GenAlg

P. Baillehache

August 28, 2018

Contents

Introduction

GenAlg is a C library providing structures and functions implementing a Genetic Algorithm.

The genes are memorized as a VecFloat and/or VecShort. The user can defined a range of possible values for each gene. The user can define the size of the pool of entities and the size of the breeding pool. Selection, reproduction and mutation are designed to efficiently explore all the possible gene combination, and avoid local optimum. It is also possible to save and load

the GenAlg.

It uses the `PBErr`, `PBMath` and `GSet` libraries.

1 Definitions

A genetic algorithm has 3 steps. In a pool of entities it discards a given number of entities based on their ranking (given by a mean external to the algorithm). Then it replaces each of the discarded entity by a new one created from two selected entities from hte non discarded one. The newly created entity's properties are a mix of these two selected entities, plus a certain amount of random modification. The detail of the implementation in GenAlg of these 3 steps (selection, reproduction and mutation) are given below.

1.1 Selection

The non discarded entities are called 'elite' in GenAlg. The size of the pool of elite is configurable by the user. The selection of two elite entities is simply a random selection in the pool of elites. Selection of the same elite twice is allowed.

1.2 Reproduction

The reproduction step copies the genes of the elite entity into the new entity. Each gene has a probability of 50% to be chosen in one or the other elite.

1.3 Mutation

The mutation occurs as follow. First we calculate the probability of mutation for every gene as follow: $P = \frac{rank}{nbEntity} * (1 - \frac{1}{\sqrt{age+1}})$ where rank is the rank of the discarded entity in the pool of entities, and nbEntity is the number of entities in the pool, and age is the age of the oldest elite entity used during the reproduction step for the entity. A gene affected by a mutation according to this probability is modified as follow. The amplitude of the mutation is equal to $1 - \frac{1}{\sqrt{age+1}}$ where age is the age of the oldest elite entity used during

the reproduction step for the entity. Then the new value of the gene is equals to $gene + range * amp * (rnd + delta)$ where gene is the current value of the gene, range is equal to $max_{gene} - min_{gene}$ (the difference of the maximum allowed value for this gene and its minimum value), amp is the amplitude calculated above, rnd is a random value between -0.5 and 0.5, and delta is the mutation that has been applied to this gene in the corresponding elite entity. Genes' value is kept in bounds by bouncing it on the bounds when necessary ($gene = 2 * bound - gene$)

To counteract inbreeding (the algorithm getting stuck into a local minimum), when the diversity level of the elite pool falls below a threshold, we also reset the adn of all the non entities and all the elite entities (except the best one) having at least one diversity level with another elite entity below the diversity level of the elite pool (set to 0.01 by default). The diversity level of the whole elite pool is calculated as follow $Avg_{i,j} \frac{||\overrightarrow{adn}(elite_i) - \overrightarrow{adn}(elite_j)||}{||\overrightarrow{bound}_{max} - \overrightarrow{bound}_{min}||}$ where $\overrightarrow{adn}(elite_i)$ is the genes vector of the i-th elite entity, and $\overrightarrow{bound}_{max}$ and $\overrightarrow{bound}_{min}$ are the vector of maximum and minimum values of the genes.

Some explanation: delta bias the mutation toward the direction that improved the result at previous step; in the pool of discarded entities high ranked ones tend to have few mutations and low ranked ones tend to have more mutation, this tends to cover any posibilities of evolution; entities newly entered in the elite pool tends to produce new entities near to them (in term of distance in the genes space), while older ones tend to produce more diverse new entities, thus the exploration of solution space occurs from the vicinity of newly better solutions toward larger areas; from the previous point, a good entity tends to create a lot of similar entity, which may lead to an elite pool saturated with very similar entities (inbreeding) from which the algorithm can't escape, this is prevented by the forced mutation of elites when the inbreeding level gets too high.

2 Interface

```
// ============ GENALG.H ================

#ifndef GENALG_H
#define GENALG_H

// ================= Include =================

#include <stdlib.h>
```

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdbool.h>
#include "pberr.h"
#include "pbmath.h"
#include "gset.h"

// ================= Define ==================

#define GENALG_NBENTITIES 100
#define GENALG_NBELITES 20

// ------------- GenAlgAdn

// ================= Data structure ==================

typedef struct GenAlg GenAlg;

typedef struct GenAlgAdn {
  // ID
  unsigned long _id;
  // Age
  unsigned long _age;
  // Adn for floating point value
  VecFloat* _adnF;
  // Delta Adn during mutation for floating point value
  VecFloat* _deltaAdnF;
  // Adn for integer point value
  VecLong* _adnI;
  // Value
  float _val;
} GenAlgAdn;

// ================ Functions declaration ====================

// Create a new GenAlgAdn with ID 'id', 'lengthAdnF' and 'lengthAdnI'
// 'lengthAdnF' and 'lengthAdnI' must be greater than or equal to 0
GenAlgAdn* GenAlgAdnCreate(const unsigned long id, const long lengthAdnF,
  const long lengthAdnI);

