A Genetic Algorithm for Pascal

Prof. Alan D. Freed
Department of Mechanical Engineering, Texas A&M University, College Station, TX

General Information

Contact

All feedback is welcome. Please email it to: `afreed @ tamu.edu`.

Software Version

1.3.3, November 2015

Programming Language and Required Packages

Library 'GenAlg' is written in Pascal, compiled in Free Pascal, and was developed using Lazarus. Library 'GenAlg' requires the 'Arrays' and 'rng' libraries written by the same author.

Copyright

Licensing

This software is licensed under the GNU Lesser General Public License, vs. 3 or later. The license reads:

GENALG is a free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

GENALG is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU

Lesser General Public License for more details.
You should have received a copy of the GUN Lesser General Public License along with RNG. If not, see <http://www.gnu.org/licenses/>.

References

1. Goldberg, D.E., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Boston, 1989.
2. Goldberg, D.E., *The Design of Innovation: Lessons learned from and for competent genetic algorithms.* In: *Genetic algorithms and evolutionary computation*, Vol. 7, Klewer, Boston, 2002.

Discussion of the Library Interface

A template comprised of four programming units that utilizes the interface of library 'GenAlg' is listed in the Appendix. One is for defining basic type and functions used throughout the suite. One is for entering the data. Another is for creating a model. And the third is a solver that calls them. You will need to write a driver that calls the solver, e.g., in the three demos supplied with the software the driver is a Lazarus program that creates figures to display output from the genetic algorithm. These template files are supplied with the software.

Genetic Algorithm

Details of the genetic algorithm are buried within the library. They do not permeate up to the interface. The author drew heavily on the books of Goldberg cited above in his construction of this genetic algorithm.

At the core of this genetic algorithm are haploid genes that can admit two states, passive and recessive, represented by a binary value. At a very low probability of occurrence, a gene may mutate from passive to active or vice versa.

A chromosome is an array of genes assembled as a Grey binary number. Grey numbers are selected over regular binary numbers so that mutation events have a more refined effect. Each chromosome represents a model parameter whose optimal value is being sought. A decoder/encoder pair maps a floating point real into a Grey binary array and back again. These are one-to-one mappings.

A genome is a collection of chromosomes, one for each parameter being sought. It contains the genetic information.

A creature is comprised of a genome and a fitness that associates with it. Fitness is taken to be a range-weighted harmonic sum of the root mean squared errors between model predictions acquired over a collection of experiments.

A colony is a population of creatures, usually numbering in the hundreds. To evolve a colony from one generation into the next, creatures from the existing generation are picked at random for mate selection via tournament play. From two tournaments, a pair of parent creatures get selected for mating with the outcome being a child creature that will move onto the next generation. Only the elite creature from an existing population moves onto the next generation. All other creatures perish at the moment when a new population of children from the current generation become the adults of the next generation. Mating involves a crossover event with a high probability of occurrence between like pairs of chromosomes from two parents. Where a pair of chromosomes are to be split and their genetic information swapped (i.e., crossover) is a probabilistic outcome. The end result is that a child's chromosome strands are comprised of gene segments from both parents.

The original generation is procreated. Every gene is assigned randomly with 50-50 odds in that generation. This provides a dispersed population over the window of admissible parametric values, a Monte Carlo sampling. To aid in maintaining a diverse population, a procreated immigrant or two may get introduced into a population at each generation at a low probability of occurrence. Mutation also aids in maintaining diversity. No clones are allowed in this implementation of a genetic algorithm. Instead of evolving a population of creatures to an existence where all creatures are clones of one another, this implementation of a genetic algorithm determines its convergence criteria and population size according to algorithms derived by Goldberg (2002) from probability theory.

Constants and Types

Constants and types are not exported by library 'GenAlg'. You must declare those that you use within your application.

The vector and matrix types used by 'GenAlg' come from the 'Arrays' library. The functions used to manage them can be found in their separate documentation and are imported in unit gaCore.

Library 'GenAlg' has five constants that dimension the problem which the programmer must size. From this, there are three additional variables used for dimensioning that are created at run time.

Library 'GenAlg' uses a procedure type to interface a model with the optimizer.

Functions and Procedures

These can be broken down into different classifications.

System Functions and Procedures

Library 'GenAlg' utilizes the following array types:

```
type
   TBVector = array of Boolean;
   TIVector = array of Integer;
   TRVector = array of Real;
   TSVector = array of String;
   TRMatrix = array of TRVector;
   TRData   = array of TRMatrix;
```

where functions to manage them can be found in the library 'Arrays'. *All arrays are considered to index from 1.*

'GenAlg' uses instances of a function type to assign model parameters that are to be held fixed during the optimization process. They are actually created after the varied parameters have been assigned. Sometimes it becomes useful to define a 'fixed' parameter as a function of 'varied' parameters, e.g., one may be able to provide more reliable max/min boundaries to the varied parameters when using a fixed parameter that is defined as a constraint in terms of the varied parameters.

```
type
   TVectorFn = function (var v : TRVector) : TRVector;
```

In the case of how a `TVectorFn` is used within `'GenAlg'`, the argument sent through the function contains the varied parameters for a creature, while the returned vector contains the fixed parameters for that creature.

