whose rate form is hypoelastic

$$\dot{\mathbf{T}} + \dot{\wp}\,\mathbf{I} - 2\wp\mathbf{D} = \frac{\partial \Psi(\mathbf{T},\mathbf{D})}{\partial \mathbf{D}} + \mathbf{L}\mathbf{T} + \mathbf{T}\mathbf{L}^{T},$$

with $\partial\Psi/\partial\mathbf{D}$ being the Lie derivative of stress, viz., $\mathcal{L}_{\boldsymbol{v}}(\mathbf{T}) = \partial\Psi/\partial\mathbf{D}$ (recall that $\mathcal{L}_{\boldsymbol{v}}(\mathbf{E}) = \mathbf{D}$). Here $\wp$ is a Lagrange multiplier that forces an isochoric (constant volume) response.

From invariant theory, the potential function Ψ is taken to have form

$$\Psi(\mathbf{T},\mathbf{D}) = \mu\,\mathrm{tr}(\mathbf{D}\mathbf{D}) + \beta\,\mathrm{tr}(\mathbf{D}\mathbf{T}\mathbf{D}),$$

with μ being the shear modulus, and β being Fung's parameter. To be elastic, in the sense of Truesdell [21], Ψ must be quadratic in $\mathbf{D}$. This constitutive assumption produces the following material model

$$\dot{\mathbf{T}} + \dot{\wp}\,\mathbf{I} - 2\wp\mathbf{D} = 2\mu\mathbf{D} + \beta\big(\mathbf{D}\mathbf{T} + \mathbf{T}\mathbf{D}\big) + \mathbf{L}\mathbf{T} + \mathbf{T}\mathbf{L}^{T}.$$

From the isochoric constraint, and from the plane-stress assumption $\mathrm{T}_{33} = 0$ that membrane theory implies, one arrives at the ODE

$$\dot{\wp} = -2(\mu + \wp)\mathrm{tr}\,\mathbf{D},$$

which governs the Lagrange multiplier.

Procedure `Integrate` solves these two ODEs. The supplied arguments are: μ, β, $\mathrm{d}t = t_{n+1} - t_n$, $\wp_n$, $\mathbf{T}_n$, $\hat{\mathbf{F}}$, $\mathbf{L}_n$ and $\mathbf{L}_{n+1}$, while returning: $\wp_{n+1}$ and $\mathbf{T}_{n+1}$.

Module `Bel.TM.Hypoelastic.Isotropic` also provides a solver for the above isotropic, hypoelastic, constitutive equation. A call to the command `InitializeDataTree` prepares a `Bel.DATA.Tree` as the data repository for this integrator, `BuildDataTree` constructs the tree, and `Solve` fills in the data tree by calling procedure `Integrate`.

The `Bel.Object` type, `Datum`, is the data record held at each node of the exported `dataTree`. These nodes are keyed according to the following strategy: each experiment increments the key by 1000, with the initial condition be assigned to the thousands slot. The fields held by type `Datum` include: time t as `t`, the deformation gradient $\mathbf{F}$ as `F`, the velocity gradient $\mathbf{L}$ as `L`, the experimental first–Piola-Kirchhoff stress $\mathbf{P}$ as `P`, the Lagrange multiplier $\wp$ as `wp`, the strain $\mathbf{E}$ as `E`, and the Kirchhoff/Cauchy stress $\mathbf{T}$ as `T`.

Before calling `Solve`, an external procedure needs to fill in the `dataTree`, providing the experimental (i.e., control) variables: the time t, the deformation gradient $\mathbf{F}$, the velocity gradient $\mathbf{L}$, and the first Piola-Kirchhoff stress $\mathbf{P}$; plus initial conditions for the integrated (i.e., response) variables: strain $\mathbf{E}$ and stress $\mathbf{T}$, both of which are typically set to $\mathbf{0}$, plus the Lagrange multiplier $\wp$, which is also typically set to 0.

4.4.2 Example

The best way to learn how to apply this hypoelastic model to data is through an example (cf. the file `testHypoelastic.znn` in the test directory of the BEL software). An equi-biaxial experiment, like the example problem considered here, can be solved analytically yielding

$$\frac{1}{\lambda}\frac{\mathrm{d}\tau}{\mathrm{d}\lambda} \approx 6\mu + 2\beta\tau,$$

where λ is the stretch, and τ is the true stress. Consequently, the shear modulus μ is proportional to the intercept, and Fung's parameter β is proportional to the slope in an XY plot of τ versus $\lambda^{-1}\mathrm{d}\tau/\mathrm{d}\lambda$—a so-called Fung plot. This solution dismisses the presence of shears, which are measured and present in this experimental data set, so this solution is only an approximation of reality. Such a plot for the data provided is shown in Fig. 5,

Least Squares	parameter	lower	optimum	upper
linear	μ (kPa)	86.4	95.5	101.9
$R^2 = 0.840$	β	19.1	19.6	20.3
generalized	μ (kPa)	12.6	12.7	19.0
$R^2 = 0.988$	β	28.6	30.5	30.6
non-linear	μ (kPa)	6.44	6.45	7.99
$R^2 = 0.995$	β	36.1	37.3	37.4

Table 3: Parameters and their 95% confidence intervals for the isotropic, hypoelastic, material model when fit to equi-biaxial experimental data using different objective functions in the genetic algorithm.

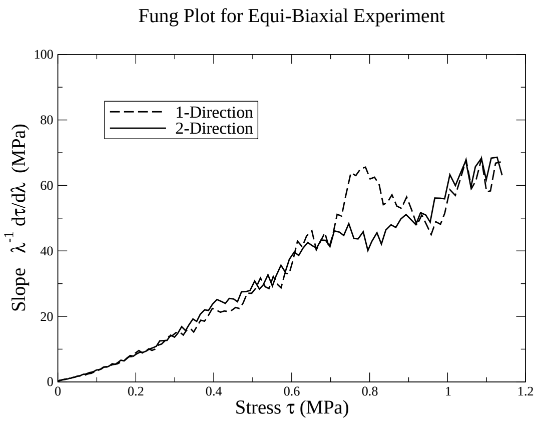

Figure 5: Fung plot for a data set from an equi-biaxial experiment.

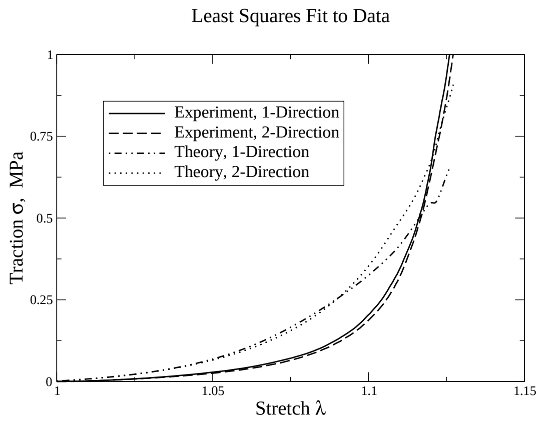

Figure 6: A GA fit of the hypoelastic model to soft tissue data using a linear, least-squares, objective function with $R^2 = 0.840$.

where one measures $\beta \approx 30$, while μ will be small relative to the scale of the ordinate axis.

Figures 6 & 7 show two seperate fits to the same experimental data using the same material model. They differ in their choice of objective function, clearly showing that one's selection of objective function can have a huge impact on the fitted parameters, whose values are displayed in Table 3. There is a discrepancy between model and experiment in the 1-direction. Its cause is likely due to noise observed in the experimental data stream of the Fung plot, which are the inputs to the material model.

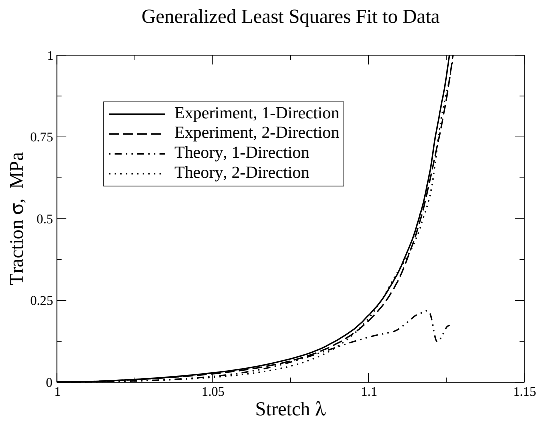

Figure 7: A GA fit of the hypoelastic model to soft tissue data using a generalized, least-squares, objective function with $R^2 = 0.988$.

Acknowledgment

The speed by which both Eugene Zueff and Roman Mitin have fixed compiler bugs uncovered during the writing of BEL has made this a fun project for me to be involved with. I was able create something useful for myself, and to also help the team

of compiler developers at ETH Zurich debug their product during its development. After implementing math extensions into the Zonnon compiler (Sec. 10 in [1]), Nina Gonova then incorporated these enhancements into BEL, for which I am most grateful.

References

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

[2] Wirth, N., 1986, Algorithms + Data Structures = Programs, 2nd ed., Prentic-Hall.

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

[4] 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.

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

[6] Goldberg, D. E., 1989, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, Boston.

[7] Goldberg, D. E., 2002, The Design of Innovation: Lessons learned from and for Competent Genetic Algorithms, vol. 7 of Genetic Algorithms and Evolutionary Computation, Kluwer, Boston.

[8] Sivanandam, S. N. and Deepa, S. N., 2008, Introduction to Genetic Algorithms, Springer, Berlin.

[9] Buse, A., 1973, "Goodness of fit in generalized least squares estimation," American Statistician, **27**, pp. 106–108.

[10] Doehring, T. C., Carew, E. O., and Vesely, I., 2004, "The effect of strain rate on the viscoelastic response of aortic valve tissues: A direct-fit approach," Annals of Biomedical Engineering, **32**, pp. 223–232.

[11] Bard, Y., 1974, Nonlinear Parameter Estimation, Academic Press, New York.

[12] Michalewicz, Z., 1996, Genetic Algorithms + Data Structures = Evolution Programs, 3rd ed., Springer, Berlin.

[13] Yao, L. and Sethares, W. A., 1994, "Nonlinear parameter estimation via the genetic algorithm," IEEE Transactions on Signal Processing, **42**, pp. 927–935.

[14] Draper, N. R. and Smith, H., 1998, Applied Regression Analysis, 3rd ed., Wiley Series in Probability and Statistics, Wiley, New York.

[15] Walter, E. and Pronzato, L., 1994, Identification de modèles paramétriques à partir de données expérimentales, Masson, Paris.

[16] Holzapfel, G. A., 2000, Nonlinear Solid Mechanics: A continuum approach for engineering, Wiley, Chichester.

[17] Freed, A. D., Einstein, D. R., and Sacks, M. S., 2010, "Hypoelastic Soft Tissues, Part II: in-plane biaxial experiments," Submitted: Acta Mechanica.

[18] Freed, A. D., 2010, "Hypoelastic Soft Tissues, Part I: theory," In Press: Acta Mechanica.

[19] Freed, A. D., Einstein, D. R., and Sacks, M. S., 2010, "Hypoelastic Soft Tissue Composites: An application to bovine pericardium," In preparation for: Biomechanics and Modeling in Mechanobiology.

[20] Freed, A. D., 2008, "Anisotropy in hypoelastic soft--tissue mechanics, I: theory," Journal of Mechanics of Materials and Structures, **3** (5), pp. 911–928.

[21] Truesdell, C., 1955, "Hypoelasticity," Journal of Rational Mechanics and Analysis, **4**, pp. 83–133.

A Example GA Output

```
--------------------------------------------------------------
  R-squared statistic was based on linear least squares
generation  :  max fitness   :   mean fitness  :  min fitness
--------------------------------------------------------------
       1       9.99043E-001      7.58534E-001     3.58992E-003
       2       9.99114E-001      8.97776E-001     1.06065E-001
       3       9.99120E-001      9.38629E-001     1.44723E-001
       5       9.99211E-001      9.83981E-001     6.48252E-001
       7       9.99213E-001      9.87075E-001     7.23040E-001
      10       9.99216E-001      9.93831E-001     8.27895E-001
      13       9.99217E-001      9.93343E-001     5.52911E-001
      16       9.99217E-001      9.91229E-001     2.10834E-001
      20       9.99217E-001      9.95235E-001     8.32049E-001
      25       9.99217E-001      9.91022E-001     5.35719E-001
      30       9.99217E-001      9.91906E-001     4.46091E-001
--------------------------------------------------------------
total number of genes  = 42
size of the population = 507
number of contestants  = 4
number of generations  = 30
number of individuals  = 15181
number that are viable = 9398
number of crossovers : actual   = 5468
    P = 7.50E-001    : probable = 5502
number of mutations  : acutal   = 6180
    P = 1.00E-002    : probable = 6175
number of immigrants : actual   = 77
    P = 5.00E-003    : probable = 73
--------------------------------------------------------------
95% confidence intervals for all of the parameters are:
    1)   4.193E-002  <  6.428E-002  <  8.708E-002
    2)   5.319E-001  <  5.620E-001  <  5.770E-001
    3)   8.970E-002  <  9.167E-002  <  9.507E-002
--------------------------------------------------------------
gen : par)  -- genotype --
    1 : 1)  10001111100010
        2)  11011100000000
        3)  01010000000101
   30 : 1)  10111011011001
        2)  11010111000010
        3)  01110011001100
--------------------------------------------------------------
```

B Definitions

```
definition {value} Bel.Object;
   import
      System.IO.BinaryReader as BinaryReader;
      System.IO.BinaryWriter as BinaryWriter;
   procedure Initialize;
   procedure Nullify;
   procedure Clone () :object{Object};
   procedure Load (br : BinaryReader);
   procedure Store (bw : BinaryWriter);
end Object.

definition {value} Bel.Field refines Bel.Object;
   procedure Negative () : object{Field};
   procedure Add (f : object{Field}) : object{Field};
   procedure Subtract (f : object{Field}) : object{Field};
   procedure Multiply (f : object{Field}) : object{Field};
   procedure Divide (f : object{Field}) : object{Field};
end Field.

definition {value} Bel.EvaluateSeries;
   import
      Bel.MF.Numbers as N;
   var
      eos : boolean;
   procedure GetCoef (n : integer; x : N.Number) : N.Number;
end EvaluateSeries.

definition {value} Bel.Gene;
   import
      Bel.MF.Numbers as N;
   procedure Initialize;
   procedure Clone () : object{Gene};
   procedure IsEqualTo (g : object{Gene}) : boolean;
   procedure Parse (allele : char);
   procedure Typeset () : char;
   procedure Mutate (mutationProbability : N.Number;
                     var numberOfMutations : integer);
```

```
end Gene.

definition {ref} Bel.Chromosome;
   import
      Bel.MF.Numbers as N;
   procedure Initialize (minParameter, maxParameter : N.Number;
                         numberOfSignificantFigures : integer);
   procedure Clone () : object{Chromosome};
   procedure IsEqualTo (g : object{Chromosome}) : boolean;
   procedure Length () : integer;
   procedure Parse (genotype : string);
   procedure Typeset () : string;
   procedure Mutate (mutationProbability : N.Number;
                         var numberOfMutations : integer);
   procedure Encode (phenotype : N.Number);
   procedure Decode () : N.Number;
end Chromosome.
```