// Free memory used by the GenAlgAdn 'that'
void GenAlgAdnFree(GenAlgAdn** that);

// Return the adn for floating point values of the GenAlgAdn 'that'
#if BUILDMODE != 0
inline
#endif
const VecFloat* GAAdnAdnF(const GenAlgAdn* const that);

// Return the delta of adn for floating point values of the
// GenAlgAdn 'that'
#if BUILDMODE != 0
inline
#endif
const VecFloat* GAAdnDeltaAdnF(const GenAlgAdn* const that);

// Return the adn for integer values of the GenAlgAdn 'that'
#if BUILDMODE != 0
inline
#endif
VecLong* GAAdnAdnI(const GenAlgAdn* const that);
```

```
// Initialise randomly the genes of the GenAlgAdn 'that' of the
// GenAlg 'ga' according to the type of the GenAlg
void GAAdnInit(const GenAlgAdn* const that, const GenAlg* ga);

// Initialise randomly the genes of the GenAlgAdn 'that' of the
// GenAlg 'ga', version used to calculate the parameters of a NeuraNet
void GAAdnInitNeuraNet(const GenAlgAdn* const that, const GenAlg* ga);

// Get the 'iGene'-th gene of the adn for floating point values of the
// GenAlgAdn 'that'
#if BUILDMODE != 0
inline
#endif
float GAAdnGetGeneF(const GenAlgAdn* const that, const long iGene);

// Get the delta of the 'iGene'-th gene of the adn for floating point
// values of the GenAlgAdn 'that'
#if BUILDMODE != 0
inline
#endif
float GAAdnGetDeltaGeneF(const GenAlgAdn* const that, const long iGene);

// Get the 'iGene'-th gene of the adn for int values of the
// GenAlgAdn 'that'
#if BUILDMODE != 0
inline
#endif
int GAAdnGetGeneI(const GenAlgAdn* const that, const long iGene);

// Set the 'iGene'-th gene of the adn for floating point values of the
// GenAlgAdn 'that' to 'gene'
#if BUILDMODE != 0
inline
#endif
void GAAdnSetGeneF(GenAlgAdn* const that, const long iGene,
  const float gene);

// Set the delta of the 'iGene'-th gene of the adn for floating point
// values of the GenAlgAdn 'that' to 'delta'
#if BUILDMODE != 0
inline
#endif
void GAAdnSetDeltaGeneF(GenAlgAdn* const that, const long iGene,
  const float delta);

// Set the 'iGene'-th gene of the adn for int values of the
// GenAlgAdn 'that'to 'gene'
#if BUILDMODE != 0
inline
#endif
void GAAdnSetGeneI(GenAlgAdn* const that, const long iGene,
  const long gene);

// Get the id of the GenAlgAdn 'that'
#if BUILDMODE != 0
inline
#endif
unsigned long GAAdnGetId(const GenAlgAdn* const that);

// Get the age of the GenAlgAdn 'that'
#if BUILDMODE != 0
```

```
inline
#endif
unsigned long GAAdnGetAge(const GenAlgAdn* const that);

// Get the value of the GenAlgAdn 'that'
#if BUILDMODE != 0
inline
#endif
float GAAdnGetVal(const GenAlgAdn* const that);

// Print the information about the GenAlgAdn 'that' on the
// stream 'stream'
void GAAdnPrintln(const GenAlgAdn* const that, FILE* const stream);

// Return true if the GenAlgAdn 'that' is new, i.e. is age equals 1
// Return false
#if BUILDMODE != 0
inline
#endif
bool GAAdnIsNew(const GenAlgAdn* const that);

// Copy the GenAlgAdn 'tho' into the GenAlgAdn 'that'
#if BUILDMODE != 0
inline
#endif
void GAAdnCopy(GenAlgAdn* const that, const GenAlgAdn* const tho);

// ------------- GenAlg

// ================= Define ==================

#define GABestAdnF(that) GAAdnAdnF(GABestAdn(that))
#define GABestAdnI(that) GAAdnAdnI(GABestAdn(that))

// ================= Data structure ==================

typedef enum GenAlgType {
  genAlgTypeDefault,
  genAlgTypeNeuraNet,
  genAlgTypeNeuraNetConv
} GenAlgType;

// Data used when GenAlg is applied to a NeuraNet
typedef struct GANeuraNet {
  // Nb of input, hidden and output of the NeuraNet
  int _nbIn;
  int _nbHid;
  int _nbOut;
  long _nbBaseConv;
  long _nbBaseCellConv;
} GANeuraNet;

typedef struct GenAlg {
  // GSet of GenAlgAdn, sortval == score so the head of the set is the
  // worst adn and the tail of the set is the best
  GSet* _adns;
  // Copy of the best adn
  GenAlgAdn* _bestAdn;
  // Type of the GenAlg
  GenAlgType _type;
  // Current epoch
  unsigned long _curEpoch;
```