'GenAlg' requires that the model you use to interface with it be a procedure of type

```
type
   TModel = procedure (
      experiment          : Integer;      // INPUT     exp is in range 1..dimE
      var modelParameters : TRVector;     // INPUT     [1..dimP]
      var controlData     : TRMatrix;     // INPUT     [1..dimNC[exp]][1..dimNS[exp]]
      var responseData    : TRMatrix);    // OUTPUT    [1..dimNR[exp]][1..dimNS[exp]]
```

where there are dimE experiments that can be handled (different formulæ describing the same model may apply to different boundary conditions, for example) with the model being comprised of dimP parameters, some may be fixed. A set of control data are provided for the selected experiment to which the model returns a compatible set of response data. For experiment exp, say, there are dimNC[exp] control variables and dimNR[exp] response variables collected at dimNS[exp] sampling points. The objective of the genetic algorithm is to determine an optimal set of model parameters that describe these response data.

Running the Genetic Algorithm

This interface is simple and intuitive. It is comprised of three procedures.

Procedure StartGeneticAlgorithm creates a genetic algorithm and populates it with its first generation. It requires eleven arguments to achieve this objective.

1. Argument numericalModel is a procedure that is an instance of the model type TModel which the programmer creates. It is the model whose parameters are being sought by this optimization technique.
2. Argument expControlData is an instance of type TRData. It specifies the independent data controlled in the various experiments to be considered.
3. Argument expResponseData is also an instance of type TRData. It specifies the dependent data whose responses were measured in the various experiments to be considered.
4. Argument varyParameters is an instance of type TBVector. It specifies whether or not a given parameter is to be allowed to be varied by the optimizer, or whose value is to be held fixed throughout the process.
5. Argument fixedParameters is an instance of type TVectorFn. This function assigns values to the fixed parameters which may depend on the varied parameters. If all parameters are to vary, then assign fixedParameters to nil.
6. Argument alienParameters is an instance of type TRVector. It is the user's best guess as to what the varied parametric values likely are.
7. Argument minParameters is an instance of type TRVector. It is a reasonable guess by the user as to what the lower bounds are for the varied parametric values.
8. Argument maxParameters is an instance of type TRVector. It is a reasonable guess by the user as to what the upper bounds are for the varied parametric values.
9. Argument significantFigs in an **Integer** that specifies how many significant figures one hopes to be able to secure in the solution provided. It is restricted to lie within the range of 1…7. This effects both the population size and the number of generations required to attain convergence.
10. Argument parameterNames is an instance of type TSVector. It provides the name for each parameter, varied and fixed, and is used when writing out the report.
11. The final argument reportFileName is a **String** that contains the name of the file (less any extension, it will

be given a .txt extension) that the report is to be written into. This report will appear in the directory that you execute your program in.

Procedure AdvanceToNextGeneration creates a new population of children and advances them to the next generation of adults. It outputs a single argument that specifies whether or not the algorithm has reached 'theoretical' convergence. Even if the algorithm has reached convergence, users can continue to call AdvanceToNextGeneration as many times as they like before stopping the program. The genetic algorithm itself is very fast. What effectively controls the performance of a particular application run is the efficiency of the solver used by the model whose parameter are being fit.

Procedure StopGeneticAlgorithm is a command that terminates the runtime of the genetic algorithm and writes out a report of its findings to a file specified in StartGeneticAlgorithm.

Procedure DeleteGeneticAlgorithm removes the dynamically allocated memory created by a genetic algorithm. Do no call this unless you no longer need to access the data from a running of the genetic algorithm.

Algorithmic Data

Procedure GeneticAlgorithmData outputs the number of creatures per generation (it is the same from one generation to the next in this implementation of a genetic algorithm), the number of generations that were created over the lifetime of the colony, the total number of immigrants that were introduced into the colony over all of the generations, the total number of crossovers that took place during mating over all of the generations, and the total number of gene mutations that took place over all of the generations.

Elite Creature Data

There are two procedures belonging to this category.

1. Procedure EliteCreatureData provides the fitness of the elite creature. The model parameters that associate with this fitness are returned as an instance of type TRVector whose associated Grey encodings are supplied as an instance of type TSVector. Fitness is defined by

$$\varphi(y) = \frac{1}{\mathrm{E}} \sum_{\mathrm{exp}=1}^{\mathrm{E}} \frac{\max\left(y^{\mathrm{exp}}\right) - \min\left(y^{\mathrm{exp}}\right)}{\mathrm{RMSE}\left(y^{\mathrm{model}} - y^{\mathrm{exp}}\right)}$$

   where E is the number of experiments, and y represents the response, either model or experiment, as specified, with RMSE being the root mean squared error between model and experiment. Each term in this harmonic sum is normalized by the range in its response, thereby rendering the overall fitness a dimensionless measure.
2. Procedure EliteCreatureFits provides data from which curves can be constructed to contrast model verses experiment for a selected experiment. All arrays are of type TRMatrix. Matrices expControl and expResponse are both returned values. Matrix modelControl is the only input matrix. It specifies the points where matrix modelResponse matrix is to be populated. It is not necessary that expControl and modelControl be of the same dimension, although that is the likely scenario. Typically, one wants a modelControl that is sufficiently dense so as to provide a smooth curve when plotted.

Population Fitness Data

There are two procedures belonging to this category.

1. Procedure FitnessMoments provides the first four sampling statistics, here created using the fitness data of the current population. These are the sample mean, the sample standard deviation, the sample skewness and the sample excess kurtosis. Skewness and kurtosis are both zero in a normally distributed sampling. If skewness is negative, the distribution has a heavy tail on the left side of the mean. And if it is positive, it has a heavy tail on the right side of the mean. If the excess kurtosis is negative, the distribution is flatter than a normal distribution. And if it is positive, the distribution is more peaked than a normal distribution.