C IO Modules

```
module Bel.IO.Log;
   procedure Open (directory : string);
   procedure Close;
   procedure Message (message : string);
   procedure WarningMessage (inputErrorCode, outputErrorCode : integer;
                         originOfError : string);
   procedure ErrorMessage (inputErrorCode, outputErrorCode : integer;
                         originOfError : string);
end Log.

module Bel.IO.DataFiles;
   import
      System.IO.BinaryReader as BinaryReader,
      System.IO.BinaryWriter as BinaryWriter,
   procedure FileExists (fileName : string) : boolean;
   procedure OpenReader (fileName : string) : BinaryReader;
   procedure OpenWriter (fileName : string) : BinaryWriter;
   procedure CloseReader (br : BinaryReader);
   procedure CloseWriter (bw : BinaryWriter);
```

```
end DataFiles.

module Bel.IO.TextFiles;
   import
      System.IO.StreamReader as StreamReader,
      System.IO.StreamWriter as StreamWriter,
   procedure FileExists (fileName : string) : boolean;
   procedure OpenReader (fileName : string) : StreamReader;
   procedure OpenWriter (fileName : string) : StreamWriter;
   procedure CloseReader (sr : StreamReader);
   procedure CloseWriter (sw : StreamWriter);
end TextFiles.
```

D Data Modules

The procedures inherited from the *Bel.Object* definition interface have been implemented in the actual code, but are not repeated here in their export interfaces in an effort to keep them concise.

```
object Bel.DATA.Queue implements Bel.Object;
   procedure Configure (dataClone : object{Object});
   procedure Length () : integer;
   procedure Pop () : object{Object};
   procedure Push (datum : object{Object});
end Queue.

object Bel.DATA.Stack implements Bel.Object;
   procedure Configure (dataClone : object{Object});
   procedure Length () : integer;
   procedure Pop () : object{Object};
   procedure Push (datum : object{Object});
end Stack.

module Bel.DATA.Keys;
   import
      Bel.Object as Object;
   type KeyType = integer{64};
   type Key = object implements Object
      procedure Get () : KeyType;
      procedure Set (k : KeyType);
      procedure Parse (s : string);
      procedure Typeset () : string;
      procedure Equals (k : Key) : boolean;
      procedure LessThan (k : Key) : boolean;
      procedure GreaterThan (k : Key) : boolean;
   end Key;
   operator “:=” (var l : Key; r : Key);
   operator “:=” (var l : Key; r : integer{64});
   operator “:=” (var l : Key; r : integer{32});
   operator “:=” (var l : Key; r : integer{16});
   operator “:=” (var l : Key; r : integer{8});
   operator “:=” (var l : Key; r : cardinal{64});
   operator “:=” (var l : Key; r : cardinal{32});
   operator “:=” (var l : Key; r : cardinal{16});
   operator “:=” (var l : Key; r : cardinal{8});
   operator "=" (l, r : Key) : boolean;
   operator "#" (l, r : Key) : boolean;
   operator "<" (l, r : Key) : boolean;
   operator "<=" (l, r : Key) : boolean;
   operator ">" (l, r : Key) : boolean;
   operator ">=" (l, r : Key) : boolean;
end Keys.

object Bel.DATA.List implements Bel.Object;
   import
      Bel.DATA.Keys as K;
   procedure Configure (dataClone : object{Object});
   procedure Length () : integer;
   procedure Delete (key : K.Key; var success : boolean);
   procedure Insert (datum : object{Object}; key : K.Key
                     var success : boolean);
   procedure Find (key : K.Key; var found : boolean);
   procedure Home;
   procedure Next (var moved : boolean);
   procedure Previous (var moved : boolean);
   procedure GetData () : object{Object};
   procedure GetKey () : K.Key;
   procedure Update (datum : object{Object}; key : K.Key;
                     var success : boolean);
```

```
end List.

object Bel.DATA.Tree implements Bel.Object;
   import
      Bel.DATA.Keys as K;
   procedure Configure (dataClone : object{Object});
   procedure Entries () : integer;
   procedure Height () : integer;
   procedure Delete (key : K.Key; var success : boolean);
   procedure Insert (datum : object{Object}; key : K.Key;
                     var success : boolean);
   procedure Find (key : K.Key; var found : boolean);
   procedure Home;
   procedure Left (var moved : boolean);
   procedure Right (var moved : boolean);
   procedure Previous;
   procedure GetData () : object{Object};
   procedure GetKey () : K.Key;
   procedure Update (datum : object{Object}; key : K.Key;
                     var success : boolean);
end Tree.
```