```
  // Nb elite entities in population
  int _nbElites;
  // Id of the next new GenAlgAdn
  unsigned long _nextId;
  // Length of adn for floating point value
  const long _lengthAdnF;
  // Length of adn for integer value
  const long _lengthAdnI;
  // Bounds (min, max) for floating point values adn
  VecFloat2D* _boundsF;
  // Bounds (min, max) for integer values adn
  VecLong2D* _boundsI;
  // Norm of the range value for adns (optimization for diversity
  // calculation)
  float _normRangeFloat;
  float _normRangeInt;
  // Data used if the GenAlg is applied to a NeuraNet
  GANeuraNet _NNdata;
  // Number of ktevent
  unsigned long _nbKTEvent;
} GenAlg;

// ================ Functions declaration ====================

// Create a new GenAlg with 'nbEntities', 'nbElites', 'lengthAdnF'
// and 'lengthAdnI'
// 'nbEntities' must greater than 2
// 'nbElites' must greater than 1
// 'lengthAdnF' and 'lengthAdnI' must be greater than or equal to 0
GenAlg* GenAlgCreate(const int nbEntities, const int nbElites,
  const long lengthAdnF, const long lengthAdnI);

// Free memory used by the GenAlg 'that'
void GenAlgFree(GenAlg** that);

// Get the type of the GenAlg 'that'
#if BUILDMODE != 0
inline
#endif
GenAlgType GAGetType(const GenAlg* const that);

// Set the type of the GenAlg 'that' to genAlgTypeNeuraNet, the GenAlg
// will be used with a NeuraNet having 'nbIn' inputs, 'nbHid' hidden
// values and 'nbOut' outputs
#if BUILDMODE != 0
inline
#endif
void GASetTypeNeuraNet(GenAlg* const that, const int nbIn,
  const int nbHid, const int nbOut);

// Set the type of the GenAlg 'that' to genAlgTypeNeuraNetConv,
// the GenAlg will be used with a NeuraNet having 'nbIn' inputs,
// 'nbHid' hidden values, 'nbOut' outputs, 'nbBaseConv' bases function
// dedicated to the convolution and 'nbBaseCellConv' bases function per cell of convolution
#if BUILDMODE != 0
inline
#endif
void GASetTypeNeuraNetConv(GenAlg* const that, const int nbIn,
  const int nbHid, const int nbOut, const long nbBaseConv,
  const long nbBaseCellConv);

// Return the GSet of the GenAlg 'that'
```

```
#if BUILDMODE != 0
inline
#endif
GSet* GAAdns(const GenAlg* const that);

// Return the nb of entities of the GenAlg 'that'
#if BUILDMODE != 0
inline
#endif
int GAGetNbAdns(const GenAlg* const that);

// Return the nb of elites of the GenAlg 'that'
#if BUILDMODE != 0
inline
#endif
int GAGetNbElites(const GenAlg* const that);

// Return the current epoch of the GenAlg 'that'
#if BUILDMODE != 0
inline
#endif
unsigned long GAGetCurEpoch(const GenAlg* const that);

// Return the number of KTEvent of the GenAlg 'that'
#if BUILDMODE != 0
inline
#endif
unsigned long GAGetNbKTEvent(const GenAlg* const that);

// Set the nb of entities of the GenAlg 'that' to 'nb'
// 'nb' must be greater than 1, if 'nb' is lower than the current nb
// of elite the number of elite is set to 'nb' - 1
void GASetNbEntities(GenAlg* const that, const int nb);

// Set the nb of elites of the GenAlg 'that' to 'nb'
// 'nb' must be greater than 0, if 'nb' is greater or equal to the
// current nb of entities the number of entities is set to 'nb' + 1
void GASetNbElites(GenAlg* const that, const int nb);

// Get the length of adn for floating point value
#if BUILDMODE != 0
inline
#endif
long GAGetLengthAdnFloat(const GenAlg* const that);

// Get the length of adn for integer value
#if BUILDMODE != 0
inline
#endif
long GAGetLengthAdnInt(const GenAlg* const that);

// Get the bounds for the 'iGene'-th gene of adn for floating point
// values
#if BUILDMODE != 0
inline
#endif
const VecFloat2D* GABoundsAdnFloat(const GenAlg* const that,
  const long iGene);

// Get the bounds for the 'iGene'-th gene of adn for integer values
#if BUILDMODE != 0
inline
```

```
#endif
const VecLong2D* GABoundsAdnInt(const GenAlg* const that,
  const long iGene);

// Set the bounds for the 'iGene'-th gene of adn for floating point
// values to a copy of 'bounds'
#if BUILDMODE != 0
inline
#endif
void GASetBoundsAdnFloat(GenAlg* const that, const long iGene,
  const VecFloat2D* const bounds);

// Set the bounds for the 'iGene'-th gene of adn for integer values
// to a copy of 'bounds'
#if BUILDMODE != 0
inline
#endif
void GASetBoundsAdnInt(GenAlg* const that, const long iGene,
  const VecLong2D* bounds);