2. Procedure FitnessStatistics provides the type of Johnson distribution (SB, bounded; SL, log-normal; SN, normal; SU, unbounded; and ST which is an SB distribution with $\gamma = 0$) that best describes the fitness data of the current population along with its four statistics: γ, δ, ξ and λ. Johnson distributions, as a collective set {SB, SL, SU} (SN and ST are special cases within this set), have the capability of describing the first, four, sample moments in any data set. A normal distribution, on the other hand, has the capability of describing the first, two, sample moments in any data set.

   Johnson distributions obey transformations that map them into normal distributions. Juxtapositions $\div$ between type designations and their mappings are as follows:

$$\text{SB} \div Z = \gamma + \delta \ln\left(\frac{X-\xi}{\xi+\lambda-X}\right) \quad \text{where} \quad \delta > 0,\ \xi < X_{min},\ X_{max} < \xi+\lambda$$

$$\text{SL} \div Z = \gamma + \delta \ln\left(\frac{X-\xi}{\lambda}\right) \quad \text{where} \quad \delta > 0,\ \lambda = \pm 1$$

$$\text{SN} \div Z = \gamma + \delta X \quad \text{where} \quad \delta = \frac{1}{\sigma} > 0,\ \gamma = -\frac{\mu}{\sigma}$$

$$\text{SU} \div Z = \gamma + \delta \operatorname{arcsinh}\left(\frac{X-\xi}{\lambda}\right) \quad \text{where} \quad \delta > 0,\ \lambda > 0$$

   where X is a random variate and Z is a standard, normal, random variate (with zero mean and a standard deviation of one) wherein μ denotes the sample mean and σ the sample standard deviation.

Example

A trivial example is to determine the parameters of a linear model used to fit noisy data. This is overkill in the sense that we are using a hammer to swat a fly, but the method is illustrative and the model is intuitive for the reader, so it makes a good case study. The linear model considered is distributed as demo 1.

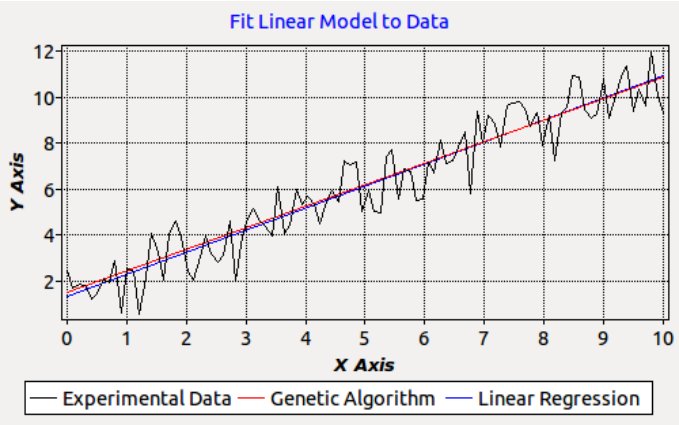


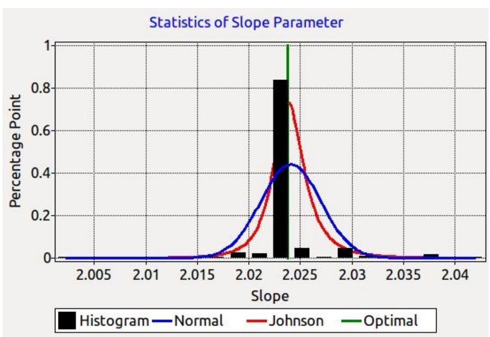
Illustration 1: Genetic algorithm vs. linear regression after two generations.

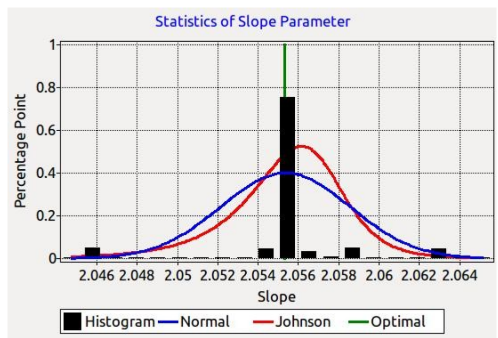
Illustration 1 presents typical output from demo 1. Here results from the genetic algorithm are presented after the 2nd generation (theoretical convergence is after the 27th generation, actual convergence was by the 5th generation). We see there is but a small difference between the fittings of these data from linear regression and from the genetic algorithm. This quickly disappears with increasing generations in that the lines become indistinguishable. This provides a measure of confidence in the overall implementation of the genetic algorithm.

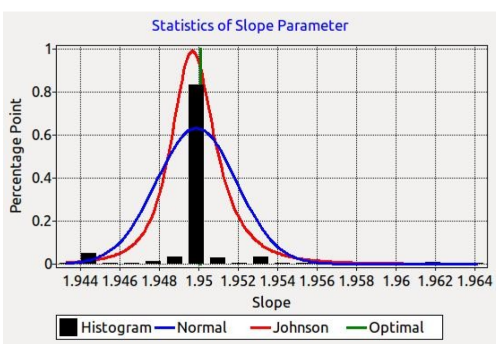
A Random Process

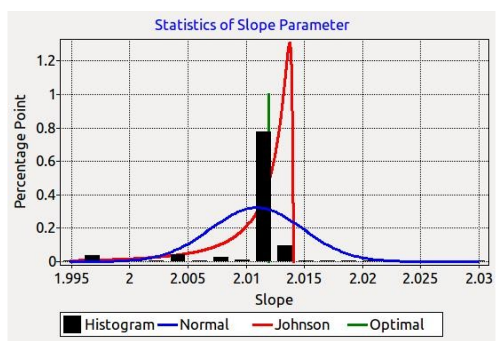
The genetic algorithm is truly a random process. I say this because the parameters held by the creatures of a population do not appear to have any common distribution. In the illustrations put forward below, distributions for the slope and the y-intercept in four successive runs are plotted looking at the fittest 50% of creatures. Each run is described by different sets of noisy data whose mean tendency is a straight line with a slope of 2 and a y-intercept of 1 generated with a standard deviation of 1 in white noise. The outcomes are all different.