E Mathematical Fields

The procedures inherited from the *Bel.Object* and *Bel.Field* definition interfaces have been implemented in the actual code, but are not repeated here to help keep the export interfaces more concise.

```
module Bel.MF.Numbers;
   import
      Bel.Field as Field;
   var{immutable}
      Epsilon, MaximumPositiveNumber, MinimumPositiveNumber, NaN,
         NegativeInfinity, PositiveInfinity : Number;
   type
      NumberType = real{64};
   type Number = object implements Field
      procedure Parse (input : string);
      procedure Typeset () : string;
      procedure ToString (significantDigits : integer) : string;
      procedure Get () : NumberType;
      procedure Set (r : NumberType);
      procedure IsFinite () : boolean;
      procedure IsInfinite () : boolean;
      procedure IsPositiveInfinity () : boolean;
      procedure IsNegativeInfinity () : boolean;
      procedure IsNaN () : boolean;
      procedure Equals (n : Number) : boolean;
      procedure NotEqual (n : Number) : boolean;
      procedure LessThan (n : Number) : boolean;
      procedure LessThanOrEqual (n : Number) : boolean;
      procedure GreaterThan (n : Number) : boolean;
      procedure GreaterThanOrEqual (n : Number) : boolean;
      procedure Magnitude () : Number;
      procedure Power (exponent : Number) : Number;
   end Number;
   operator ":=" (var l : Number; r : Number);
   operator ":=" (var l : Number; r : real{64});
   operator ":=" (var l : Number; r : real{32});
   operator ":=" (var l : Number; r : integer{64});
   operator ":=" (var l : Number; r : integer{32});
   operator ":=" (var l : Number; r : integer{16});
   operator ":=" (var l : Number; r : integer{8});
   operator ":=" (var l : Number; r : cardinal{64});
   operator ":=" (var l : Number; r : cardinal{32});
   operator ":=" (var l : Number; r : cardinal{16});
   operator ":=" (var l : Number; r : cardinal{8});
   operator ":=" (var l : Number; r : string);
   operator "-" (x : Number) : Number;
   operator "=" (l, r : Number) : boolean;
   operator "=" (l : Number; r : real{64}) : boolean;
   operator "=" (l : Number; r : real{32}) : boolean;
   operator "=" (l : Number; r : integer{64}) : boolean;
   operator "=" (l : Number; r : integer{32}) : boolean;
   operator "=" (l : Number; r : integer{16}) : boolean;
   operator "=" (l : Number; r : integer{8}) : boolean;
   operator "=" (l : Number; r : cardinal{64}) : boolean;
   operator "=" (l : Number; r : cardinal{32}) : boolean;
   operator "=" (l : Number; r : cardinal{16}) : boolean;
   operator "=" (l : Number; r : cardinal{8}) : boolean;
   operator "=" (l : Number; r : string) : boolean;
   operator "=" (l : real{64}; r : Number) : boolean;
   operator "=" (l : real{32}; r : Number) : boolean;
   operator "=" (l : integer{64}; r : Number) : boolean;
   operator "=" (l : integer{32}; r : Number) : boolean;
   operator "=" (l : integer{16}; r : Number) : boolean;
   operator "=" (l : integer{8}; r : Number) : boolean;
   operator "=" (l : cardinal{64}; r : Number) : boolean;
   operator "=" (l : cardinal{32}; r : Number) : boolean;
   operator "=" (l : cardinal{16}; r : Number) : boolean;
   operator "=" (l : cardinal{8}; r : Number) : boolean;
   operator "=" (l : string; r : Number) : boolean;
   operator "#" (l, r : Number) : boolean;
   operator "#" (l : Number; r : real{64}) : boolean;
   operator "#" (l : Number; r : real{32}) : boolean;
   operator "#" (l : Number; r : integer{64}) : boolean;
   operator "#" (l : Number; r : integer{32}) : boolean;
   operator "#" (l : Number; r : integer{16}) : boolean;
   operator "#" (l : Number; r : integer{8}) : boolean;
   operator "#" (l : Number; r : cardinal{64}) : boolean;
   operator "#" (l : Number; r : cardinal{32}) : boolean;
   operator "#" (l : Number; r : cardinal{16}) : boolean;
   operator "#" (l : Number; r : cardinal{8}) : boolean;
   operator "#" (l : Number; r : string) : boolean;
   operator "#" (l : real{64}; r : Number) : boolean;
   operator "#" (l : real{32}; r : Number) : boolean;
   operator "#" (l : integer{64}; r : Number) : boolean;
   operator "#" (l : integer{32}; r : Number) : boolean;
   operator "#" (l : integer{16}; r : Number) : boolean;
   operator "#" (l : integer{8}; r : Number) : boolean;
   operator "#" (l : cardinal{64}; r : Number) : boolean;
   operator "#" (l : cardinal{32}; r : Number) : boolean;
   operator "#" (l : cardinal{16}; r : Number) : boolean;
   operator "#" (l : cardinal{8}; r : Number) : boolean;
   operator "#" (l : string; r : Number) : boolean;
   operator "<" (l, r : Number) : boolean;
   operator "<" (l : Number; r : real{64}) : boolean;
   operator "<" (l : Number; r : real{32}) : boolean;
   operator "<" (l : Number; r : integer{64}) : boolean;
   operator "<" (l : Number; r : integer{32}) : boolean;
   operator "<" (l : Number; r : integer{16}) : boolean;
   operator "<" (l : Number; r : integer{8}) : boolean;
   operator "<" (l : Number; r : cardinal{64}) : boolean;
   operator "<" (l : Number; r : cardinal{32}) : boolean;
   operator "<" (l : Number; r : cardinal{16}) : boolean;
   operator "<" (l : Number; r : cardinal{8}) : boolean;
   operator "<" (l : Number; r : string) : boolean;
   operator "<" (l : real{64}; r : Number) : boolean;
   operator "<" (l : real{32}; r : Number) : boolean;
   operator "<" (l : integer{64}; r : Number) : boolean;
   operator "<" (l : integer{32}; r : Number) : boolean;
   operator "<" (l : integer{16}; r : Number) : boolean;
   operator "<" (l : integer{8}; r : Number) : boolean;
   operator "<" (l : cardinal{64}; r : Number) : boolean;
   operator "<" (l : cardinal{32}; r : Number) : boolean;
   operator "<" (l : cardinal{16}; r : Number) : boolean;
   operator "<" (l : cardinal{8}; r : Number) : boolean;
   operator "<" (l : string; r : Number) : boolean;
   operator "<=" (l, r : Number) : boolean;
   operator "<=" (l : Number; r : real{64}) : boolean;
   operator "<=" (l : Number; r : real{32}) : boolean;
   operator "<=" (l : Number; r : integer{64}) : boolean;
   operator "<=" (l : Number; r : integer{32}) : boolean;
   operator "<=" (l : Number; r : integer{16}) : boolean;
   operator "<=" (l : Number; r : integer{8}) : boolean;
   operator "<=" (l : Number; r : cardinal{64}) : boolean;
   operator "<=" (l : Number; r : cardinal{32}) : boolean;
   operator "<=" (l : Number; r : cardinal{16}) : boolean;
   operator "<=" (l : Number; r : cardinal{8}) : boolean;
   operator "<=" (l : Number; r : string) : boolean;
   operator "<=" (l : real{64}; r : Number) : boolean;
   operator “<=” (l : real{32}; r : Number) : boolean;
   operator “<=” (l : integer{64}; r : Number) : boolean;
   operator “<=” (l : integer{32}; r : Number) : boolean;
   operator “<=” (l : integer{16}; r : Number) : boolean;
   operator “<=” (l : integer{8}; r : Number) : boolean;
   operator “<=” (l : cardinal{64}; r : Number) : boolean;
   operator “<=” (l : cardinal{32}; r : Number) : boolean;
   operator “<=” (l : cardinal{16}; r : Number) : boolean;
   operator “<=” (l : cardinal{8}; r : Number) : boolean;
   operator “<=” (l : string; r : Number) : boolean;
   operator “>” (l, r : Number) : boolean;
   operator “>” (l : Number; r : real{64}) : boolean;
   operator “>” (l : Number; r : real{32}) : boolean;
   operator “>” (l : Number; r : integer{64}) : boolean;
   operator “>” (l : Number; r : integer{32}) : boolean;
   operator “>” (l : Number; r : integer{16}) : boolean;
   operator “>” (l : Number; r : integer{8}) : boolean;
   operator “>” (l : Number; r : cardinal{64}) : boolean;
   operator “>” (l : Number; r : cardinal{32}) : boolean;
   operator “>” (l : Number; r : cardinal{16}) : boolean;
   operator “>” (l : Number; r : cardinal{8}) : boolean;
   operator “>” (l : Number; r : string) : boolean;
   operator “>” (l : real{64}; r : Number) : boolean;
   operator “>” (l : real{32}; r : Number) : boolean;
   operator “>” (l : integer{64}; r : Number) : boolean;
   operator “>” (l : integer{32}; r : Number) : boolean;
   operator “>” (l : integer{16}; r : Number) : boolean;
   operator “>” (l : integer{8}; r : Number) : boolean;
   operator “>” (l : cardinal{64}; r : Number) : boolean;
   operator “>” (l : cardinal{32}; r : Number) : boolean;
   operator “>” (l : cardinal{16}; r : Number) : boolean;
   operator “>” (l : cardinal{8}; r : Number) : boolean;
   operator “>” (l : string; r : Number) : boolean;
   operator “>=” (l, r : Number) : boolean;
   operator “>=” (l : Number; r : real{64}) : boolean;
   operator “>=” (l : Number; r : real{32}) : boolean;
   operator “>=” (l : Number; r : integer{64}) : boolean;
   operator “>=” (l : Number; r : integer{32}) : boolean;
   operator “>=” (l : Number; r : integer{16}) : boolean;
   operator “>=” (l : Number; r : integer{8}) : boolean;
   operator “>=” (l : Number; r : cardinal{64}) : boolean;
   operator “>=” (l : Number; r : cardinal{32}) : boolean;
   operator “>=” (l : Number; r : cardinal{16}) : boolean;
   operator “>=” (l : Number; r : cardinal{8}) : boolean;
   operator “>=” (l : Number; r : string) : boolean;
   operator “>=” (l : real{64}; r : Number) : boolean;
   operator “>=” (l : real{32}; r : Number) : boolean;
   operator “>=” (l : integer{64}; r : Number) : boolean;
   operator “>=” (l : integer{32}; r : Number) : boolean;
   operator “>=” (l : integer{16}; r : Number) : boolean;
   operator “>=” (l : integer{8}; r : Number) : boolean;
   operator “>=” (l : cardinal{64}; r : Number) : boolean;
   operator “>=” (l : cardinal{32}; r : Number) : boolean;
   operator “>=” (l : cardinal{16}; r : Number) : boolean;
   operator “>=” (l : cardinal{8}; r : Number) : boolean;
   operator “>=” (l : string; r : Number) : boolean;
   operator “+” (l, r : Number) : Number;
   operator “+” (l : Number; r : real{64}) : Number;
   operator “+” (l : Number; r : real{32}) : Number;
   operator “+” (l : Number; r : integer{64}) : Number;
   operator “+” (l : Number; r : integer{32}) : Number;
   operator “+” (l : Number; r : integer{16}) : Number;
   operator “+” (l : Number; r : integer{8}) : Number;
   operator “+” (l : Number; r : cardinal{64}) : Number;
   operator “+” (l : Number; r : cardinal{32}) : Number;
   operator “+” (l : Number; r : cardinal{16}) : Number;
   operator “+” (l : Number; r : cardinal{8}) : Number;
   operator “+” (l : Number; r : string) : Number;
   operator “+” (l : real{64}; r : Number) : Number;
   operator “+” (l : real{32}; r : Number) : Number;
   operator “+” (l : integer{64}; r : Number) : Number;
   operator “+” (l : integer{32}; r : Number) : Number;
   operator “+” (l : integer{16}; r : Number) : Number;
   operator “+” (l : integer{8}; r : Number) : Number;
   operator "+" (l : cardinal{64}; r : Number) : Number;
   operator "+" (l : cardinal{32}; r : Number) : Number;
   operator "+" (l : cardinal{16}; r : Number) : Number;
   operator "+" (l : cardinal{8}; r : Number) : Number;
   operator "+" (l : string; r : Number) : Number;
   operator "-" (l, r : Number) : Number;
   operator "-" (l : Number; r : real{64}) : Number;
   operator "-" (l : Number; r : real{32}) : Number;
   operator "-" (l : Number; r : integer{64}) : Number;
   operator "-" (l : Number; r : integer{32}) : Number;
   operator "-" (l : Number; r : integer{16}) : Number;
   operator "-" (l : Number; r : integer{8}) : Number;
   operator "-" (l : Number; r : cardinal{64}) : Number;
   operator "-" (l : Number; r : cardinal{32}) : Number;
   operator "-" (l : Number; r : cardinal{16}) : Number;
   operator "-" (l : Number; r : cardinal{8}) : Number;
   operator "-" (l : Number; r : string) : Number;
   operator "-" (l : real{64}; r : Number) : Number;
   operator "-" (l : real{32}; r : Number) : Number;
   operator "-" (l : integer{64}; r : Number) : Number;
   operator "-" (l : integer{32}; r : Number) : Number;
   operator "-" (l : integer{16}; r : Number) : Number;
   operator "-" (l : integer{8}; r : Number) : Number;
   operator "-" (l : cardinal{64}; r : Number) : Number;
   operator "-" (l : cardinal{32}; r : Number) : Number;
   operator "-" (l : cardinal{16}; r : Number) : Number;
   operator "-" (l : cardinal{8}; r : Number) : Number;
   operator "-" (l : string; r : Number) : Number;
   operator "*" (l, r : Number) : Number;
   operator "*" (l : Number; r : real{64}) : Number;
   operator "*" (l : Number; r : real{32}) : Number;
   operator "*" (l : Number; r : integer{64}) : Number;
   operator "*" (l : Number; r : integer{32}) : Number;
   operator "*" (l : Number; r : integer{16}) : Number;
   operator "*" (l : Number; r : integer{8}) : Number;
   operator "*" (l : Number; r : cardinal{64}) : Number;
   operator "*" (l : Number; r : cardinal{32}) : Number;
   operator “*” (l : Number; r : cardinal{16}) : Number;
   operator “*” (l : Number; r : cardinal{8}) : Number;
   operator “*” (l : Number; r : string) : Number;
   operator “*” (l : real{64}; r : Number) : Number;
   operator “*” (l : real{32}; r : Number) : Number;
   operator “*” (l : integer{64}; r : Number) : Number;
   operator “*” (l : integer{32}; r : Number) : Number;
   operator “*” (l : integer{16}; r : Number) : Number;
   operator “*” (l : integer{8}; r : Number) : Number;
   operator “*” (l : cardinal{64}; r : Number) : Number;
   operator “*” (l : cardinal{32}; r : Number) : Number;
   operator “*” (l : cardinal{16}; r : Number) : Number;
   operator “*” (l : cardinal{8}; r : Number) : Number;
   operator “*” (l : string; r : Number) : Number;
   operator “/” (l, r : Number) : Number;
   operator “/” (l : Number; r : real{64}) : Number;
   operator “/” (l : Number; r : real{32}) : Number;
   operator “/” (l : Number; r : integer{64}) : Number;
   operator “/” (l : Number; r : integer{32}) : Number;
   operator “/” (l : Number; r : integer{16}) : Number;
   operator “/” (l : Number; r : integer{8}) : Number;
   operator “/” (l : Number; r : cardinal{64}) : Number;
   operator “/” (l : Number; r : cardinal{32}) : Number;
   operator “/” (l : Number; r : cardinal{16}) : Number;
   operator “/” (l : Number; r : cardinal{8}) : Number;
   operator “/” (l : Number; r : string) : Number;
   operator “/” (l : real{64}; r : Number) : Number;
   operator “/” (l : real{32}; r : Number) : Number;
   operator “/” (l : integer{64}; r : Number) : Number;
   operator “/” (l : integer{32}; r : Number) : Number;
   operator “/” (l : integer{16}; r : Number) : Number;
   operator “/” (l : integer{8}; r : Number) : Number;
   operator “/” (l : cardinal{64}; r : Number) : Number;
   operator “/” (l : cardinal{32}; r : Number) : Number;
   operator “/” (l : cardinal{16}; r : Number) : Number;
   operator “/” (l : cardinal{8}; r : Number) : Number;
   operator “/” (l : string; r : Number) : Number;
```

```
   operator “**” (l, r : Number) : Number;
   operator “**” (l : Number; r : real{64}) : Number;
   operator “**” (l : Number; r : real{32}) : Number;
   operator “**” (l : Number; r : integer{64}) : Number;
   operator “**” (l : Number; r : integer{32}) : Number;
   operator “**” (l : Number; r : integer{16}) : Number;
   operator “**” (l : Number; r : integer{8}) : Number;
   operator “**” (l : Number; r : cardinal{64}) : Number;
   operator “**” (l : Number; r : cardinal{32}) : Number;
   operator “**” (l : Number; r : cardinal{16}) : Number;
   operator “**” (l : Number; r : cardinal{8}) : Number;
   operator “**” (l : Number; r : string) : Number;
   operator “**” (l : real{64}; r : Number) : Number;
   operator “**” (l : real{32}; r : Number) : Number;
   operator “**” (l : integer{64}; r : Number) : Number;
   operator “**” (l : integer{32}; r : Number) : Number;
   operator “**” (l : integer{16}; r : Number) : Number;
   operator “**” (l : integer{8}; r : Number) : Number;
   operator “**” (l : cardinal{64}; r : Number) : Number;
   operator “**” (l : cardinal{32}; r : Number) : Number;
   operator “**” (l : cardinal{16}; r : Number) : Number;
   operator “**” (l : cardinal{8}; r : Number) : Number;
   operator “**” (l : string; r : Number) : Number;
end Numbers.

module Bel.MF.Arrays;
   import
      Bel.MF.Numbers as N,
      Bel.Field as Field;
   type
      ArrayOfNumber = array * of N.Number;
      ArrayOfReal64 = array * of real{64};
      ArrayOfReal32 = array * of real{32};
      ArrayOfInt64 = array * of integer{64};
      ArrayOfInt32 = array * of integer{32};
      ArrayOfInt16 = array * of integer{16};
      ArrayOfInt8 = array * of integer{8};
      ArrayOfCard64 = array * of cardinal{64};
      ArrayOfCard32 = array * of cardinal{32};
      ArrayOfCard16 = array * of cardinal{16};
      ArrayOfCard8 = array * of cardinal{8};
      ArrayOfString = array * of string;
      MathArray = array {math} * of N.Number;
   type Array = object implements [], Field
      procedure Get (row : integer) : N.Number implements [].Get;
      procedure Set (row : integer; x : N.Number) implements [].Set;
      procedure Create (length : integer);
      procedure Length () : integer;
      procedure GetSubArray (startAt, endAt : integer) : Array;
      procedure SetSubArray (startAt : integer; a : Array);
      procedure GetArray () : ArrayOfNumber;
      procedure GetMathArray () : MathArray;
      procedure SetArray (a : Array);
      procedure SetMathArray (a : MathArray);
      procedure SetArrayOfNumber (a : ArrayOfNumber);
      procedure SetArrayOfReal64 (a : ArrayOfReal64);
      procedure SetArrayOfReal32 (a : ArrayOfReal32);
      procedure SetArrayOfInt64 (a : ArrayOfInt64);
      procedure SetArrayOfInt32 (a : ArrayOfInt32);
      procedure SetArrayOfInt16 (a : ArrayOfInt16);
      procedure SetArrayOfInt8 (a : ArrayOfInt8);
      procedure SetArrayOfCard64 (a : ArrayOfCard64);
      procedure SetArrayOfCard32 (a : ArrayOfCard32);
      procedure SetArrayOfCard16 (a : ArrayOfCard16);
      procedure SetArrayOfCard8 (a : ArrayOfCard8);
      procedure SetArrayOfString (a : ArrayOfString);
      procedure Equals (a : Array) : boolean;
      procedure IsANumber() : boolean;
      procedure Dot (a : Array) : N.Number;
      procedure Normalize (var scaleFactor : N.Number);
      procedure Sort;
      procedure Swap (row1, row2 : integer);
   end Array;
   operator ":=" (var l : Array; r : Array);
   operator ":=" (var l : Array; r : ArrayOfNumber);
   operator ":=" (var l : Array; r : ArrayOfReal64);
   operator ":=" (var l : Array; r : ArrayOfReal32);
   operator ":=" (var l : Array; r : ArrayOfInt64);
   operator ":=" (var l : Array; r : ArrayOfInt32);
   operator ":=" (var l : Array; r : ArrayOfInt16);
   operator ":=" (var l : Array; r : ArrayOfInt8);
   operator ":=" (var l : Array; r : ArrayOfCard64);
   operator ":=" (var l : Array; r : ArrayOfCard32);
   operator ":=" (var l : Array; r : ArrayOfCard16);
   operator ":=" (var l : Array; r : ArrayOfCard8);
   operator ":=" (var l : Array; r : ArrayOfString);
   operator "-" (a : Array) : Array;
   operator "=" (l, r : Array) : boolean;
   operator "=" (l : Array; r : ArrayOfNumber) : boolean;
   operator "=" (l : Array; r : ArrayOfReal64) : boolean;
   operator "=" (l : Array; r : ArrayOfReal32) : boolean;
   operator "=" (l : Array; r : ArrayOfInt64) : boolean;
   operator "=" (l : Array; r : ArrayOfInt32) : boolean;
   operator "=" (l : Array; r : ArrayOfInt16) : boolean;
   operator "=" (l : Array; r : ArrayOfInt8) : boolean;
   operator "=" (l : Array; r : ArrayOfCard64) : boolean;
   operator "=" (l : Array; r : ArrayOfCard32) : boolean;
   operator "=" (l : Array; r : ArrayOfCard16) : boolean;
   operator "=" (l : Array; r : ArrayOfCard8) : boolean;
   operator "=" (l : Array; r : ArrayOfString) : boolean;
   operator "=" (l : ArrayOfNumber; r : Array) : boolean;
   operator "=" (l : ArrayOfReal64; r : Array) : boolean;
   operator "=" (l : ArrayOfReal32; r : Array) : boolean;
   operator "=" (l : ArrayOfInt64; r : Array) : boolean;
   operator "=" (l : ArrayOfInt32; r : Array) : boolean;
   operator "=" (l : ArrayOfInt16; r : Array) : boolean;
   operator "=" (l : ArrayOfInt8; r : Array) : boolean;
   operator "=" (l : ArrayOfCard64; r : Array) : boolean;
   operator "=" (l : ArrayOfCard32; r : Array) : boolean;
   operator "=" (l : ArrayOfCard16; r : Array) : boolean;
   operator "=" (l : ArrayOfCard8; r : Array) : boolean;
   operator "=" (l : ArrayOfString; r : Array) : boolean;
   operator “#” (l, r : Array) : boolean;
   operator “#” (l : Array; r : ArrayOfNumber) : boolean;
   operator “#” (l : Array; r : ArrayOfReal64) : boolean;
   operator “#” (l : Array; r : ArrayOfReal32) : boolean;
   operator “#” (l : Array; r : ArrayOfInt64) : boolean;
   operator “#” (l : Array; r : ArrayOfInt32) : boolean;
   operator “#” (l : Array; r : ArrayOfInt16) : boolean;
   operator “#” (l : Array; r : ArrayOfInt8) : boolean;
   operator “#” (l : Array; r : ArrayOfCard64) : boolean;
   operator “#” (l : Array; r : ArrayOfCard32) : boolean;
   operator “#” (l : Array; r : ArrayOfCard16) : boolean;
   operator “#” (l : Array; r : ArrayOfCard8) : boolean;
   operator “#” (l : Array; r : ArrayOfString) : boolean;
   operator “#” (l : ArrayOfNumber; r : Array) : boolean;
   operator “#” (l : ArrayOfReal64; r : Array) : boolean;
   operator “#” (l : ArrayOfReal32; r : Array) : boolean;
   operator “#” (l : ArrayOfInt64; r : Array) : boolean;
   operator “#” (l : ArrayOfInt32; r : Array) : boolean;
   operator “#” (l : ArrayOfInt16; r : Array) : boolean;
   operator “#” (l : ArrayOfInt8; r : Array) : boolean;
   operator “#” (l : ArrayOfCard64; r : Array) : boolean;
   operator “#” (l : ArrayOfCard32; r : Array) : boolean;
   operator “#” (l : ArrayOfCard16; r : Array) : boolean;
   operator “#” (l : ArrayOfCard8; r : Array) : boolean;
   operator “#” (l : ArrayOfString; r : Array) : boolean;
   operator “+” (l, r : Array) : Array;
   operator “+” (l : Array; r : ArrayOfNumber) : Array;
   operator “+” (l : Array; r : ArrayOfReal64) : Array;
   operator “+” (l : Array; r : ArrayOfReal32) : Array;
   operator “+” (l : Array; r : ArrayOfInt64) : Array;
   operator “+” (l : Array; r : ArrayOfInt32) : Array;
   operator “+” (l : Array; r : ArrayOfInt16) : Array;
   operator “+” (l : Array; r : ArrayOfInt8) : Array;
   operator “+” (l : Array; r : ArrayOfCard64) : Array;
   operator “+” (l : Array; r : ArrayOfCard32) : Array;
   operator “+” (l : Array; r : ArrayOfCard16) : Array;
   operator “+” (l : Array; r : ArrayOfCard8) : Array;
   operator "+" (l : Array; r : ArrayOfString) : Array;
   operator "+" (l : ArrayOfNumber; r : Array) : Array;
   operator "+" (l : ArrayOfReal64; r : Array) : Array;
   operator "+" (l : ArrayOfReal32; r : Array) : Array;
   operator "+" (l : ArrayOfInt64; r : Array) : Array;
   operator "+" (l : ArrayOfInt32; r : Array) : Array;
   operator "+" (l : ArrayOfInt16; r : Array) : Array;
   operator "+" (l : ArrayOfInt8; r : Array) : Array;
   operator "+" (l : ArrayOfCard64; r : Array) : Array;
   operator "+" (l : ArrayOfCard32; r : Array) : Array;
   operator "+" (l : ArrayOfCard16; r : Array) : Array;
   operator "+" (l : ArrayOfCard8; r : Array) : Array;
   operator "+" (l : ArrayOfString; r : Array) : Array;
   operator "-" (l, r : Array) : Array;
   operator "-" (l : Array; r : ArrayOfNumber) : Array;
   operator "-" (l : Array; r : ArrayOfReal64) : Array;
   operator "-" (l : Array; r : ArrayOfReal32) : Array;
   operator "-" (l : Array; r : ArrayOfInt64) : Array;
   operator "-" (l : Array; r : ArrayOfInt32) : Array;
   operator "-" (l : Array; r : ArrayOfInt16) : Array;
   operator "-" (l : Array; r : ArrayOfInt8) : Array;
   operator "-" (l : Array; r : ArrayOfCard64) : Array;
   operator "-" (l : Array; r : ArrayOfCard32) : Array;
   operator "-" (l : Array; r : ArrayOfCard16) : Array;
   operator "-" (l : Array; r : ArrayOfCard8) : Array;
   operator "-" (l : Array; r : ArrayOfString) : Array;
   operator "-" (l : ArrayOfNumber; r : Array) : Array;
   operator "-" (l : ArrayOfReal64; r : Array) : Array;
   operator "-" (l : ArrayOfReal32; r : Array) : Array;
   operator "-" (l : ArrayOfInt64; r : Array) : Array;
   operator "-" (l : ArrayOfInt32; r : Array) : Array;
   operator "-" (l : ArrayOfInt16; r : Array) : Array;
   operator "-" (l : ArrayOfInt8; r : Array) : Array;
   operator "-" (l : ArrayOfCard64; r : Array) : Array;
   operator "-" (l : ArrayOfCard32; r : Array) : Array;
   operator "-" (l : ArrayOfCard16; r : Array) : Array;
   operator "-" (l : ArrayOfCard8; r : Array) : Array;
```

```
   operator "-" (l : ArrayOfString; r : Array) : Array;
   operator "*" (l : N.Number; r : Array) : Array;
   operator "*" (l : real{64}; r : Array) : Array;
   operator "*" (l : real{32}; r : Array) : Array;
   operator "*" (l : integer{64}; r : Array) : Array;
   operator "*" (l : integer{32}; r : Array) : Array;
   operator "*" (l : integer{16}; r : Array) : Array;
   operator "*" (l : integer{8}; r : Array) : Array;
   operator "*" (l : cardinal{64}; r : Array) : Array;
   operator "*" (l : cardinal{32}; r : Array) : Array;
   operator "*" (l : cardinal{16}; r : Array) : Array;
   operator "*" (l : cardinal{8}; r : Array) : Array;
   operator "*" (l : string; r : Array) : Array;
   operator "/" (l : Array; r : N.Number) : Array;
   operator "/" (l : Array; r : real{64}) : Array;
   operator "/" (l : Array; r : real{32}) : Array;
   operator "/" (l : Array; r : integer{64}) : Array;
   operator "/" (l : Array; r : integer{32}) : Array;
   operator "/" (l : Array; r : integer{16}) : Array;
   operator "/" (l : Array; r : integer{8}) : Array;
   operator "/" (l : Array; r : cardinal{64}) : Array;
   operator "/" (l : Array; r : cardinal{32}) : Array;
   operator "/" (l : Array; r : cardinal{16}) : Array;
   operator "/" (l : Array; r : cardinal{8}) : Array;
   operator "/" (l : Array; r : string) : Array;
   procedure OneNorm (v : Array) : N.Number;
   procedure TwoNorm (v : Array) : N.Number;
   procedure InfinityNorm (v : Array) : N.Number;
end Arrays.

module Bel.MF.Matrices;
   import
      Bel.MF.Numbers as N,
      Bel.MF.Arrays as A,
      Bel.Field as Field;
   type
      MathMatrix = array {math} *, * of N.Number;
      MatrixOfNumber = array *, * of N.Number;
      MatrixOfReal64 = array *, * of real{64};
      MatrixOfReal32 = array *, * of real{32};
      MatrixOfInt64 = array *, * of integer{64};
      MatrixOfInt32 = array *, * of integer{32};
      MatrixOfInt16 = array *, * of integer{16};
      MatrixOfInt8 = array *, * of integer{8};
      MatrixOfCard64 = array *, * of cardinal{64};
      MatrixOfCard32 = array *, * of cardinal{32};
      MatrixOfCard16 = array *, * of cardinal{16};
      MatrixOfCard8 = array *, * of cardinal{8};
      MatrixOfString = array *, * of string;
   type Matrix = object implements [], Field
      procedure Get (row, column : integer) : N.Number implements [].Get;
      procedure Set (row, column : integer; x : N.Number) implements [].Set;
      procedure Create (numberOfRows, numberOfColumns : integer);
      procedure Rows () : integer;
      procedure Columns () : integer;
      procedure GetRow (row : integer) : A.Array;
      procedure SetRow (row : integer; a : A.Array);
      procedure GetColumn (column : integer) : A.Array;
      procedure SetColumn (column : integer; a : A.Array);
      procedure GetDiagonal () : A.Array;
      procedure SetDiagonal (a : A.Array);
      procedure GetSubMatrix (startRow, endRow,
                       startColumn, endColumn : integer) : Matrix;
      procedure SetSubMatrix (atRow, atColumn : integer; m : Matrix);
      procedure GetMatrix () : MatrixOfNumber;
      procedure GetMathMatrix () : MathMatrix;
      procedure SetMatrix (m : Matrix);
      procedure SetMathMatrix (m : MathMatrix);
      procedure SetMatrixOfNumber (m : MatrixOfNumber);
      procedure SetMatrixOfReal64 (m : MatrixOfReal64);
      procedure SetMatrixOfReal32 (m : MatrixOfReal32);
      procedure SetMatrixOfInt64 (m : MatrixOfInt64);
      procedure SetMatrixOfInt32 (m : MatrixOfInt32);
      procedure SetMatrixOfInt16 (m : MatrixOfInt16);
      procedure SetMatrixOfInt8 (m : MatrixOfInt8);
      procedure SetMatrixOfCard64 (m : MatrixOfCard64);
      procedure SetMatrixOfCard32 (m : MatrixOfCard32);
      procedure SetMatrixOfCard16 (m : MatrixOfCard16);
      procedure SetMatrixOfCard8 (m : MatrixOfCard8);
      procedure SetMatrixOfString (m : MatrixOfString);
      procedure Equals (m : Matrix) : boolean;
      procedure IsAnArray () : boolean;
      procedure IsARowVector () : boolean;
      procedure IsAColumnVector () : boolean;
      procedure IsANumber () : boolean;
      procedure Dot (m : Matrix) : Matrix;
      procedure DotTranspose (m : Matrix) : Matrix;
      procedure TransposeDot (m : Matrix) : Matrix;
      procedure Contract (a : A.Array) : A.Array;
      procedure TransposeContract (a : A.Array) : A.Array;
      procedure DoubleDot (m : Matrix) : N.Number;
      procedure TransposeDoubleDot (m : Matrix) : N.Number;
      procedure Normalize (var scaleFactor : N.Number);
      procedure SwapRows (row1, row2 : integer);
      procedure SwapColumns (column1, column2 : integer);
   end Matrix;
   operator ":=" (var l : Matrix; r : Matrix);
   operator ":=" (var l : Matrix; r : MatrixOfNumber);
   operator ":=" (var l : Matrix; r : MatrixOfReal64);
   operator ":=" (var l : Matrix; r : MatrixOfReal32);
   operator ":=" (var l : Matrix; r : MatrixOfInt64);
   operator ":=" (var l : Matrix; r : MatrixOfInt32);
   operator ":=" (var l : Matrix; r : MatrixOfInt16);
   operator ":=" (var l : Matrix; r : MatrixOfInt8);
   operator ":=" (var l : Matrix; r : MatrixOfCard64);
   operator ":=" (var l : Matrix; r : MatrixOfCard32);
   operator ":=" (var l : Matrix; r : MatrixOfCard16);
   operator ":=" (var l : Matrix; r : MatrixOfCard8);
   operator ":=" (var l : Matrix; r : MatrixOfString);
   operator "-" (m : Matrix) : Matrix;
   operator "=" (l, r : Matrix) : boolean;
   operator "=" (l : Matrix; r : MatrixOfNumber) : boolean;
   operator "=" (l : Matrix; r : MatrixOfReal64) : boolean;
```