// Get the GenAlgAdn of the GenAlg 'that' currently at rank 'iRank'
#if BUILDMODE != 0
inline
#endif
GenAlgAdn* GAAdn(const GenAlg* const that, const int iRank);

// Init the GenAlg 'that'
// Must be called after the bounds have been set
// The random generator must have been initialised before calling this
// function
void GAInit(GenAlg* const that);

// Step an epoch for the GenAlg 'that' with the current ranking of
// GenAlgAdn
void GAStep(GenAlg* const that);

// Print the information about the GenAlg 'that' on the stream 'stream'
void GAPrintln(const GenAlg* const that, FILE* const stream);

// Print a summary about the elite entities of the GenAlg 'that'
// on the stream 'stream'
void GAEliteSummaryPrintln(const GenAlg* const that,
  FILE* const stream);

// Get the diversity of the GenAlg 'that'
#if BUILDMODE != 0
inline
#endif
float GAGetDiversity(const GenAlg* const that);

// Function which return the JSON encoding of 'that'
JSONNode* GAEncodeAsJSON(const GenAlg* const that);

// Function which decode from JSON encoding 'json' to 'that'
bool GADecodeAsJSON(GenAlg** that, const JSONNode* const json);

// Load the GenAlg 'that' from the stream 'stream'
// If the GenAlg is already allocated, it is freed before loading
// Return true in case of success, else false
bool GALoad(GenAlg** that, FILE* const stream);

// Save the GenAlg 'that' to the stream 'stream'
```

```
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true in case of success, else false
bool GASave(const GenAlg* const that, FILE* const stream,
  const bool compact);

// Set the value of the GenAlgAdn 'adn' of the GenAlg 'that' to 'val'
#if BUILDMODE != 0
inline
#endif
void GASetAdnValue(GenAlg* const that, GenAlgAdn* const adn,
  const float val);

// Update the norm of the range value for adans of the GenAlg 'that'
void GAUpdateNormRange(GenAlg* const that);

// Reset the GenAlg 'that'
// Randomize all the gene except those of the first adn
void GAKTEvent(GenAlg* const that);

// Return the best adn of the GenAlg 'that'
#if BUILDMODE != 0
inline
#endif
const GenAlgAdn* GABestAdn(const GenAlg* const that);

// ================= Polymorphism ==================

// ================ Inliner ====================

#if BUILDMODE != 0
#include "genalg-inline.c"
#endif


#endif
```

3 Code

3.1 genalg.c

```
// ============ GENALG.C ================

// ================= Include =================

#include "genalg.h"
#if BUILDMODE == 0
#include "genalg-inline.c"
#endif

// ------------- GenAlgAdn

// ================ Functions declaration ====================

// Get the diversity value of 'adnA' against 'adnB'
// The diversity is equal to
float GAAdnGetDiversity(const GenAlgAdn* const adnA,
  const GenAlgAdn* const adnB, const GenAlg* const ga);
```

```
// Initialise randomly the genes of the GenAlgAdn 'that' of the
// GenAlg 'ga'
void GAAdnInitDefault(const GenAlgAdn* const that, const GenAlg* ga);

// Initialise randomly the genes of the GenAlgAdn 'that' of the
// GenAlg 'ga', version used to calculate the parameters of a NeuraNet
void GAAdnInitNeuraNet(const GenAlgAdn* const that, const GenAlg* ga);

// Initialise randomly the genes of the GenAlgAdn 'that' of the
// GenAlg 'ga', version used to calculate the parameters of a NeuraNet
// with convolution
void GAAdnInitNeuraNetConv(const GenAlgAdn* const that,
  const GenAlg* const ga);

// ================ Functions implementation ====================

// Create a new GenAlgAdn with ID 'id', 'lengthAdnF' and 'lengthAdnI'
// 'lengthAdnF' and 'lengthAdnI' must be greater than or equal to 0
GenAlgAdn* GenAlgAdnCreate(const unsigned long id,
  const long lengthAdnF, const long lengthAdnI) {
#if BUILDMODE == 0
  if (lengthAdnF < 0) {
    GenAlgErr->_type = PBErrTypeInvalidArg;
    sprintf(GenAlgErr->_msg, "'lengthAdnF' is invalid (%ld>=0)",
      lengthAdnF);
    PBErrCatch(GenAlgErr);
  }
  if (lengthAdnI < 0) {
    GenAlgErr->_type = PBErrTypeInvalidArg;
    sprintf(GenAlgErr->_msg, "'lengthAdnI' is invalid (%ld>=0)",
      lengthAdnI);
    PBErrCatch(GenAlgErr);
  }
#endif
  // Allocate memory
  GenAlgAdn* that = PBErrMalloc(GenAlgErr, sizeof(GenAlgAdn));
  // Set the properties
  that->_age = 1;
  that->_id = id;
  that->_val = 0.0;
  if (lengthAdnF > 0) {
    that->_adnF = VecFloatCreate(lengthAdnF);
    that->_deltaAdnF = VecFloatCreate(lengthAdnF);
  } else {
    that->_adnF = NULL;
    that->_deltaAdnF = NULL;
  }
  if (lengthAdnI > 0)
    that->_adnI = VecLongCreate(lengthAdnI);
  else
    that->_adnI = NULL;
  // Return the new GenAlgAdn
  return that;
}