The following figures are from four successive runs where we record distributions of the slope amongst the top half of all creatures in the final population of the colony.

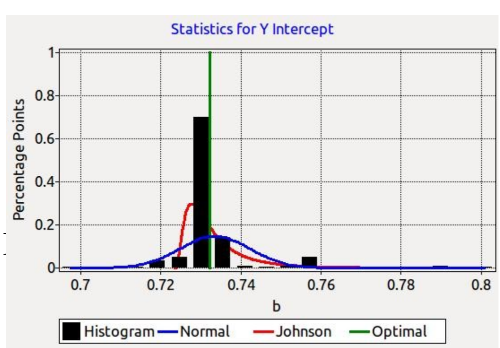


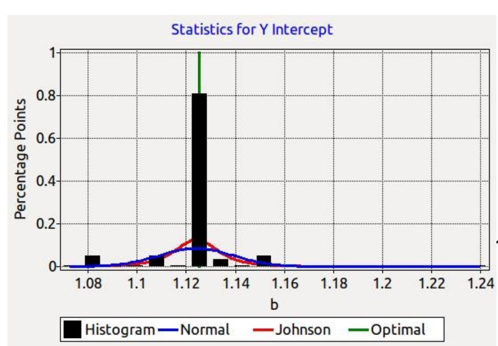


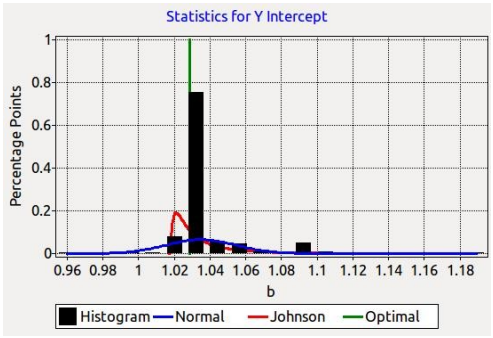


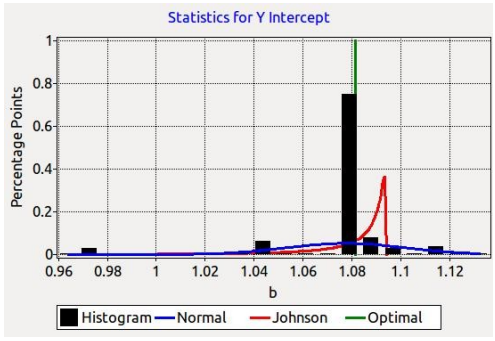


The next set of figures are from four additional successive runs where we record distributions of the y-intercept amongst the top half of all creatures in the final population of the colony.









What is apparent in all eight images is the efficiency of the genetic algorithm to lock onto an optimal solution whenever a single optimum is present.

Report

Each run of the optimizer generates a report. An example of such a report is listed below.

```
--------------------------------------------------------------------------------
    An optimization of 1 random variable extracted from 1 experiment
    comprising of 100 total data points fit with a model in 2 parameters
    using a genetic algorithm with a population of size 626.
--------------------------------------------------------------------------------
    |    elite    | ------------------ population fitness ------------------
 #  |   fitness   | --- mean --- | - std dev - | - skewness - | - kurtosis -
  1   2.142252E+1      2.65E+0       3.52E+0        2.54E+0        7.13E+0
  2   2.143342E+1      8.86E+0       5.44E+0        4.85E-1       -5.23E-1
  3   2.145444E+1      1.57E+1       7.29E+0       -9.47E-1       -6.86E-1
  4   2.145445E+1      1.66E+1       7.31E+0       -1.24E+0       -1.31E-1
  5   2.145446E+1      1.67E+1       7.32E+0       -1.21E+0       -2.14E-1
  6   2.145446E+1      1.66E+1       7.45E+0       -1.19E+0       -2.85E-1
  7   2.145446E+1      1.71E+1       7.14E+0       -1.37E+0        2.02E-1
  8   2.145446E+1      1.64E+1       7.57E+0       -1.12E+0       -4.58E-1
  9   2.145446E+1      1.72E+1       6.89E+0       -1.40E+0        3.34E-1
 10   2.145446E+1      1.70E+1       7.11E+0       -1.28E+0       -1.98E-2
 11   2.145446E+1      1.77E+1       6.55E+0       -1.61E+0        1.04E+0
 12   2.145446E+1      1.67E+1       7.42E+0       -1.26E+0       -1.17E-1
 13   2.145446E+1      1.66E+1       7.30E+0       -1.16E+0       -3.26E-1
 14   2.145446E+1      1.66E+1       7.31E+0       -1.14E+0       -4.07E-1
```

```
 15   2.145446E+1      1.66E+1       7.32E+0       -1.13E+0       -4.26E-1
 16   2.145446E+1      1.73E+1       6.81E+0       -1.38E+0        2.82E-1
 17   2.145446E+1      1.67E+1       7.16E+0       -1.21E+0       -1.74E-1
 18   2.145446E+1      1.67E+1       7.27E+0       -1.20E+0       -2.52E-1
 19   2.145446E+1      1.70E+1       7.01E+0       -1.33E+0        1.48E-1
 20   2.145446E+1      1.73E+1       6.72E+0       -1.41E+0        4.31E-1
 21   2.145446E+1      1.65E+1       7.39E+0       -1.17E+0       -3.09E-1
 22   2.145446E+1      1.65E+1       7.42E+0       -1.17E+0       -3.18E-1
 23   2.145446E+1      1.69E+1       7.18E+0       -1.32E+0        9.13E-2
 24   2.145446E+1      1.70E+1       7.14E+0       -1.31E+0        4.81E-2
 25   2.145446E+1      1.67E+1       7.11E+0       -1.21E+0       -1.77E-1
 26   2.145446E+1      1.65E+1       7.33E+0       -1.20E+0       -2.19E-1
 27   2.145446E+1      1.68E+1       7.17E+0       -1.27E+0       -2.29E-2
-  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
    Statistics from this optimization run include:
        number of generations = 27
        number of creatures   = 626
        number of immigrants  = 52
        number of crossovers  = 26192
        number of mutations   = 77877
-  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
    Fitness statistics from the last generation are:
    | Normal Distribution || ------------- Johnson   Distribution ----------
    |   mean    |  std dev || S? |  gamma    |  delta    |    xi     |  lambda
      1.68E+01   7.17E+00    SB  -7.77E-01   1.97E-01   1.04E+00   2.06E+01
-  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
    Parameters from the elite creature are:
 #       lower      optimum      upper      name of parameter
 1     4.000E-01   1.968E+00   1.000E+01    slope m
 2     2.000E-01   1.199E+00   5.000E+00    intercept b
-------------------------------------------------------------------------------
```