operator “=” (l : Matrix; r : MatrixOfReal32) : ***boolean***;
operator “=” (l : Matrix; r : MatrixOfInt64) : ***boolean***;
operator “=” (l : Matrix; r : MatrixOfInt32) : ***boolean***;
operator “=” (l : Matrix; r : MatrixOfInt16) : ***boolean***;
operator “=” (l : Matrix; r : MatrixOfInt8) : ***boolean***;
operator “=” (l : Matrix; r : MatrixOfCard64) : ***boolean***;
operator “=” (l : Matrix; r : MatrixOfCard32) : ***boolean***;
operator “=” (l : Matrix; r : MatrixOfCard16) : ***boolean***;
operator “=” (l : Matrix; r : MatrixOfCard8) : ***boolean***;
operator “=” (l : Matrix; r : MatrixOfString) : ***boolean***;
operator “=” (l : MatrixOfNumber; r : Matrix) : ***boolean***;
operator “=” (l : MatrixOfReal64; r : Matrix) : ***boolean***;
operator “=” (l : MatrixOfReal32; r : Matrix) : ***boolean***;
operator “=” (l : MatrixOfInt64; r : Matrix) : ***boolean***;
operator “=” (l : MatrixOfInt32; r : Matrix) : ***boolean***;
operator “=” (l : MatrixOfInt16; r : Matrix) : ***boolean***;
operator “=” (l : MatrixOfInt8; r : Matrix) : ***boolean***;
operator “=” (l : MatrixOfCard64; r : Matrix) : ***boolean***;
operator “=” (l : MatrixOfCard32; r : Matrix) : ***boolean***;
operator “=” (l : MatrixOfCard16; r : Matrix) : ***boolean***;
operator “=” (l : MatrixOfCard8; r : Matrix) : ***boolean***;
operator “=” (l : MatrixOfString; r : Matrix) : ***boolean***;
operator “#” (l, r : Matrix) : ***boolean***;
operator “#” (l : Matrix; r : MatrixOfNumber) : ***boolean***;
operator “#” (l : Matrix; r : MatrixOfReal64) : ***boolean***;
operator “#” (l : Matrix; r : MatrixOfReal32) : ***boolean***;
operator “#” (l : Matrix; r : MatrixOfInt64) : ***boolean***;
operator “#” (l : Matrix; r : MatrixOfInt32) : ***boolean***;
operator “#” (l : Matrix; r : MatrixOfInt16) : ***boolean***;
operator “#” (l : Matrix; r : MatrixOfInt8) : ***boolean***;
operator “#” (l : Matrix; r : MatrixOfCard64) : ***boolean***;
operator “#” (l : Matrix; r : MatrixOfCard32) : ***boolean***;
operator “#” (l : Matrix; r : MatrixOfCard16) : ***boolean***;
operator “#” (l : Matrix; r : MatrixOfCard8) : ***boolean***;
operator “#” (l : Matrix; r : MatrixOfString) : ***boolean***;
operator “#” (l : MatrixOfNumber; r : Matrix) : ***boolean***;
operator “#” (l : MatrixOfReal64; r : Matrix) : ***boolean***;