// Free memory used by the GenAlgAdn 'that'
void GenAlgAdnFree(GenAlgAdn** that) {
  // Check the argument
  if (that == NULL || *that == NULL) return;
  // Free memory
  if ((*that)->_adnF != NULL)
```

```
    VecFree(&((*that)->_adnF));
  if ((*that)->_deltaAdnF != NULL)
    VecFree(&((*that)->_deltaAdnF));
  if ((*that)->_adnI != NULL)
    VecFree(&((*that)->_adnI));
  free(*that);
  // Set the pointer to null
  *that = NULL;
}

// Initialise randomly the genes of the GenAlgAdn 'that' of the
// GenAlg 'ga' according to the type of GenAlg
void GAAdnInit(const GenAlgAdn* const that, const GenAlg* const ga) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  switch (GAGetType(ga)) {
    case genAlgTypeNeuraNet:
      GAAdnInitNeuraNet(that, ga);
      break;
    case genAlgTypeNeuraNetConv:
      GAAdnInitNeuraNetConv(that, ga);
      break;
    case genAlgTypeDefault:
    default:
      GAAdnInitDefault(that, ga);
  }
}

// Initialise randomly the genes of the GenAlgAdn 'that' of the
// GenAlg 'ga'
void GAAdnInitDefault(const GenAlgAdn* const that,
  const GenAlg* const ga) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  // For each floating point value gene
  for (long iGene = GAGetLengthAdnFloat(ga); iGene--;) {
    float min = VecGet(GABoundsAdnFloat(ga, iGene), 0);
    float max = VecGet(GABoundsAdnFloat(ga, iGene), 1);
    float val = min + (max - min) * rnd();
    VecSet(that->_adnF, iGene, val);
  }
  // For each integer value gene
  for (long iGene = GAGetLengthAdnInt(ga); iGene--;) {
    long min = VecGet(GABoundsAdnInt(ga, iGene), 0);
    long max = VecGet(GABoundsAdnInt(ga, iGene), 1);
    long val = (long)round((float)min + (float)(max - min) * rnd());
    VecSet(that->_adnI, iGene, val);
  }
}

// Initialise randomly the genes of the GenAlgAdn 'that' of the
// GenAlg 'ga', version used to calculate the parameters of a NeuraNet
```

```
// with convolution
void GAAdnInitNeuraNetConv(const GenAlgAdn* const that,
  const GenAlg* const ga) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  // For each floating point value gene
  for (long iGene = GAGetLengthAdnFloat(ga); iGene--;) {
    float min = VecGet(GABoundsAdnFloat(ga, iGene), 0);
    float max = VecGet(GABoundsAdnFloat(ga, iGene), 1);
    float val = min + (max - min) * rnd();
    VecSet(that->_adnF, iGene, val);
  }
}

// Initialise randomly the genes of the GenAlgAdn 'that' of the
// GenAlg 'ga', version used to calculate the parameters of a NeuraNet
void GAAdnInitNeuraNet(const GenAlgAdn* const that, const GenAlg* ga) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  // Init the base functions randomly
  // For each floating point value gene
  for (long iGene = GAGetLengthAdnFloat(ga); iGene--;) {
    float min = VecGet(GABoundsAdnFloat(ga, iGene), 0);
    float max = VecGet(GABoundsAdnFloat(ga, iGene), 1);
    float val = min + (max - min) * rnd();
    VecSet(that->_adnF, iGene, val);
  }
  // Init the links by ensuring there is at least one link reaching
  // each output and use inputs as start of the initial links
  // For each integer value gene
  int shiftOut = ga->_NNdata._nbIn + ga->_NNdata._nbHid;
  for (long iGene = 0; iGene < GAGetLengthAdnInt(ga); iGene += 3) {
    VecSet(that->_adnI, iGene, -1);
  }
  for (int iOut = 0; iOut < ga->_NNdata._nbOut; ++iOut) {
    // The base function is randomly choosen but can't be an
    // inactive link
    long min = 0;
    long max = VecGet(GABoundsAdnInt(ga, iOut * 3), 1);
    long val = (long)round((float)min + (float)(max - min) * rnd());
    VecSet(that->_adnI, iOut * 3, val);
    // The start of the link is randomly choosen amongst inputs
    min = 0;
    max = ga->_NNdata._nbIn - 1;
    val = (long)round((float)min + (float)(max - min) * rnd());
    VecSet(that->_adnI, iOut * 3 + 1, val);
    // The end of the link is choosen sequencially amongst outputs
    VecSet(that->_adnI, iOut * 3 + 2, iOut + shiftOut);
  }
}