Appendix: A Template

There are five parts to the template provided with 'GenAlg'. The first provides basic types and functions used throughout the suite. It is gaCore. The second is used to create your model in a format that is accessible to the optimizer, i.e., gaModel. A third is used to enter a data set in a format that can be utilized by the optimizer, i.e., gaData. The fourth is a solver that calls the genetic algorithm, i.e., gaSolver. And the fifth is a driver that runs the optimizer, i.e., gaDriver. Units gaModel and gaData will require a fair amount of coding for your specific application. See the demos to get an idea of how this is done. Unit gaSolver and program gaDriver will not likely require any modification, at least to start with. You may want to alter them to better suit your needs as you gain experience using this optimizer.

What is typeset in red requires your attention.

GaCore

```
// global directives
{$MODE OBJFPC}      // allows objects, classes, interfaces, exception handling
{$H+}               // use ANSI strings
{$IFDEF UNIX}
  {$CALLING CDECL}  // uses the GCC calling convention  - for Unix libraries
  {$PIC ON}         // create Position Independent Code - for Unix libraries
{$ENDIF}
unit gaCore;       // you may want to rename this to better suit your needs

interface

  uses
    {$IFDEF UNIX}{$IFDEF UseCThreads}
      cthreads,
    {$ENDIF}{$ENDIF}
    Classes, SysUtils;

  // Array types defined in 'Arrays' and used by the 'GenAlg' library

  type
    TBVector = array of Boolean;
    TIVector = array of Integer;
    TRVector = array of Real;
    TSVector = array of String;
    TRMatrix = array of TRVector;
    TRData   = array of TRMatrix;

  // Functions used to create and manage the array types of 'Arrays'.

  function NewBVector (len : Integer) : TBVector;
  // Creates a boolean vector indexing from 1 to len with elements of FALSE.

  function NewIVector (len : Integer) : TIVector;
  // Creates an integer vector indexing from 1 to len whose elements are all 0.

  function NewRVector (len : Integer) : TRVector;
  // Creates a real vector indexing from 1 to len whose elements are all 0.0.

  function NewSVector (len : Integer) : TSVector;
  // Creates a string vector indexing from 1 to len whose elements are all ''.

  function LenBVector (var v : TBVector) : Integer;
  // Returns the length of boolean vector v.

  function LenIVector (var v : TIVector) : Integer;
  // Returns the length of integer vector v.
```

```
  function LenRVector (var v : TRVector) : Integer;
  // Returns the length of real vector v.

  function LenSVector (var v : TSVector) : Integer;
  // Returns the length of string vector v.

  function NewRMatrix (rows, columns : Integer) : TRMatrix;
  // Creates a real matrix indexing as [1..rows][1..columns]
  // whose elements are assigned values of 0.0.

  function NewRData (experiments         : Integer;
                     var variablesPerExp : TIVector;
                     var samplesPerExp   : TIVector) : TRData;
  // Creates a new real-valued data structure indexing as
  // [1..experiments][1..variablesPerExp[experiment]][1..samplesPerExp[experiment]
  // whose elements are all assigned values of 0.0.

  // You may need other functions exported by 'Arrays'.  These are most common.

implementation

  function NewBVector (len : Integer) : TBVector; external 'Arrays';

  function NewIVector (len : Integer) : TIVector; external 'Arrays';

  function NewRVector (len : Integer) : TRVector; external 'Arrays';

  function NewSVector (len : Integer) : TSVector; external 'Arrays';

  function LenBVector (var v : TBVector) : Integer; external 'Arrays';

  function LenIVector (var v : TIVector) : Integer; external 'Arrays';

  function LenRVector (var v : TRVector) : Integer; external 'Arrays';

  function LenSVector (var v : TSVector) : Integer; external 'Arrays';

  function NewRMatrix (rows, columns : Integer) : TRMatrix; external 'Arrays';

  function NewRData (experiments : Integer; var variablesPerExp : TIVector;
                     var samplesPerExp : TIVector) : TRData; external 'Arrays';

begin
end.
```