operator “#” (l : MatrixOfReal32; r : Matrix) : ***boolean***;
operator “#” (l : MatrixOfInt64; r : Matrix) : ***boolean***;
operator “#” (l : MatrixOfInt32; r : Matrix) : ***boolean***;
operator “#” (l : MatrixOfInt16; r : Matrix) : ***boolean***;
operator “#” (l : MatrixOfInt8; r : Matrix) : ***boolean***;
operator “#” (l : MatrixOfCard64; r : Matrix) : ***boolean***;
operator “#” (l : MatrixOfCard32; r : Matrix) : ***boolean***;
operator “#” (l : MatrixOfCard16; r : Matrix) : ***boolean***;
operator “#” (l : MatrixOfCard8; r : Matrix) : ***boolean***;
operator “#” (l : MatrixOfString; r : Matrix) : ***boolean***;
operator “+” (l, r : Matrix) : Matrix;
operator “+” (l : Matrix; r : MatrixOfNumber) : Matrix;
operator “+” (l : Matrix; r : MatrixOfReal64) : Matrix;
operator “+” (l : Matrix; r : MatrixOfReal32) : Matrix;
operator “+” (l : Matrix; r : MatrixOfInt64) : Matrix;
operator “+” (l : Matrix; r : MatrixOfInt32) : Matrix;
operator “+” (l : Matrix; r : MatrixOfInt16) : Matrix;
operator “+” (l : Matrix; r : MatrixOfInt8) : Matrix;
operator “+” (l : Matrix; r : MatrixOfCard64) : Matrix;
operator “+” (l : Matrix; r : MatrixOfCard32) : Matrix;
operator “+” (l : Matrix; r : MatrixOfCard16) : Matrix;
operator “+” (l : Matrix; r : MatrixOfCard8) : Matrix;
operator “+” (l : Matrix; r : MatrixOfString) : Matrix;
operator “+” (l : MatrixOfNumber; r : Matrix) : Matrix;
operator “+” (l : MatrixOfReal64; r : Matrix) : Matrix;
operator “+” (l : MatrixOfReal32; r : Matrix) : Matrix;
operator “+” (l : MatrixOfInt64; r : Matrix) : Matrix;
operator “+” (l : MatrixOfInt32; r : Matrix) : Matrix;
operator “+” (l : MatrixOfInt16; r : Matrix) : Matrix;
operator “+” (l : MatrixOfInt8; r : Matrix) : Matrix;
operator “+” (l : MatrixOfCard64; r : Matrix) : Matrix;
operator “+” (l : MatrixOfCard32; r : Matrix) : Matrix;
operator “+” (l : MatrixOfCard16; r : Matrix) : Matrix;
operator “+” (l : MatrixOfCard8; r : Matrix) : Matrix;
operator “+” (l : MatrixOfString; r : Matrix) : Matrix;
operator “-” (l, r : Matrix) : Matrix;
operator “-” (l : Matrix; r : MatrixOfNumber) : Matrix;

```
operator "-" (l : Matrix; r : MatrixOfReal64) : Matrix;
operator "-" (l : Matrix; r : MatrixOfReal32) : Matrix;
operator "-" (l : Matrix; r : MatrixOfInt64) : Matrix;
operator "-" (l : Matrix; r : MatrixOfInt32) : Matrix;
operator "-" (l : Matrix; r : MatrixOfInt16) : Matrix;
operator "-" (l : Matrix; r : MatrixOfInt8) : Matrix;
operator "-" (l : Matrix; r : MatrixOfCard64) : Matrix;
operator "-" (l : Matrix; r : MatrixOfCard32) : Matrix;
operator "-" (l : Matrix; r : MatrixOfCard16) : Matrix;
operator "-" (l : Matrix; r : MatrixOfCard8) : Matrix;
operator "-" (l : Matrix; r : MatrixOfString) : Matrix;
operator "-" (l : MatrixOfNumber; r : Matrix) : Matrix;
operator "-" (l : MatrixOfReal64; r : Matrix) : Matrix;
operator "-" (l : MatrixOfReal32; r : Matrix) : Matrix;
operator "-" (l : MatrixOfInt64; r : Matrix) : Matrix;
operator "-" (l : MatrixOfInt32; r : Matrix) : Matrix;
operator "-" (l : MatrixOfInt16; r : Matrix) : Matrix;
operator "-" (l : MatrixOfInt8; r : Matrix) : Matrix;
operator "-" (l : MatrixOfCard64; r : Matrix) : Matrix;
operator "-" (l : MatrixOfCard32; r : Matrix) : Matrix;
operator "-" (l : MatrixOfCard16; r : Matrix) : Matrix;
operator "-" (l : MatrixOfCard8; r : Matrix) : Matrix;
operator "-" (l : MatrixOfString; r : Matrix) : Matrix;
operator "*" (l : N.Number; r : Matrix) : Matrix;
operator "*" (l : real{64}; r : Matrix) : Matrix;
operator "*" (l : real{32}; r : Matrix) : Matrix;
operator "*" (l : integer{64}; r : Matrix) : Matrix;
operator "*" (l : integer{32}; r : Matrix) : Matrix;
operator "*" (l : integer{16}; r : Matrix) : Matrix;
operator "*" (l : integer{8}; r : Matrix) : Matrix;
operator "*" (l : cardinal{64}; r : Matrix) : Matrix;
operator "*" (l : cardinal{32}; r : Matrix) : Matrix;
operator "*" (l : cardinal{16}; r : Matrix) : Matrix;
operator "*" (l : cardinal{8}; r : Matrix) : Matrix;
operator "*" (l : string; r : Matrix) : Matrix;
operator "/" (l : Matrix; r : N.Number) : Matrix;
operator "/" (l : Matrix; r : real{64}) : Matrix;
```

```
   operator "/" (l : Matrix; r : real{32}) : Matrix;
   operator "/" (l : Matrix; r : integer{64}) : Matrix;
   operator "/" (l : Matrix; r : integer{32}) : Matrix;
   operator "/" (l : Matrix; r : integer{16}) : Matrix;
   operator "/" (l : Matrix; r : integer{8}) : Matrix;
   operator "/" (l : Matrix; r : cardinal{64}) : Matrix;
   operator "/" (l : Matrix; r : cardinal{32}) : Matrix;
   operator "/" (l : Matrix; r : cardinal{16}) : Matrix;
   operator "/" (l : Matrix; r : cardinal{8}) : Matrix;
   operator "/" (l : Matrix; r : string) : Matrix;
   procedure OneNorm (m : Matrix) : N.Number;
   procedure FrobeniusNorm (m : Matrix) : N.Number;
   procedure InfinityNorm (m : Matrix) : N.Number;
   procedure VectorProduct (l, r : A.Array) : Matrix;
end Matrices.
```

F Math Modules

```
module Bel.MATH.Series;
   import
      Bel.MF.Numbers as N,
      Bel.EvaluateSeries as E;
   procedure ContinuedFraction (a, b : object{E}; x : N.Number) : N.Number;
   procedure TruncatedContinuedFraction
                    (a, b : array * of N.Number; x : N.Number) : N.Number;
   procedure PowerSeries (a : object{E}; x : N.Number) : N.Number;
   procedure TruncatedPowerSeries
                    (a : array * of N.Number; x : N.Number) : N.Number;
   procedure RationalSeries (a, b : object{E}; x : N.Number) : N.Number;
   procedure TruncatedRationalSeries
                    (a, b : array * of N.Number; x : N.Number) : N.Number;
end Series.

module Bel.MATH.Functions;
   import
      Bel.MF.Numbers as N;
   var {immutable}
      E, Pi : N.Number;
   procedure Random () : N.Number;
   procedure Max (x, y : N.Number) : N.Number;
   procedure Min (x, y : N.Number) : N.Number;
   procedure Ceiling (x : N.Number) : N.Number;
   procedure Floor (x : N.Number) : N.Number;
   procedure Round (x : N.Number) : N.Number;
   procedure Abs (x : N.Number) : N.Number;
   procedure Sign (x : N.Number) : N.Number;
   procedure Sqrt (x : N.Number) : N.Number;
   procedure Pythag (x, y : N.Number) : N.Number;
   procedure Log (x : N.Number) : N.Number;
   procedure Ln (x : N.Number) : N.Number;
   procedure Exp (x : N.Number) : N.Number;
   procedure Sin (x : N.Number) : N.Number;
   procedure Cos (x : N.Number) : N.Number;
   procedure Tan (x : N.Number) : N.Number;
   procedure ArcSin (x : N.Number) : N.Number;
   procedure ArcCos (x : N.Number) : N.Number;
   procedure ArcTan (x : N.Number) : N.Number;
   procedure ArcTan2 (y, x : N.Number) : N.Number;
   procedure Sinh (x : N.Number) : N.Number;
   procedure Cosh (x : N.Number) : N.Number;
   procedure Tanh (x : N.Number) : N.Number;
   procedure ArcSinh (x : N.Number) : N.Number;
   procedure ArcCosh (x : N.Number) : N.Number;
   procedure ArcTanh (x : N.Number) : N.Number;
   procedure Gamma (x : N.Number) : N.Number;
   procedure Beta (x, y : N.Number) : N.Number;
end Functions.

module Bel.MATH.LinearAlgebra;
   import
      Bel.MF.Numbers as N,
      Bel.MF.Arrays as A,
      Bel.MF.Matrices as M;
   type {value} Lu = object
      procedure Initialize;
      procedure Factorize (a : M.Matrix);
      procedure Dimension () : integer;
      procedure Rank () : integer;
      procedure Determinant () : N.Number;
      procedure Inverse () : M.Matrix;
      procedure Solve (b : A.Array) : A.Array;
   end Lu;
   procedure RefineInverse (m : M.Matrix; var mInv : M.Matrix);
end LinearAlgebra.

module Bel.MATH.Interpolations;
   import
      Bel.MF.Numbers as N,
      Bel.MF.Arrays as A;
   procedure Neville (xVec, yVec : A.Array; x : N.Number) : N.Number;
   procedure Bilinear (x1, x2, y1, y2, valX1Y1, valX1Y2, valX2Y1, valX2Y2,
                       atX, atY : N.Number) : N.Number;
end Interpolations.

module Bel.MATH.Distributions;
   import
      Bel.MF.Numbers as N;
   type
      Certainty = (ninety, ninetyFive, ninetySevenPointFive,
                   ninetyNine, ninetyNinePointFive);
   procedure ChiSquared (percentagePoint : Certainty;
                       degreesOfFreedom : integer) : N.Number;
   procedure StudentT (percentagePoint : Certainty;
                       degreesOfFreedom : integer) : N.Number;
   procedure F (percentagePoint : Certainty;
                       degreesOfFreedom1, degreesOfFreedom2 : integer) : N.Number;
end Distributions.

module Bel.MATH.Derivatives;
   import
      Bel.MF.Numbers as N;
   type
      Method = (forward, central, backward, richardson);
      Y = procedure (N.Number) : N.Number;
   procedure Differentiate (y : Y; x, h : N.Number; m : Method) : N.Number;
end Derivatives.
```

```
module Bel.MATH.Integrals;
   import
      Bel.MF.Numbers as N;
   type
      Method = (trapezoidal, simpson, threeEights, romberg, gauss);
      F = procedure (N.Number) : N.Number;
   procedure Integrate (f : F; a, b : N.Number; m : Method) : N.Number;
end Integrals.
```

```
module Bel.MATH.RungeKutta;
   import
      Bel.MF.Numbers as N,
      Bel.MF.Arrays as A;
   type
      Order = (second, third, fourth);
      F = procedure (N.Number; A.Array) : A.Array;
   var
      count : integer;
   procedure Solve (f : F; tol : N.Number; ord : Order;
                      var x : N.Number; var y : A.Array; var h, err : N.Number);
end RungeKutta.
```