// Print the information about the GenAlgAdn 'that' on the
```

```
// stream 'stream'
void GAAdnPrintln(const GenAlgAdn* const that, FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (stream == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'stream' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  fprintf(stream, "id:%lu age:%lu", GAAdnGetId(that), GAAdnGetAge(that));
  fprintf(stream, "\n");
  fprintf(stream, "  adnF:");
  VecFloatPrint(GAAdnAdnF(that), stream,6);
  fprintf(stream, "\n");
  fprintf(stream, "  deltaAdnF:");
  VecFloatPrint(GAAdnDeltaAdnF(that), stream,6);
  fprintf(stream, "\n");
  fprintf(stream, "  adnI:");
  VecPrint(GAAdnAdnI(that), stream);
  fprintf(stream, "\n");
}

// ------------- GenAlg

// ================ Functions declaration ====================

// Select the rank of two parents for the SRM algorithm
// Return the ranks in 'parents', with parents[0] <= parents[1]
void GASelectParents(const GenAlg* const that, int* const parents);

// Set the genes of the entity at rank 'iChild' as a 50/50 mix of the
// genes of entities at ranks 'parents[0]' and 'parents[1]'
void GAReproduction(GenAlg* const that, const int* const parents,
  const int iChild);

// Set the genes of the entity at rank 'iChild' as a 50/50 mix of the
// genes of entities at ranks 'parents[0]' and 'parents[1]'
void GAReproductionDefault(GenAlg* const that,
  const int* const parents, const int iChild);

// Set the genes of the adn at rank 'iChild' as a 50/50 mix of the
// genes of adns at ranks 'parents[0]' and 'parents[1]'
// This version is optimised to calculate the parameters of a NeuraNet
// with convolution by inheriting whole bases from parents
void GAReproductionNeuraNetConv(GenAlg* const that,
  const int* const parents, const int iChild);

// Set the genes of the entity at rank 'iChild' as a 50/50 mix of the
// genes of entities at ranks 'parents[0]' and 'parents[1]'
// This version is optimised to calculate the parameters of a NeuraNet
// by inheriting whole bases and links from parents
void GAReproductionNeuraNet(GenAlg* const that,
  const int* const parents, const int iChild);

// Router toward the appropriate Mute function according to the type
// of GenAlg
void GAMute(GenAlg* const that, const int* const parents,
```

```
  const int iChild);

// Mute the genes of the entity at rank 'iChild'
void GAMuteDefault(GenAlg* const that, const int* const parents,
  const int iChild);

// Mute the genes of the entity at rank 'iChild'
// This version is optimised to calculate the parameters of a NeuraNet
// by ensuring coherence in links: outputs have at least one link
// and there is no dead link
void GAMuteNeuraNet(GenAlg* const that, const int* const parents,
  const int iChild);

// Mute the genes of the entity at rank 'iChild'
// This version is optimised to calculate the parameters of a NeuraNet
// with convolution by muting bases function per cell
void GAMuteNeuraNetConv(GenAlg* const that, const int* const parents,
  const int iChild);

// ================ Functions implementation ====================

// Create a new GenAlg with 'nbEntities', 'nbElites', 'lengthAdnF'
// and 'lengthAdnI'
// 'nbEntities' must greater than 2
// 'nbElites' must greater than 1
// 'lengthAdnF' and 'lengthAdnI' must be greater than or equal to 0
GenAlg* GenAlgCreate(const int nbEntities, const int nbElites,
  const long lengthAdnF, const long lengthAdnI) {
  // Allocate memory
  GenAlg* that = PBErrMalloc(GenAlgErr, sizeof(GenAlg));
  // Set the properties
  that->_type = genAlgTypeDefault;
  that->_adns = GSetCreate();
  that->_curEpoch = 0;
  that->_nbKTEvent = 0;
  that->_bestAdn = GenAlgAdnCreate(0, lengthAdnF, lengthAdnI);
  *(long*)&(that->_lengthAdnF) = lengthAdnF;
  *(long*)&(that->_lengthAdnI) = lengthAdnI;
  if (lengthAdnF > 0) {
    that->_boundsF =
      PBErrMalloc(GenAlgErr, sizeof(VecFloat2D) * lengthAdnF);
    for (long iGene = lengthAdnF; iGene--;)
      that->_boundsF[iGene] = VecFloatCreateStatic2D();
  } else
    that->_boundsF = NULL;
  if (lengthAdnI > 0) {
    that->_boundsI =
      PBErrMalloc(GenAlgErr, sizeof(VecLong2D) * lengthAdnI);
    for (long iGene = lengthAdnI; iGene--;)
      that->_boundsI[iGene] = VecLongCreateStatic2D();
  } else
    that->_boundsI = NULL;
  that->_normRangeFloat = 1.0;
  that->_normRangeInt = 1.0;
  that->_nbElites = 0;
  that->_nextId = 0;
  GASetNbEntities(that, nbEntities);
  GASetNbElites(that, nbElites);
  // Return the new GenAlg
  return that;
}
```