gaData

```
// global directives
{$MODE OBJFPC}      // allows objects, classes, interfaces, exception handling
{$H+}               // use ANSI strings
{$IFDEF UNIX}
  {$CALLING CDECL}  // uses the GCC calling convention  - for Unix libraries
  {$PIC ON}         // create Position Independent Code - for Unix libraries
{$ENDIF}

unit gaData;       // you may want to rename this to something more suitable

interface

  uses
    {$IFDEF UNIX}{$IFDEF UseCThreads}
      cthreads,
    {$ENDIF}{$ENDIF}
    Classes, SysUtils;

  // Exported constants/variables for the genetic algorithm for this application

  const
    dimE = ?;           // number of experiments
    dimR = ?;           // number of response variables over all experiments
    dimS = ?;           // number of fitted sample points over all experiments

  var
    dimNC : TIVector;   // number of control  variables per experiment
    dimNR : TIVector;   // number of response variables per experiment
    dimNS : TIVector;   // number of sampling points taken per experiment

  // ---------------------------------------------------------------------------

  // Exported procedures used by the solver

  // Data are staggered arrays of real matrices which index from
  //    [1..experiments][1..variables[experiment]][1..samples[experiment]]
  // where experiment lies within the interval of 1..experiments,
  // variables index from 1..experiments and
  // samples   index from 1..experiments.

  function ExperimentalControlData  : TRData;

  function ExperimentalResponseData : TRData;

implementation
```

```
function ExperimentalControlData : TRData;
  var
    c, e, s : Integer;
    data    : TRData;
begin
  // dimensioning the controls
  dimNC := NewIVector(dimE);
  // assign the number of control variables used in each experiment
  dimNC[1] := ?;
  ...
  // dimensioning the samplings
  if dimNS = nil then begin    // may be assigned in ExperimentalResponseData
    dimNS := NewIVector(dimE);
    // assign the number of samplings taken in each experiment
    dimNS[1] := ?;
    ...
  end;
  // dimensioning the return array of control variables
  data := NewRData(dimE, dimNC, dimNS);
  // assigning the data to be returned
  for e := 1 to dimE do
    for c := 1 to dimNC[e] do
      for s := 1 to dimNS[e] do begin
        // assign the control data to its data structure
        data[e][c][s] := ?;
        ...
      end;
  // returning the data
  ExperimentalControlData := data
end;

function ExperimentalResponseData : TRData;
  var
    e, r, s : Integer;
    data    : TRData;
begin
  // dimensioning the responses
  dimNR := NewIVector(dimE);
  // assign the number of response variables used in each experiment
  dimNR[1] := ?;
  ...
  // dimensioning the samplings
  if dimNS = nil then begin   // may be assigned in ExperimentalControlData
    dimNS := NewIVector(dimE);
    // assign the number of samplings taken in each experiment
    dimNS[1] := ?;
    ...
  end;
```

```
    // dimensioning the return array of response variables
    data := NewRData(dimE, dimNR, dimNS);
    // assigning the data to be returned
    for e := 1 to dimE do
      for r := 1 to dimNR[e] do
        for s := 1 to dimNS[e] do begin
          // assign the response data to its data structure
          data[e][r][s] := ?;
          ...
        end;
    // returning the data
    ExperimentalResponseData := data
  end;

begin

  dimNC := nil;
  dimNR := nil;
  dimNS := nil;

  // initialize any additional global variables used in this unit

end.
```