G Genetic Algorithm

The procedures inherited from the *Bel.Gene* and *Bel.Chromosome* definition interfaces have been implemented in the actual code, but are not repeated here to help keep the export interfaces concise.

```
module Bel.GA.Statistics;
   import
      Bel.MF.Numbers as N,
      Bel.MF.Arrays as A,
      Bel.MF.Matrices as M,
      Bel.MATH.Distributions as D;
   type
      LeastSquares = (linear, generalized, nonlinear);
      Model = procedure (A.Array; M.Matrix; A.Array) : M.Matrix;
   procedure FlipHeads (probabilityOfHeads : N.Number) : boolean;
   procedure RandomIntegerBetween (lowValue, highValue : integer) : integer;
   procedure Configure (expControl, expInput, expOutput : M.Matrix;
                      expEndsAtIndex, decimateTo : A.Array;
                      method : LeastSquares;
                      numericalModel : Model;
                      solveModelWithDecimatedDataOnly : boolean);
   procedure DataFitAgainst (var input, output : M.Matrix;
                      var expEndsAtIndex : A.Array);
   procedure Theory (parameters : A.Array) : M.Matrix;
   procedure Residuals (parameters : A.Array) : M.Matrix;
   procedure Covariance (parameters : A.Array) : M.Matrix;
   procedure RSquared (parameters : A.Array) : N.Number;
end Statistics.

module Bel.GA.Genes;
   import
      Bel.Gene as Gene;
   type Biallelic = object implements Gene
   end Biallelic;
   type Triallelic = object implements Gene
   end Triallelic;
   operator "=" (l, r : Biallelic) : boolean;
   operator "#" (l, r : Biallelic) : boolean;
   operator "=" (l, r : Triallelic) : boolean;
   operator "#" (l, r : Triallelic) : boolean;
end Genes.

module Bel.GA.Chromosomes;
   import
      Bel.MF.Numbers as N,
      Bel.GA.Genes as G,
      Bel.Chromosome as Chromosome;
   type
      BiChromosome = array * of G.Biallelic;
      TriChromosome = array * of G.Triallelic;
   type Haploid = object implements [], Chromosome
      procedure Get (index : integer) : G.Biallelic implements [].Get;
      procedure Set (index : integer; gene : G.Biallelic) implements [].Set;
   end Haploid;
   type Diploid = object implements [], Chromosome
      procedure Get (index : integer) : TriChromosome implements [].Get;
      procedure Set (index : integer; gene : TriChromosome) implements [].Set;
   end Diploid;
   operator "=" (l, r : Haploid) : boolean;
   operator "#" (l, r : Haploid) : boolean;
   operator "=" (l, r : Diploid) : boolean;
   operator "#" (l, r : Diploid) : boolean;
   procedure Crossover (probabilityOfCrossover : N.Number;
                        parentA, parentB : object{Chromosome};
                        var numberOfCrossovers : integer;
                        var childA, childB : object{Chromosome});
end Chromosomes.

module Bel.GA.Genomes;
   import
      Bel.MF.Numbers as N,
      Bel.MF.Arrays as A,
      Bel.Chromosome as Chromosome;
   type
      ChromosomeType = (haploid, diploid);
      Genotype = array * of object{Chromosome};
   type {ref} Genome = object implements []
      procedure Initialize (minParameters, maxParameters : A.Array;
                            numberOfSignificantFigures : integer;
                            chromosomeType : ChromosomeType);
      procedure Clone () : Genome;
      procedure Equals (g : Genome) : boolean;
      procedure Get (strand : integer) : object{Chromosome}
         implements [].Get;
      procedure Set (strand : integer; c : object{Chromosome})
         implements [].Set;
      procedure Strands () : integer;
      procedure Mutate (mutationProbability : N.Number;
                            var numberOfMutations : integer);
      procedure Encode (phenotypes : A.Array);
      procedure Decode () : A.Array;
   end Genome;
   operator "=" (l, r : Genome) : boolean;
   operator "#" (l, r : Genome) : boolean;
   procedure Crossover (probabilityOfCrossover : N.Number;
                        parentA, parentB : Genome;
```

```
                         var numberOfCrossovers : integer;
                         var childA, childB : Genome);
end Genomes.


module Bel.GA.Individuals;
   import
      Bel.MF.Numbers as N,
      Bel.MF.Arrays as A,
      Bel.GA.Genomes as G,
      Bel.Object as Object;
   type Lineage = object implements Object
      var
         birthID : integer;
         fitness : N.Number;
         parameters : A.Array;
   end Lineage;
   type {ref} Individual = object
      var {immutable}
         lineage : Lineage;
      procedure Initialize (fixedParameters : A.Array;
                           varyParameters : array of boolean;
      procedure Equals (i : Individual) : boolean;
      procedure Procreate (minParameters, maxParameters : A.Array;
                           numberOfSignificantFigures : integer;
                           chromosomeType : G.ChromosomeType);
      procedure Alien (parameters, minParameters, maxParameters : A.Array;
                           numberOfSignificantFigures : integer;
                           chromosomeType : G.ChromosomeType);
      procedure Conceive (genotype : G.Genome;
                           minParameters, maxParameters : A.Array;
                           numberOfSignificantFigures : integer;
                           chromosomeType : G.ChromosomeType);
      procedure Get () : G.Genome;
      procedure Mutate (mutationProbability : N.Number;
                           var numberOfMutations : integer);
   end Individual;
   operator "=" (l, r : Individual) : boolean;
   operator "#" (l, r : Individual) : boolean;
end Individuals.
```

```
module Bel.GA.Colonies;
   import
      System.IO.StreamWriter as StreamWriter,
      Bel.MF.Numbers as N,
      Bel.MF.Arrays as A,
      Bel.MF.Matrices as M,
      Bel.MATH.Distributions as D,
      Bel.GA.Statistics as S,
      Bel.GA.Genomes as G,
      Bel.GA.Individuals as I;
   type
      Inhabitants = array * of I.Individual;
   type {ref} Colony = object
      procedure Initialize (expControl, expInput, expOutput : M.Matrix;
                    expEndsAtIndex, decimateTo : A.Array;
                    method : S.LeastSquares;
                    numericalModel : S.Model;
                    solveModelWithDecimatedDataOnly : boolean;
                    fixedParameters, alienParameters,
                    minParameters, maxParameters : A.Array;
                    varyParameters : array * of boolean;
                    dimensionOfSchemata,
                    numberOfSignificantFigures : integer;
                    confidenceIntervalCertainty : D.Certainty;
                    probabilityOfCrossover, probabilityOfMutation,
                    probabilityOfImmigration : N.Number;
                    chromosomeType : G.ChromosomeType);
      procedure Propagate (var converged : boolean);
      procedure Parameters () : A.Array;
      procedure ConfidentParameters (percentagePoint : D.Certainty;
                    var bestParameters : A.Array;
                    var goodParameters : M.Matrix);
      procedure ConfidenceIntervals (goodParameters : M.Matrix;
                    var lowerBoundParameters,
                    upperBoundParameters : A.Array);
      procedure ReportHeader (sw : StreamWriter);
      procedure ReportBody (sw : StreamWriter);
      procedure ReportFooter (bestParameters : A.Array;
                    goodParameters : M.Matrix; sw : StreamWriter);
   end Colony;
end Colonies.
```

```
module Bel.GA.GeneticAlgorithm;
   import
      Bel.MF.Numbers as N,
      Bel.MF.Arrays as A,
      Bel.MF.Matrices as M,
      Bel.GA.Statistics as S,
      Bel.GA.Genomes as G,
      Bel.MATH.Distributions as D;
   procedure Optimize (fileNameForReport : string;
                        expControl, expInput, expOutput : M.Matrix;
                        expEndsAtIndex, decimateTo : A.Array;
                        method : S.LeastSquares;
                        numericalModel : S.Model;
                        solveModelWithDecimatedDataOnly : boolean;
                        fixedParameters, alienParameters,
                        minParameters, maxParameters : A.Array;
                        varyParameters : array * of boolean;
                        dimensionOfSchemata, numberOfSignificantFigures : integer;
                        confidenceIntervalForCertainty : D.Certainty;
                        probabilityOfCrossover, probabilityOfMutation,
                        probabilityOfImmigration : N.Number;
                        chromosomeType : G.ChromosomeType);
end GeneticAlgorithm.
```