```
// Free memory used by the GenAlg 'that'
void GenAlgFree(GenAlg** that) {
  // Check the argument
  if (that == NULL || *that == NULL) return;
  // Free memory
  GSetIterForward iter = GSetIterForwardCreateStatic(GAAdns(*that));
  do {
    GenAlgAdn* gaEnt = GSetIterGet(&iter);
    GenAlgAdnFree(&gaEnt);
  } while (GSetIterStep(&iter));
  GSetFree(&((*that)->_adns));
  if ((*that)->_boundsF != NULL)
    free((*that)->_boundsF);
  if ((*that)->_boundsI != NULL)
    free((*that)->_boundsI);
  GenAlgAdnFree(&((*that)->_bestAdn));
  free(*that);
  // Set the pointer to null
  *that = NULL;
}

// Set the nb of entities of the GenAlg 'that' to 'nb'
// 'nb' must be greater than 1, if 'nb' is lower than the current nb
// of elite the number of elite is set to 'nb' - 1
void GASetNbEntities(GenAlg* const that, const int nb) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (nb <= 1) {
    GenAlgErr->_type = PBErrTypeInvalidArg;
    sprintf(GenAlgErr->_msg, "'nb' is invalid (%d>1)", nb);
    PBErrCatch(GenAlgErr);
  }
#endif
  while (GSetNbElem(GAAdns(that)) > nb) {
    GenAlgAdn* gaEnt = GSetPop(GAAdns(that));
    GenAlgAdnFree(&gaEnt);
  }
  while (GSetNbElem(GAAdns(that)) < nb) {
    GenAlgAdn* ent = GenAlgAdnCreate(that->_nextId++,
      GAGetLengthAdnFloat(that), GAGetLengthAdnInt(that));
    GSetPush(GAAdns(that), ent);
  }
  if (GAGetNbElites(that) >= nb)
    GASetNbElites(that, nb - 1);
}

// Set the nb of elites of the GenAlg 'that' to 'nb'
// 'nb' must be greater than 0, if 'nb' is greater or equal to the
// current nb of entities the number of entities is set to 'nb' + 1
void GASetNbElites(GenAlg* const that, const int nb) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (nb <= 1) {
    GenAlgErr->_type = PBErrTypeInvalidArg;
```

```
    sprintf(GenAlgErr->_msg, "'nb' is invalid (%d>1)", nb);
    PBErrCatch(GenAlgErr);
  }
#endif
  if (GAGetNbAdns(that) <= nb)
    GASetNbEntities(that, nb + 1);
  that->_nbElites = nb;
}

// Init the GenAlg 'that'
// Must be called after the bounds have been set
// The random generator must have been initialised before calling this
// function
void GAInit(GenAlg* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  // For each adn
  GSetIterForward iter = GSetIterForwardCreateStatic(GAAdns(that));
  do {
    // Get the adn
    GenAlgAdn* adn = GSetIterGet(&iter);
    // Initialise randomly the genes of the adn
    GAAdnInit(adn, that);
  } while (GSetIterStep(&iter));
  GAAdnCopy(that->_bestAdn, GAAdn(that, 0));
}

// Reset the GenAlg 'that'
// Randomize all the gene except those of the best adn
void GAKTEvent(GenAlg* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  ++(that->_nbKTEvent);
  GenAlgAdn* adn = GAAdn(that, 0);
  unsigned long int age = adn->_age;
  GAAdnCopy(adn, GABestAdn(that));
  adn->_age = age;
  int parents[2] = {0};
  GAMute(that, parents, 0);
  adn->_age = 1;
  for (int iEnt = 1; iEnt < GAGetNbAdns(that);
    ++iEnt) {
    GenAlgAdn* adn = GAAdn(that, iEnt);
    GAAdnInit(adn, that);
    adn->_age = 1;
    adn->_id = (that->_nextId)++;
  }
}

// Step an epoch for the GenAlg 'that' with the current ranking of
// GenAlgAdn
void GAStep(GenAlg* const that) {
```