gaModel

```
// global directives
{$MODE OBJFPC}      // allows objects, classes, interfaces, exception handling
{$H+}               // use ANSI strings
{$IFDEF UNIX}
  {$CALLING CDECL}  // uses the GCC calling convention  - for Unix libraries
  {$PIC ON}         // create Position Independent Code - for Unix libraries
{$ENDIF}

unit gaModel;      // you may rename this to suit your needs

interface

  uses
    {$IFDEF UNIX}{$IFDEF UseCThreads}
     cthreads,
    {$ENDIF}{$ENDIF}
    Classes, SysUtils, gaData;

  // Exported constants for the genetic algorithm pertinent to this application
```

```
  const
    dimP  = ?;          // number of parameters, both fixed and varied
    dimPV = ?;          // number of parameters that are allowed to vary

  // A model is implemented for parameter estimation via an instance of type

  type
    TModel = procedure (experiment           : Integer;      // input
                        var modelParameters  : TRVector;     // input
                        var controlData      : TRMatrix;     // input
                        var responseData     : TRMatrix);    // output
{ where
    experiment       : used to determine which experiment to get response data
                       an interger between 1 and dimE
    modelParameters  : all model parameters (both fixed and adjustable)
                       indices range over [1..dimP]
    controlData      : a sequence of experimentally controlled values
                       [controlVariable][valueAtASamplingInstant]
                       indices range over [1..dimNC[e]][1..dimNS[e]]
    responseData     : predicted responses for these controls & parameters
                       [responseVariable][valueAtASamplingInstant]
                       indices range over [1..dimNR[e]][1..dimNS[e]]          }

  // Instances are used to assign values to the fixed paramters of a model

  type
    TVectorFn = function (var v : TRVector) : TRVector;

  // Exported procedures and functions used by the driver

  procedure MyModel (experiment           : Integer;      // input
                     var modelParameters  : TRVector;     // input
                     var controlData      : TRMatrix;     // input
                     var responseData     : TRMatrix);    // output
  { an instance of type TModel }

  function MyFixedParameters (var variedParameters : TRVector) : TRVector;
  { an instance of type TVectorFn }

  // Associated functions that are sent to the genetic algorithm

  function VaryParameters : TBVector;
  { specify which parameters are to be varied (TRUE) and which are not (FALSE) }

  function NameParameters : TSVector;
  { assign names to all of the parameters, varied and fixed, for the report }

  // The remaining procedure provide information for the varied parameters only
```

```
  function AlienParameters : TRVector;
  { best guess at what the varied parameters might be }

  function MaximumParameters : TRVector;
  { upper boundary of the search domain for all of the varied parameters }

  function MinimumParameters : TRVector;
  { lower boundary of the search domain for all of the varied parameters }

implementation

  // Create the model to be used to fit the data against

  procedure MyModel (experiment           : Integer;      // input
                     var modelParameters  : TRVector;     // input
                     var controlData      : TRMatrix;     // input
                     var responseData     : TRMatrix);    // output
  begin
    ...
  end;

  // Assign the fixed parameters, which may be functions of the ones varied

  function MyFixedParameters (var variedParameters : TRVector) : TRVector;
    var
      fixed : TRVector;
  begin
    if dimP = dimPV then
      fixed := nil
    else begin
      // these sequence according to VaryParameters, i.e., the first occurance
      // where VaryParameters[i] is FALSE gets assigned to FixedParameters[1]
      fixed := NewRVector(dimP-dimPV);
      fixed[1] := ?;
      ...
    end;
    FixedParameters := fixed
  end;

  // user-defined procedures required in every GenAlg application

  // the lengths of VaryParameters and NameParameters are for all parameters
  // both varied and fixed

  function VaryParameters : TBVector;
    var
      vary : TBVector;
```

```
begin
  vary := NewBVector(dimP);
  vary[1] := ?;
  ...
  VaryParameters := vary
end;

function NameParameters : TSVector;
  var
    names : TSVector;
begin
  names := NewSVector(dimP);
  // assign names to both the fixed and varied parameters
  names[1] := ?;
  ...
  NameParameters := names
end;

// all remaining functions return vectors of the varied parameter length

function AlienParameters : TRVector;
  var
    alien : TRVector;
begin
  // this is a guess at what one expects the varied parameters should be
  alien := NewRVector(dimPV);
  alien[1] := ?;
  ...
  AlienParameters := alien
end;

function MaximumParameters : TRVector;
  var
    maxP : TRVector;
begin
  // the greatest (or most positive) parameter values to be considered
  maxP := NewRVector(dimPV);
  maxP[1] := ?;
  ...
  MaximumParameters := maxP
end;

function MinimumParameters : TRVector;
  var
    minP : TRVector;
begin
  // the smallest (or most negative) parameter values to be considered
  minP := NewRVector(dimPV);
```

```
    minP[1] := ?;
    ...
    MinimumParameters := minP
  end;

begin
end.
```