H Physical Fields for Membrane Analysis

The procedures inherited from the *Bel.Object* and *Bel.Field* definition interfaces have been implemented in the actual code, but are not repeated here to help keep the export interfaces more concise.

```
module Bel.PF.Units;
   import
      Bel.Object as Object;
   var {immutable}
      Acceleration, Ampere, Candela, Dimensionless, Force, Hertz, Joule,
      Kelvin, Kilogram, Meter, Mole, Momentum, Newton, Pascal, Power, Rate,
      Second, Strain, StrainRate, Stress, StressRate, Time, Velocity, Watt,
      Work : Si;
   type Si = object implements Object
      procedure Parse (s : string);
      procedure Typeset () : string;
      procedure GetAmpere () : integer;
```

```
    procedure GetCandela () : integer;
    procedure GetKelvin () : integer;
    procedure GetKilogram () : integer;
    procedure GetMeter () : integer;
    procedure GetMole () : integer;
    procedure GetSecond () : integer;
    procedure SetAmpere (n : integer);
    procedure SetCandela (n : integer);
    procedure SetKelvin (n : integer);
    procedure SetKilogram (n : integer);
    procedure SetMeter (n : integer);
    procedure SetMole (n : integer);
    procedure SetSecond (n : integer);
    procedure IsVoid () : boolean;
    procedure Equals (u : Si) : boolean;
    procedure Add (u : Si) : Si;
    procedure Subtract (u : Si) : Si;
  end Si;
  operator "=" (l, r : Si) : boolean;
  operator '#' (l, r : Si) : boolean;
  operator '+' (l, r : Si) : Si;
  operator '-' (l, r : Si) : Si;
end Units.

module Bel.PF.Scalars;
  import
    Bel.MF.Numbers as N,
    Bel.PF.Units as U,
    Bel.Field as Field;
  type Scalar = object implements Field
    procedure Parse (s : string);
    procedure Typeset () : string;
    procedure ToString (significantDigits : integer) : string;
    procedure Get () : N.Number;
    procedure GetUnits () : U.Si;
    procedure Set (n : N.Number);
    procedure SetUnits (si : U.Si);
    procedure IsFinite () : boolean;
    procedure IsInfinite () : boolean;
    procedure IsNegativeInfinity () : boolean;
    procedure IsPositiveInfinity () : boolean;
    procedure IsNaN () : boolean;
    procedure IsVoid () : boolean;
    procedure Equals (s : Scalar) : boolean;
    procedure NotEqual (s : Scalar) : boolean;
    procedure GreaterThan (s : Scalar) : boolean;
    procedure GreaterThanOrEqual (s : Scalar) : boolean;
    procedure LessThan (s : Scalar) : boolean;
    procedure LessThanOrEqual (s : Scalar) : boolean;
    procedure Reciprocal () : Scalar;
    procedure Power (exponent : N.Number) : Scalar;
    procedure Magnitude () : Scalar;
end Scalar;
operator ":=" (var l : Scalar; r : Scalar);
operator ":=" (var l : Scalar; r : N.Number);
operator ":=" (var l : Scalar; r : real{64});
operator ":=" (var l : Scalar; r : real{32});
operator ":=" (var l : Scalar; r : integer{64});
operator ":=" (var l : Scalar; r : integer{32});
operator ":=" (var l : Scalar; r : integer{16});
operator ":=" (var l : Scalar; r : integer{8});
operator ":=" (var l : Scalar; r : cardinal{64});
operator ":=" (var l : Scalar; r : cardinal{32});
operator ":=" (var l : Scalar; r : cardinal{16});
operator ":=" (var l : Scalar; r : cardinal{8});
operator ":=" (var l : Scalar; r : string);
operator "-" (x : Scalar) : Scalar;
operator "=" (l, r : Scalar) : boolean;
operator "=" (l : Scalar; r : N.Number) : boolean;
operator "=" (l : Scalar; r : real{64}) : boolean;
operator "=" (l : Scalar; r : real{32}) : boolean;
operator "=" (l : Scalar; r : integer{64}) : boolean;
operator "=" (l : Scalar; r : integer{32}) : boolean;
operator "=" (l : Scalar; r : integer{16}) : boolean;
operator "=" (l : Scalar; r : integer{8}) : boolean;
operator "=" (l : Scalar; r : cardinal{64}) : boolean;
operator "=" (l : Scalar; r : cardinal{32}) : boolean;
operator "=" (l : Scalar; r : cardinal{16}) : boolean;
operator "=" (l : Scalar; r : cardinal{8}) : boolean;
operator "=" (l : N.Number; r : Scalar) : boolean;
operator "=" (l : real{64}; r : Scalar) : boolean;
operator "=" (l : real{32}; r : Scalar) : boolean;
operator "=" (l : integer{64}; r : Scalar) : boolean;
operator "=" (l : integer{32}; r : Scalar) : boolean;
operator "=" (l : integer{16}; r : Scalar) : boolean;
operator "=" (l : integer{8}; r : Scalar) : boolean;
operator "=" (l : cardinal{64}; r : Scalar) : boolean;
operator "=" (l : cardinal{32}; r : Scalar) : boolean;
operator "=" (l : cardinal{16}; r : Scalar) : boolean;
operator "=" (l : cardinal{8}; r : Scalar) : boolean;
operator "#" (l, r : Scalar) : boolean;
operator "#" (l : Scalar; r : N.Number) : boolean;
operator "#" (l : Scalar; r : real{64}) : boolean;
operator "#" (l : Scalar; r : real{32}) : boolean;
operator "#" (l : Scalar; r : integer{64}) : boolean;
operator "#" (l : Scalar; r : integer{32}) : boolean;
operator "#" (l : Scalar; r : integer{16}) : boolean;
operator "#" (l : Scalar; r : integer{8}) : boolean;
operator "#" (l : Scalar; r : cardinal{64}) : boolean;
operator "#" (l : Scalar; r : cardinal{32}) : boolean;
operator "#" (l : Scalar; r : cardinal{16}) : boolean;
operator "#" (l : Scalar; r : cardinal{8}) : boolean;
operator "#" (l : N.Number; r : Scalar) : boolean;
operator "#" (l : real{64}; r : Scalar) : boolean;
operator "#" (l : real{32}; r : Scalar) : boolean;
operator "#" (l : integer{64}; r : Scalar) : boolean;
operator "#" (l : integer{32}; r : Scalar) : boolean;
operator "#" (l : integer{16}; r : Scalar) : boolean;
operator "#" (l : integer{8}; r : Scalar) : boolean;
operator "#" (l : cardinal{64}; r : Scalar) : boolean;
operator "#" (l : cardinal{32}; r : Scalar) : boolean;
operator "#" (l : cardinal{16}; r : Scalar) : boolean;
operator "#" (l : cardinal{8}; r : Scalar) : boolean;
operator "<" (l, r : Scalar) : boolean;
operator "<" (l : Scalar; r : N.Number) : boolean;
operator "<" (l : Scalar; r : real{64}) : boolean;
operator "<" (l : Scalar; r : real{32}) : boolean;
operator "<" (l : Scalar; r : integer{64}) : boolean;
operator "<" (l : Scalar; r : integer{32}) : boolean;
operator "<" (l : Scalar; r : integer{16}) : boolean;
operator "<" (l : Scalar; r : integer{8}) : boolean;
operator "<" (l : Scalar; r : cardinal{64}) : boolean;
operator "<" (l : Scalar; r : cardinal{32}) : boolean;
operator "<" (l : Scalar; r : cardinal{16}) : boolean;
operator "<" (l : Scalar; r : cardinal{8}) : boolean;
operator "<" (l : N.Number; r : Scalar) : boolean;
operator "<" (l : real{64}; r : Scalar) : boolean;
operator "<" (l : real{32}; r : Scalar) : boolean;
operator "<" (l : integer{64}; r : Scalar) : boolean;
operator "<" (l : integer{32}; r : Scalar) : boolean;
operator "<" (l : integer{16}; r : Scalar) : boolean;
operator "<" (l : integer{8}; r : Scalar) : boolean;
operator "<" (l : cardinal{64}; r : Scalar) : boolean;
operator "<" (l : cardinal{32}; r : Scalar) : boolean;
operator "<" (l : cardinal{16}; r : Scalar) : boolean;
operator "<" (l : cardinal{8}; r : Scalar) : boolean;
operator "<=" (l, r : Scalar) : boolean;
operator "<=" (l : Scalar; r : N.Number) : boolean;
operator "<=" (l : Scalar; r : real{64}) : boolean;
operator "<=" (l : Scalar; r : real{32}) : boolean;
operator "<=" (l : Scalar; r : integer{64}) : boolean;
operator "<=" (l : Scalar; r : integer{32}) : boolean;
operator "<=" (l : Scalar; r : integer{16}) : boolean;
operator "<=" (l : Scalar; r : integer{8}) : boolean;
operator "<=" (l : Scalar; r : cardinal{64}) : boolean;
operator "<=" (l : Scalar; r : cardinal{32}) : boolean;
operator "<=" (l : Scalar; r : cardinal{16}) : boolean;
operator "<=" (l : Scalar; r : cardinal{8}) : boolean;
operator "<=" (l : N.Number; r : Scalar) : boolean;
operator "<=" (l : real{64}; r : Scalar) : boolean;
operator "<=" (l : real{32}; r : Scalar) : boolean;
operator "<=" (l : integer{64}; r : Scalar) : boolean;
operator "<=" (l : integer{32}; r : Scalar) : boolean;
operator "<=" (l : integer{16}; r : Scalar) : boolean;
operator "<=" (l : integer{8}; r : Scalar) : boolean;
operator "<=" (l : cardinal{64}; r : Scalar) : boolean;
operator "<=" (l : cardinal{32}; r : Scalar) : boolean;
operator "<=" (l : cardinal{16}; r : Scalar) : boolean;
operator "<=" (l : cardinal{8}; r : Scalar) : boolean;
operator ">" (l, r : Scalar) : boolean;
operator ">" (l : Scalar; r : N.Number) : boolean;
operator ">" (l : Scalar; r : real{64}) : boolean;
operator ">" (l : Scalar; r : real{32}) : boolean;
operator ">" (l : Scalar; r : integer{64}) : boolean;
operator ">" (l : Scalar; r : integer{32}) : boolean;
operator ">" (l : Scalar; r : integer{16}) : boolean;
operator ">" (l : Scalar; r : integer{8}) : boolean;
operator ">" (l : Scalar; r : cardinal{64}) : boolean;
operator ">" (l : Scalar; r : cardinal{32}) : boolean;
operator ">" (l : Scalar; r : cardinal{16}) : boolean;
operator ">" (l : Scalar; r : cardinal{8}) : boolean;
operator ">" (l : N.Number; r : Scalar) : boolean;
operator ">" (l : real{64}; r : Scalar) : boolean;
operator ">" (l : real{32}; r : Scalar) : boolean;
operator ">" (l : integer{64}; r : Scalar) : boolean;
operator ">" (l : integer{32}; r : Scalar) : boolean;
operator ">" (l : integer{16}; r : Scalar) : boolean;
operator ">" (l : integer{8}; r : Scalar) : boolean;
operator ">" (l : cardinal{64}; r : Scalar) : boolean;
operator ">" (l : cardinal{32}; r : Scalar) : boolean;
operator ">" (l : cardinal{16}; r : Scalar) : boolean;
operator ">" (l : cardinal{8}; r : Scalar) : boolean;
operator ">=" (l, r : Scalar) : boolean;
operator ">=" (l : Scalar; r : N.Number) : boolean;
operator ">=" (l : Scalar; r : real{64}) : boolean;
operator ">=" (l : Scalar; r : real{32}) : boolean;
operator “>=” (l : Scalar; r : integer{64}) : boolean;
operator “>=” (l : Scalar; r : integer{32}) : boolean;
operator “>=” (l : Scalar; r : integer{16}) : boolean;
operator “>=” (l : Scalar; r : integer{8}) : boolean;
operator “>=” (l : Scalar; r : cardinal{64}) : boolean;
operator “>=” (l : Scalar; r : cardinal{32}) : boolean;
operator “>=” (l : Scalar; r : cardinal{16}) : boolean;
operator “>=” (l : Scalar; r : cardinal{8}) : boolean;
operator “>=” (l : N.Number; r : Scalar) : boolean;
operator “>=” (l : real{64}; r : Scalar) : boolean;
operator “>=” (l : real{32}; r : Scalar) : boolean;
operator “>=” (l : integer{64}; r : Scalar) : boolean;
operator “>=” (l : integer{32}; r : Scalar) : boolean;
operator “>=” (l : integer{16}; r : Scalar) : boolean;
operator “>=” (l : integer{8}; r : Scalar) : boolean;
operator “>=” (l : cardinal{64}; r : Scalar) : boolean;
operator “>=” (l : cardinal{32}; r : Scalar) : boolean;
operator “>=” (l : cardinal{16}; r : Scalar) : boolean;
operator “>=” (l : cardinal{8}; r : Scalar) : boolean;
operator “+” (l, r : Scalar) : Scalar;
operator “+” (l : Scalar; r : N.Number) : Scalar;
operator “+” (l : Scalar; r : real{64}) : Scalar;
operator “+” (l : Scalar; r : real{32}) : Scalar;
operator “+” (l : Scalar; r : integer{64}) : Scalar;
operator “+” (l : Scalar; r : integer{32}) : Scalar;
operator “+” (l : Scalar; r : integer{16}) : Scalar;
operator “+” (l : Scalar; r : integer{8}) : Scalar;
operator “+” (l : Scalar; r : cardinal{64}) : Scalar;
operator “+” (l : Scalar; r : cardinal{32}) : Scalar;
operator “+” (l : Scalar; r : cardinal{16}) : Scalar;
operator “+” (l : Scalar; r : cardinal{8}) : Scalar;
operator “+” (l : N.Number; r : Scalar) : Scalar;
operator “+” (l : real{64}; r : Scalar) : Scalar;
operator “+” (l : real{32}; r : Scalar) : Scalar;
operator “+” (l : integer{64}; r : Scalar) : Scalar;
operator “+” (l : integer{32}; r : Scalar) : Scalar;
operator “+” (l : integer{16}; r : Scalar) : Scalar;
operator "+" (l : integer{8}; r : Scalar) : Scalar;
operator "+" (l : cardinal{64}; r : Scalar) : Scalar;
operator "+" (l : cardinal{32}; r : Scalar) : Scalar;
operator "+" (l : cardinal{16}; r : Scalar) : Scalar;
operator "+" (l : cardinal{8}; r : Scalar) : Scalar;
operator "-" (l, r : Scalar) : Scalar;
operator "-" (l : Scalar; r : N.Number) : Scalar;
operator "-" (l : Scalar; r : real{64}) : Scalar;
operator "-" (l : Scalar; r : real{32}) : Scalar;
operator "-" (l : Scalar; r : integer{64}) : Scalar;
operator "-" (l : Scalar; r : integer{32}) : Scalar;
operator "-" (l : Scalar; r : integer{16}) : Scalar;
operator "-" (l : Scalar; r : integer{8}) : Scalar;
operator "-" (l : Scalar; r : cardinal{64}) : Scalar;
operator "-" (l : Scalar; r : cardinal{32}) : Scalar;
operator "-" (l : Scalar; r : cardinal{16}) : Scalar;
operator "-" (l : Scalar; r : cardinal{8}) : Scalar;
operator "-" (l : N.Number; r : Scalar) : Scalar;
operator "-" (l : real{64}; r : Scalar) : Scalar;
operator "-" (l : real{32}; r : Scalar) : Scalar;
operator "-" (l : integer{64}; r : Scalar) : Scalar;
operator "-" (l : integer{32}; r : Scalar) : Scalar;
operator "-" (l : integer{16}; r : Scalar) : Scalar;
operator "-" (l : integer{8}; r : Scalar) : Scalar;
operator "-" (l : cardinal{64}; r : Scalar) : Scalar;
operator "-" (l : cardinal{32}; r : Scalar) : Scalar;
operator "-" (l : cardinal{16}; r : Scalar) : Scalar;
operator "-" (l : cardinal{8}; r : Scalar) : Scalar;
operator "*" (l, r : Scalar) : Scalar;
operator "*" (l : Scalar; r : N.Number) : Scalar;
operator "*" (l : Scalar; r : real{64}) : Scalar;
operator "*" (l : Scalar; r : real{32}) : Scalar;
operator "*" (l : Scalar; r : integer{64}) : Scalar;
operator "*" (l : Scalar; r : integer{32}) : Scalar;
operator "*" (l : Scalar; r : integer{16}) : Scalar;
operator "*" (l : Scalar; r : integer{8}) : Scalar;
operator "*" (l : Scalar; r : cardinal{64}) : Scalar;
operator "*" (l : Scalar; r : cardinal{32}) : Scalar;
operator "*" (l : Scalar; r : cardinal{16}) : Scalar;
operator "*" (l : Scalar; r : cardinal{8}) : Scalar;
operator "*" (l : N.Number; r : Scalar) : Scalar;
operator "*" (l : real{64}; r : Scalar) : Scalar;
operator "*" (l : real{32}; r : Scalar) : Scalar;
operator "*" (l : integer{64}; r : Scalar) : Scalar;
operator "*" (l : integer{32}; r : Scalar) : Scalar;
operator "*" (l : integer{16}; r : Scalar) : Scalar;
operator "*" (l : integer{8}; r : Scalar) : Scalar;
operator "*" (l : cardinal{64}; r : Scalar) : Scalar;
operator "*" (l : cardinal{32}; r : Scalar) : Scalar;
operator "*" (l : cardinal{16}; r : Scalar) : Scalar;
operator "*" (l : cardinal{8}; r : Scalar) : Scalar;
operator "/" (l, r : Scalar) : Scalar;
operator "/" (l : Scalar; r : N.Number) : Scalar;
operator "/" (l : Scalar; r : real{64}) : Scalar;
operator "/" (l : Scalar; r : real{32}) : Scalar;
operator "/" (l : Scalar; r : integer{64}) : Scalar;
operator "/" (l : Scalar; r : integer{32}) : Scalar;
operator "/" (l : Scalar; r : integer{16}) : Scalar;
operator "/" (l : Scalar; r : integer{8}) : Scalar;
operator "/" (l : Scalar; r : cardinal{64}) : Scalar;
operator "/" (l : Scalar; r : cardinal{32}) : Scalar;
operator "/" (l : Scalar; r : cardinal{16}) : Scalar;
operator "/" (l : Scalar; r : cardinal{8}) : Scalar;
operator "/" (l : N.Number; r : Scalar) : Scalar;
operator "/" (l : real{64}; r : Scalar) : Scalar;
operator "/" (l : real{32}; r : Scalar) : Scalar;
operator "/" (l : integer{64}; r : Scalar) : Scalar;
operator "/" (l : integer{32}; r : Scalar) : Scalar;
operator "/" (l : integer{16}; r : Scalar) : Scalar;
operator "/" (l : integer{8}; r : Scalar) : Scalar;
operator "/" (l : cardinal{64}; r : Scalar) : Scalar;
operator "/" (l : cardinal{32}; r : Scalar) : Scalar;
operator "/" (l : cardinal{16}; r : Scalar) : Scalar;
operator "/" (l : cardinal{8}; r : Scalar) : Scalar;
```

```
   operator "**" (l, r : Scalar) : Scalar;
   operator "**" (l : Scalar; r : N.Number) : Scalar;
   operator "**" (l : Scalar; r : real{64}) : Scalar;
   operator "**" (l : Scalar; r : real{32}) : Scalar;
   operator "**" (l : Scalar; r : integer{64}) : Scalar;
   operator "**" (l : Scalar; r : integer{32}) : Scalar;
   operator "**" (l : Scalar; r : integer{16}) : Scalar;
   operator "**" (l : Scalar; r : integer{8}) : Scalar;
   operator "**" (l : Scalar; r : cardinal{64}) : Scalar;
   operator "**" (l : Scalar; r : cardinal{32}) : Scalar;
   operator "**" (l : Scalar; r : cardinal{16}) : Scalar;
   operator "**" (l : Scalar; r : cardinal{8}) : Scalar;
   operator "**" (l : N.Number; r : Scalar) : Scalar;
   operator "**" (l : real{64}; r : Scalar) : Scalar;
   operator "**" (l : real{32}; r : Scalar) : Scalar;
   operator "**" (l : integer{64}; r : Scalar) : Scalar;
   operator "**" (l : integer{32}; r : Scalar) : Scalar;
   operator "**" (l : integer{16}; r : Scalar) : Scalar;
   operator "**" (l : integer{8}; r : Scalar) : Scalar;
   operator "**" (l : cardinal{64}; r : Scalar) : Scalar;
   operator "**" (l : cardinal{32}; r : Scalar) : Scalar;
   operator "**" (l : cardinal{16}; r : Scalar) : Scalar;
   operator "**" (l : cardinal{8}; r : Scalar) : Scalar;
end Scalars.

module Bel.PF.Vectors2;
   import
      Bel.MF.Numbers as N,
      Bel.MF.Arrays as A,
      Bel.PF.Units as U,
      Bel.PF.Scalars as S,
      Bel.Field as Field;
   type Vector = object implements [], Field
      procedure Get (row : integer) : S.Scalar implements [].Get;
      procedure Set (row : integer; s : S.Scalar) implements [].Set;
      procedure Parse (s : string);
      procedure Typeset () : string;
      procedure GetUnits () : U.Si;
      procedure SetUnits (si : U.Si);
      procedure GetArray () : A.Array;
      procedure SetArray (a : A.Array);
      procedure IsVoid () : boolean;
      procedure Equals (v : Vector) : boolean;
      procedure Dot (v : Vector) : S.Scalar;
   end Vector;
   operator ":=" (var l : Vector; r : Vector);
   operator ":=" (var l : Vector; r : A.Array);
   operator "-" (v : Vector) : Vector;
   operator "=" (l, r : Vector) : boolean;
   operator "=" (l : Vector; r : A.Array) : boolean;
   operator "=" (l : A.Array; r : Vector) : boolean;
   operator "#" (l, r : Vector) : boolean;
   operator "#" (l : Vector; r : A.Array) : boolean;
   operator "#" (l : A.Array; r : Vector) : boolean;
   operator "+" (l, r : Vector) : Vector;
   operator "+" (l : Vector; r : A.Array) : Vector;
   operator "+" (l : A.Array; r : Vector) : Vector;
   operator "-" (l, r : Vector) : Vector;
   operator "-" (l : Vector; r : A.Array) : Vector;
   operator "-" (l : A.Array; r : Vector) : Vector;
   operator "*" (l : S.Scalar; r : Vector) : Vector;
   operator "*" (l : N.Number; r : Vector) : Vector;
   operator "*" (l : real{64}; r : Vector) : Vector;
   operator "*" (l : real{32}; r : Vector) : Vector;
   operator "*" (l : integer{64}; r : Vector) : Vector;
   operator "*" (l : integer{32}; r : Vector) : Vector;
   operator "*" (l : integer{16}; r : Vector) : Vector;
   operator "*" (l : integer{8}; r : Vector) : Vector;
   operator "*" (l : cardinal{64}; r : Vector) : Vector;
   operator "*" (l : cardinal{32}; r : Vector) : Vector;
   operator "*" (l : cardinal{16}; r : Vector) : Vector;
   operator "*" (l : cardinal{8}; r : Vector) : Vector;
   operator "/" (l : Vector; r : S.Scalar) : Vector;
   operator "/" (l : Vector; r : N.Number) : Vector;
   operator "/" (l : Vector; r : real{64}) : Vector;
```

```
   operator "/" (l : Vector; r : real{32}) : Vector;
   operator "/" (l : Vector; r : integer{64}) : Vector;
   operator "/" (l : Vector; r : integer{32}) : Vector;
   operator "/" (l : Vector; r : integer{16}) : Vector;
   operator "/" (l : Vector; r : integer{8}) : Vector;
   operator "/" (l : Vector; r : cardinal{64}) : Vector;
   operator "/" (l : Vector; r : cardinal{32}) : Vector;
   operator "/" (l : Vector; r : cardinal{16}) : Vector;
   operator "/" (l : Vector; r : cardinal{8}) : Vector;
   procedure Norm (v : Vector) : S.Scalar;
   procedure UnitVector (v : Vector) : Vector;
end Vectors2.

module Bel.PF.Tensors2;
   import
      Bel.MF.Numbers as N,
      Bel.MF.Arrays as A,
      Bel.MF.Matrices as M,
      Bel.PF.Units as U,
      Bel.PF.Scalars as S,
      Bel.PF.Vectors2 as V,
      Bel.Field as Field;
   var {immutable}
      I : Tensor;
   type Tensor = object implements [], Field
      procedure Get (row, column : integer) : S.Scalar implements [].Get;
      procedure Set (row, column : integer; s : S.Scalar) implements [].Set;
      procedure Parse (s : string);
      procedure Typeset () : string;
      procedure TypesetRow (row : integer) : string;
      procedure GetUnits () : U.Si;
      procedure SetUnits (si : U.Si);
      procedure GetArray () : A.Array;
      procedure SetArray (a : A.Array);
      procedure GetMatrix () : M.Matrix;
      procedure SetMatrix (m : M.Matrix);
      procedure IsVoid () : boolean;
      procedure Equals (t : Tensor) : boolean;
      procedure Transpose () : Tensor;
      procedure Inverse () : Tensor;
      procedure Dot (t : Tensor) : Tensor;
      procedure DotTranspose (t : Tensor) : Tensor;
      procedure TransposeDot (t : Tensor) : Tensor;
      procedure Contract (v : V.Vector) : V.Vector;
      procedure TransposeContract (v : V.Vector) : V.Vector;
      procedure DoubleDot (t : Tensor) : S.Scalar;
      procedure TransposeDoubleDot (t : Tensor) : S.Scalar;
   end Tensor;
   operator ":=" (var l : Tensor; r : Tensor);
   operator ":=" (var l : Tensor; r : M.Matrix);
   operator ":=" (var l : Tensor; r : A.Array);
   operator "-" (t : Tensor) : Tensor;
   operator "=" (l, r : Tensor) : boolean;
   operator "=" (l : Tensor; r : M.Matrix) : boolean;
   operator "=" (l : Tensor; r : A.Array) : boolean;
   operator "=" (l : M.Matrix; r : Tensor) : boolean;
   operator "=" (l : A.Array; r : Tensor) : boolean;
   operator "#" (l, r : Tensor) : boolean;
   operator "#" (l : Tensor; r : M.Matrix) : boolean;
   operator "#" (l : Tensor; r : A.Array) : boolean;
   operator "#" (l : M.Matrix; r : Tensor) : boolean;
   operator "#" (l : A.Array; r : Tensor) : boolean;
   operator "+" (l, r : Tensor) : Tensor;
   operator "+" (l : Tensor; r : M.Matrix) : Tensor;
   operator "+" (l : Tensor; r : A.Array) : Tensor;
   operator "+" (l : M.Matrix; r : Tensor) : Tensor;
   operator "+" (l : A.Array; r : Tensor) : Tensor;
   operator "-" (l, r : Tensor) : Tensor;
   operator "-" (l : Tensor; r : M.Matrix) : Tensor;
   operator "-" (l : Tensor; r : A.Array) : Tensor;
   operator "-" (l : M.Matrix; r : Tensor) : Tensor;
   operator "-" (l : A.Array; r : Tensor) : Tensor;
   operator "*" (l : S.Scalar; r : Tensor) : Tensor;
   operator "*" (l : N.Number; r : Tensor) : Tensor;
   operator "*" (l : real{64}; r : Tensor) : Tensor;
```

```
   operator “*” (l : real{32}; r : Tensor) : Tensor;
   operator “*” (l : integer{64}; r : Tensor) : Tensor;
   operator “*” (l : integer{32}; r : Tensor) : Tensor;
   operator “*” (l : integer{16}; r : Tensor) : Tensor;
   operator “*” (l : integer{8}; r : Tensor) : Tensor;
   operator “*” (l : cardinal{64}; r : Tensor) : Tensor;
   operator “*” (l : cardinal{32}; r : Tensor) : Tensor;
   operator “*” (l : cardinal{16}; r : Tensor) : Tensor;
   operator “*” (l : cardinal{8}; r : Tensor) : Tensor;
   operator “/” (l : Tensor; r : S.Scalar) : Tensor;
   operator “/” (l : Tensor; r : N.Number) : Tensor;
   operator “/” (l : Tensor; r : real{64}) : Tensor;
   operator “/” (l : Tensor; r : real{32}) : Tensor;
   operator “/” (l : Tensor; r : integer{64}) : Tensor;
   operator “/” (l : Tensor; r : integer{32}) : Tensor;
   operator “/” (l : Tensor; r : integer{16}) : Tensor;
   operator “/” (l : Tensor; r : integer{8}) : Tensor;
   operator “/” (l : Tensor; r : cardinal{64}) : Tensor;
   operator “/” (l : Tensor; r : cardinal{32}) : Tensor;
   operator “/” (l : Tensor; r : cardinal{16}) : Tensor;
   operator “/” (l : Tensor; r : cardinal{8}) : Tensor;
   procedure Norm (t : Tensor) : S.Scalar;
   procedure FirstInvariant (t : Tensor) : S.Scalar;
   procedure SecondInvariant (t : Tensor) : S.Scalar;
   procedure Trace (t : Tensor) : S.Scalar;
   procedure Determinant (t : Tensor) : S.Scalar;
   procedure SymmetricPart (t : Tensor) : Tensor;
   procedure SkewPart (t : Tensor) : Tensor;
   procedure Eigenvalues (t : Tensor; var lambda1, lambda2 : S.Scalar);
   procedure SpectralDecomposition (t : Tensor; var Lambda, Q : Tensor);
end Tensors2.

module Bel.PF.QuadTensors2;
   import
      Bel.MF.Numbers as N,
      Bel.MF.Arrays as A,
      Bel.MF.Matrices as M,
      Bel.PF.Units as U,
      Bel.PF.Scalars as Sc,
      Bel.PF.Tensors2 as T,
      Bel.Field as Field;
   var {immutable}
      I, IBar, S, W, One, P : Tensor;
   type Tensor = object implements [], Field
      procedure Get (i, j, k, l : integer) : Sc.Scalar implements [].Get;
      procedure Set (i, j, k, l : integer; s : Sc.Scalar) implements [].Set;
      procedure TypesetRow (row : integer) : string;
      procedure GetUnits () : U.Si;
      procedure SetUnits (si : U.Si);
      procedure GetMatrix () : M.Matrix;
      procedure SetMatrix (m : M.Matrix);
      procedure IsVoid () : boolean;
      procedure Equals (t : Tensor) : boolean;
      procedure Transpose () : Tensor;
      procedure Inverse () : Tensor;
      procedure Dot (t : Tensor) : Tensor;
      procedure DotTranspose (t : Tensor) : Tensor;
      procedure TransposeDot (t : Tensor) : Tensor;
      procedure Contract (t : T.Tensor) : T.Tensor;
      procedure TransposeContract (t : T.Tensor) : T.Tensor;
   end Tensor;
   operator ":=" (var l : Tensor; r : Tensor);
   operator ":=" (var l : Tensor; r : M.Matrix);
   operator "-" (t : Tensor) : Tensor;
   operator "=" (l, r : Tensor) : boolean;
   operator "=" (l : Tensor; r : M.Matrix) : boolean;
   operator "=" (l : M.Matrix; r : Tensor) : boolean;
   operator "#" (l, r : Tensor) : boolean;
   operator "#" (l : Tensor; r : M.Matrix) : boolean;
   operator "#" (l : M.Matrix; r : Tensor) : boolean;
   operator "+" (l, r : Tensor) : Tensor;
   operator "+" (l : Tensor; r : M.Matrix) : Tensor;
   operator "+" (l : M.Matrix; r : Tensor) : Tensor;
   operator "-" (l, r : Tensor) : Tensor;
   operator "-" (l : Tensor; r : M.Matrix) : Tensor;
```

```
   operator "-" (l : M.Matrix; r : Tensor) : Tensor;
   operator "*" (l : Sc.Scalar; r : Tensor) : Tensor;
   operator "*" (l : N.Number; r : Tensor) : Tensor;
   operator "*" (l : real{64}; r : Tensor) : Tensor;
   operator "*" (l : real{32}; r : Tensor) : Tensor;
   operator "*" (l : integer{64}; r : Tensor) : Tensor;
   operator "*" (l : integer{32}; r : Tensor) : Tensor;
   operator "*" (l : integer{16}; r : Tensor) : Tensor;
   operator "*" (l : integer{8}; r : Tensor) : Tensor;
   operator "*" (l : cardinal{64}; r : Tensor) : Tensor;
   operator "*" (l : cardinal{32}; r : Tensor) : Tensor;
   operator "*" (l : cardinal{16}; r : Tensor) : Tensor;
   operator "*" (l : cardinal{8}; r : Tensor) : Tensor;
   operator "/" (l : Tensor; r : Sc.Scalar) : Tensor;
   operator "/" (l : Tensor; r : N.Number) : Tensor;
   operator "/" (l : Tensor; r : real{64}) : Tensor;
   operator "/" (l : Tensor; r : real{32}) : Tensor;
   operator "/" (l : Tensor; r : integer{64}) : Tensor;
   operator "/" (l : Tensor; r : integer{32}) : Tensor;
   operator "/" (l : Tensor; r : integer{16}) : Tensor;
   operator "/" (l : Tensor; r : integer{8}) : Tensor;
   operator "/" (l : Tensor; r : cardinal{64}) : Tensor;
   operator "/" (l : Tensor; r : cardinal{32}) : Tensor;
   operator "/" (l : Tensor; r : cardinal{16}) : Tensor;
   operator "/" (l : Tensor; r : cardinal{8}) : Tensor;
   procedure Norm (t : Tensor) : Sc.Scalar;
   procedure TensorProduct (l, r : T.Tensor) : Tensor;
   procedure SymTensorProduct (l, r : T.Tensor) : Tensor;
   procedure ODotProduct (l, r : T.Tensor) : Tensor;
   procedure SymODotProduct (l, r : T.Tensor) : Tensor;
end QuadTensors2.

module Bel.PF.Functions;
   import
      Bel.PF.Scalars as S;
   procedure Abs (s : S.Scalar) : S.Scalar;
   procedure Sign (s : S.Scalar) : S.Scalar;
   procedure Ceiling (s : S.Scalar) : S.Scalar;
   procedure Floor (s : S.Scalar) : S.Scalar;
   procedure Round (s : S.Scalar) : S.Scalar;
   procedure Max (s1, s2 : S.Scalar) : S.Scalar;
   procedure Min (s1, s2 : S.Scalar) : S.Scalar;
   procedure Pythag (s1, s2 : S.Scalar) : S.Scalar;
   procedure Sqrt (s : S.Scalar) : S.Scalar;
   procedure Log (s : S.Scalar) : S.Scalar;
   procedure Ln (s : S.Scalar) : S.Scalar;
   procedure Exp (s : S.Scalar) : S.Scalar;
   procedure Sin (s : S.Scalar) : S.Scalar;
   procedure Cos (s : S.Scalar) : S.Scalar;
   procedure Tan (s : S.Scalar) : S.Scalar;
   procedure ArcSin (s : S.Scalar) : S.Scalar;
   procedure ArcCos (s : S.Scalar) : S.Scalar;
   procedure ArcTan (s : S.Scalar) : S.Scalar;
   procedure ArcTan2 (y, x : S.Scalar) : S.Scalar;
   procedure Sinh (s : S.Scalar) : S.Scalar;
   procedure Cosh (s : S.Scalar) : S.Scalar;
   procedure Tanh (s : S.Scalar) : S.Scalar;
   procedure ArcSinh (s : S.Scalar) : S.Scalar;
   procedure ArcCosh (s : S.Scalar) : S.Scalar;
   procedure ArcTanh (s : S.Scalar) : S.Scalar;
   procedure Gamma (s : S.Scalar) : S.Scalar;
   procedure Beta (s1, s2 : S.Scalar) : S.Scalar;
end Functions.
```