gaSolver

```
// global directives
{$MODE OBJFPC}      // allows objects, classes, interfaces, exception handling
{$H+}               // use ANSI strings
{$IFDEF UNIX}
  {$CALLING CDECL}  // uses the GCC calling convention  - for Unix libraries
  {$PIC ON}         // create Position Independent Code - for Unix libraries
{$ENDIF}

unit gaSolver;  // you may want to rename this to better suit your needs

interface

  uses
    {$IFDEF UNIX}{$IFDEF UseCThreads}
      cthreads,
    {$ENDIF}{$ENDIF}
    Classes, SysUtils, gaData, gaModel;

  // Exported procedures

  procedure RunGeneticAlgorithm (reportName : String);
  // execute the genetic algorithm to fit a model to experimental data

  procedure GetEliteData (out fitness    : Real;
                          var parameters : TRVector;
                          var genome     : TSVector);
  // model parameters that fit the data with the minimum RMSE to the data

  procedure GetModelData (selectExperiment : Integer;             // 1..dimE
                          var xExperiment  : TRMatrix;            // output
                          var yExperiment  : TRMatrix;            // output
                          var xModel       : TRMatrix;            // input
                          var yModel       : TRMatrix);           // output
  // get data pairs for the experimental data and the model fits to these data
  // the experimental matrices are indexed as [variable][sample]
  // the model matrices index as [variable][node] where the node may differ
  // from the sampling points
```

```
implementation

  // Procedures used to manage the genetic algorithm genAlg.

  procedure StartGeneticAlgorithm (numericalModel      : TModel;
                                   var expControlData  : TRData;
                                   var expResponseData : TRData;
                                   var varyParameters  : TBVector;
                                   fixedParameters     : TVectorFn;
                                   var alienParameters : TRVector;
                                   var minParameters   : TRVector;
                                   var maxParameters   : TRVector;
                                   significantFigs     : Integer;
                                   var parameterNames  : TSVector;
                                   reportFileName      : String);
                                   external 'GenAlg';
  { Create a new optimization project and run it through the first generation.
    where
      numericalModel  : the model whose parameters are sought
      expControlData  : the control  values in the experimental data set
      expResponseData : the response values in the experimental data set
      varyParameters  : those parameters that the optimizer will vary
      fixedParameters : values for the fixed parameters - nil if all will vary
      alienParameters : best guess at the values of the varied parameters
      minParameters   : minimum allowed values for the varied parameters
      maxParameters   : maximum allowed values for the varied parameters
      significantFigs : significant figures sought in parameters - 2 to 7
      parameterNames  : array of strings that provide names for the parameters
      reportFileName  : name of the file for writing the report to, less .txt  }

  procedure AdvanceToNextGeneration (out converged : Boolean); external 'GenAlg';
  { Advance the genetic algorithm to its next generation.
    where
      converged : indicates if algorithm has 'theoretically' converged or not  }

  procedure StopGeneticAlgorithm; external 'GenAlg';
  { Terminate the genetic algorithm and close the associated GA report file.   }

  procedure EliteCreatureData (out fitness    : Real;
                               var parameters : TRVector;
                               var genome     : TSVector); external 'GenAlg';
  { Retreive data about the elite creature from the current generation.
    where
      fitness    : value of the elite creature's fitness or quality parameter
                                1  E     RANGE(expY_e)
                   fitness   = - SUM --------------------
                                E e=1 RSME(expY_e, modY_e)
```

```
                E     is  the number of experiments being fit against
                expY  are the response data measured in experiments
                modY  are the response data predicted by the model
    parameters : the model parameters that associate with with this creature
    genome     : an array of chromosomes (represent parameters) whose genes
                 are bits, dominant=1 and recessive=0, written in a string   }

procedure EliteCreatureFits (selectExperiment  : Integer;           // 1..dimE
                             var expControl    : TRMatrix;          // output
                             var expResponse   : TRMatrix;          // output
                             var modelControl  : TRMatrix;          // input
                             var modelResponse : TRMatrix);         // output
                             external 'GenAlg';
{ Retreive XY data that can be used to create graphs of model vs. experiment
  where
    selectExperiment  : choose an experiment for which the model is to be run
    expControl    : x data from the experiments for use in creating graphs
    expResponse   : y data from the experiments for use in creating graphs
    modelControl  : x data from model using the elite creature's parameters
    modelResponse : y data from model using the elite creature's parameters  }

// ---------------------------------------------------------------------------
// ---------------------------------------------------------------------------

// code specific to this application follows below

// ---------------------------------------------------------------------------
// ---------------------------------------------------------------------------

// Exported procedures

procedure RunGeneticAlgorithm (reportName : String);
  var
    alienParam : TRVector;
    converged  : Boolean;
    expCtrl    : TRData;
    expResp    : TRData;
    fixedParam : TVectorFn;
    generation : Integer;
    model      : TModel;
    maxParam   : TRVector;
    minParam   : TRVector;
    names      : TSVector;
    sigFigs    : Integer;
    varyParam  : TBVector;
begin
  // initialize fields
  sigFigs := 6;  // choose a fairly large number to get a larger population
```

```
  // assign field that are imported from gaData and gaModel
  alienParam := AlienParameters;
  expCtrl    := ExperimentalControlData;
  expResp    := ExperimentalResponseData;
  fixedParam := @MyFixedParameters;
  maxParam   := MaximumParameters;
  minParam   := MinimumParameters;
  model      := @MyModel;
  names      := NameParameters;
  varyParam  := VaryParameters;
  WriteLn;
  StartGeneticAlgorithm(model, expCtrl, expResp,
                        varyParam, fixedParam, alienParam, minParam, maxParam,
                        sigFigs, names, reportName);
  Write('Working on generation  1');
  WriteLn('  done');
  generation := 1;
  converged  := False;
  repeat
    Inc(generation);
    if generation < 10 then
      Write('Working on generation  ', generation)
    else
      Write('Working on generation ', generation);
    AdvanceToNextGeneration(converged);
    WriteLn('  done')
  until converged;
  // this final step writes out a report describing the findings from the GA
  StopGeneticAlgorithm
end;

procedure GetEliteData (out fitness    : Real;
                        var parameters : TRVector;
                        var genome     : TSVector);
begin
  EliteCreatureData(fitness, parameters, genome)
end;


procedure GetModelData (selectExperiment : Integer;
                        var xExperiment  : TRMatrix;
                        var yExperiment  : TRMatrix;
                        var xModel       : TRMatrix;
                        var yModel       : TRMatrix);
begin
  EliteCreatureFits(selectExperiment, xExperiment, yExperiment, xModel, yModel)
end;
```

```
begin
end.
```

gaDriver

```
// global directives
{$MODE OBJFPC}      // allows objects, classes, interfaces, exception handling
{$H+}               // use ANSI strings
{$IFDEF UNIX}
  {$CALLING CDECL}  // uses the GCC calling convention  - for Unix libraries
  {$PIC ON}         // create Position Independent Code - for Unix libraries
{$ENDIF}

program gaDriver;

  uses
    {$IFDEF UNIX}{$IFDEF UseCThreads}
      cthreads,
    {$ENDIF}{$ENDIF}
    Classes, SysUtils, gaSolver;

  procedure Run;
    var
      reportName : String;
  begin
    // enter the name of the text file that the report is to be written to
    reportName := '...';
    // run the genetic algorithm
    RunGeneticAlgorithm(reportName)

    (* the following functions are also available to be called

    GetEliteData (var parameters : TRVector; var genome : TSVector);

    GetModelData (selectExperiment : Integer;             // 1..dimE
                  var xExperiment  : TRMatrix;            // output
                  var yExperiment  : TRMatrix;            // output
                  var xModel       : TRMatrix;            // input
                  var yModel       : TRMatrix);           // output         *)
  end;

begin
  Run
end.
```