H.1 In-Plane, Biaxial, Boundary-Value Problem

```
module Bel.BI.Kinematics;
   import
      Bel.PF.Scalars as S,
      Bel.PF.Tensors2 as T;
   procedure FInverse (f : T.Tensor) : T.Tensor;
   procedure B (f : T.Tensor) : T.Tensor;
   procedure BInverse (f : T.Tensor) : T.Tensor;
   procedure C (f : T.Tensor) : T.Tensor;
   procedure CInverse (f : T.Tensor) : T.Tensor;
   procedure R (f : T.Tensor) : T.Tensor;
   procedure U (f : T.Tensor) : T.Tensor;
   procedure UInverse (f : T.Tensor) : T.Tensor;
   procedure V (f : T.Tensor) : T.Tensor;
   procedure VInverse (f : T.Tensor) : T.Tensor;
   procedure HatF (currF, nextF : T.Tensor) : T.Tensor;
   procedure HatR (currF, nextF : T.Tensor) : T.Tensor;
   procedure L (f, dF : T.Tensor) : T.Tensor;
   procedure D (l : T.Tensor) : T.Tensor;
   procedure W (l : T.Tensor) : T.Tensor;
   procedure E (dTime : S.Scalar; currE, hatF, currL, nextL : T.Tensor) : T.Tensor;
   procedure DE (e, l : T.Tensor) : T.Tensor;
end Kinematics.

module Bel.BI.Kinetics;
   import
      Bel.PF.Vectors2 as V,
      Bel.PF.Tensors2 as T;
   procedure PtoS (f, p : T.Tensor) : T.Tensor;
   procedure PtoT (f, p : T.Tensor) : T.Tensor;
   procedure StoP (f, s : T.Tensor) : T.Tensor;
   procedure StoT (f, s : T.Tensor) : T.Tensor;
   procedure TtoP (f, t : T.Tensor) : T.Tensor;
   procedure TtoS (f, t : T.Tensor) : T.Tensor;
   procedure PtoTraction (f, p : T.Tensor) : V.Vector;
   procedure TractionToP (f : T.Tensor; t : V.Vector) : T.Tensor;
end Kinetics.
```

H.2 Tissue Mechanics

```
module Bel.TM.Hypoelastic.Isotropic;
   import
      Bel.DATA.Tree as Tree,
      Bel.PF.Scalars as SF,
      Bel.PF.Tensors2 as TF,
      Bel.Object as Object;
   type Datum = object implements Object
   var
      t : SF.Scalar;
      F : TF.Tensor;
      L : TF.Tensor;
      P : TF.Tensor;
      wp : SF.Scalar;
      E : TF.Tensor;
      T : TF.Tensor;
   end Datum;
   var
      dataTree : Tree;
   procedure InitializeDataTree;
   procedure BuildDataTree (experimentNumber, numberOfData : integer);
   procedure Solve (mu, beta : SF.Scalar);
   procedure Integrate (mu, beta, dTime, curWp : S.Scalar;
                        currT, hatF, currL, nextL : T.Tensor;
                        var nexWp : S.Scalar; var nextT : T.Tensor);
end Isotropic.
```

I GNU Free Documentation License

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

10. FUTURE REVISIONS OF THIS LICENSE

11. RELICENSING

ADDENDUM: How to use this License for your documents

J GNU Lesser General Public License

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc.

<http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions.

   As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

   "The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

   An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

   A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

   The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

   The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

   You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

   If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

   a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or

   b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

   The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

   a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.

   b) Accompany the object code with a copy of the GNU GPL and this license document.