```
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  // Selection, Reproduction, Mutation
  // Ensure the set of adns is sorted
  GSetSort(GAAdns(that));
  // Update the best adn if necessary
  if (that->_curEpoch == 1 ||
    GAAdnGetVal(GAAdn(that, 0)) > GAAdnGetVal(GABestAdn(that))) {
    GAAdnCopy(that->_bestAdn, GAAdn(that, 0));
  }
  // Declare a variable to memorize the parents
  int parents[2];
  // Get the diversity level
  float diversity = GAGetDiversity(that);
  // Correct the diversity level with the age of the best adn
  //diversity *=
    //1.0 - fsquare((float)(GAAdnGetAge(GAAdn(that, 0))) / 1000.0);
  // If the diversity level is too low
  if (diversity < PBMATH_EPSILON || GAAdnGetAge(GAAdn(that, 0)) > 200) {
    // Renew diversity by applying a KT event (in memory of
    // chickens' grand pa and grand ma)
    GAKTEvent(that);
  // Else, the diversity level is ok
  } else {
    // For each adn which is an elite
    for (int iAdn = 0; iAdn < GAGetNbElites(that); ++iAdn) {
      // Increment age
      ++(GAAdn(that, iAdn)->_age);
    }
    // For each adn which is not an elite
    for (int iAdn = GAGetNbElites(that); iAdn < GAGetNbAdns(that);
      ++iAdn) {
      // Select two parents for this adn
      GASelectParents(that, parents);
      // Set the genes of the adn as a 50/50 mix of parents' genes
      GAReproduction(that, parents, iAdn);
      // Mute the genes of the adn
      GAMute(that, parents, iAdn);
    }
  }
  // Increment the number of epochs
  ++(that->_curEpoch);
}

// Select the rank of two parents for the SRM algorithm
// Return the ranks in 'parents', with parents[0] <= parents[1]
void GASelectParents(const GenAlg* const that, int* const parents) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (parents == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'parents' is null");
    PBErrCatch(GenAlgErr);
```

```
  }
#endif
  // Declare a variable to memorize the parents' rank
  int p[2];
  do {
    for (int i = 2; i--;)
      // p[i] below may be equal to the rank of the highest non elite
      // adn, but it's not a problem so leave it and let's call that
      // the Hawking radiation of this function in memory of this great
      // man.
      p[i] = (int)floor(rnd() * (float)GAGetNbElites(that));
  } while (p[0] == p[1]);
  // Memorize the sorted parents' rank
  if (p[0] < p[1]) {
    parents[0] = p[0];
    parents[1] = p[1];
  } else {
    parents[0] = p[1];
    parents[1] = p[0];
  }
}

// Set the genes of the adn at rank 'iChild' as a 50/50 mix of the
// genes of adns at ranks 'parents[0]' and 'parents[1]'
void GAReproduction(GenAlg* const that,
  const int* const parents, const int iChild) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (parents == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'parents' is null");
    PBErrCatch(GenAlgErr);
  }
  if (iChild < 0 || iChild >= GAGetNbAdns(that)) {
    GenAlgErr->_type = PBErrTypeInvalidArg;
    sprintf(GenAlgErr->_msg, "'child' is invalid (0<=%d<%d)",
      iChild, GAGetNbAdns(that));
    PBErrCatch(GenAlgErr);
  }
#endif
  switch (GAGetType(that)) {
    case genAlgTypeNeuraNet:
      GAReproductionNeuraNet(that, parents, iChild);
      break;
    case genAlgTypeNeuraNetConv:
      GAReproductionNeuraNetConv(that, parents, iChild);
      break;
    case genAlgTypeDefault:
    default:
      GAReproductionDefault(that, parents, iChild);
  }
}

// Set the genes of the adn at rank 'iChild' as a 50/50 mix of the
// genes of adns at ranks 'parents[0]' and 'parents[1]'
// This version is optimised to calculate the parameters of a NeuraNet
// by inheriting whole bases and links from parents
void GAReproductionNeuraNet(GenAlg* const that,
```

```
  const int* const parents, const int iChild) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (parents == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'parents' is null");
    PBErrCatch(GenAlgErr);
  }
  if (iChild < 0 || iChild >= GAGetNbAdns(that)) {
    GenAlgErr->_type = PBErrTypeInvalidArg;
    sprintf(GenAlgErr->_msg, "'child' is invalid (0<=%d<%d)",
      iChild, GAGetNbAdns(that));
    PBErrCatch(GenAlgErr);
  }
#endif
  // Get the parents and child
  GenAlgAdn* parentA = GAAdn(that, parents[0]);
  GenAlgAdn* parentB = GAAdn(that, parents[1]);
  GenAlgAdn* child = GAAdn(that, iChild);
  // For each gene of the adn for floating point value
  for (long iGene = 0; iGene < GAGetLengthAdnFloat(that); iGene += 3) {
    // Get the gene from one parent or the other with equal
    // probabililty
    if (rnd() < 0.5) {
      for (long jGene = 3; jGene--;) {
        VecSet(child->_adnF, iGene + jGene,
          VecGet(parentA->_adnF, iGene + jGene));
        VecSet(child->_deltaAdnF, iGene + jGene,
          VecGet(parentA->_deltaAdnF, iGene + jGene));
      }
    } else {
      for (long jGene = 3; jGene--;) {
        VecSet(child->_adnF, iGene + jGene,
          VecGet(parentB->_adnF, iGene + jGene));
        VecSet(child->_deltaAdnF, iGene + jGene,
          VecGet(parentB->_deltaAdnF, iGene + jGene));
      }
    }
  }
  // For each gene of the adn for int value
  for (long iGene = 0; iGene < GAGetLengthAdnInt(that); iGene += 3) {
    // Get the gene from one parent or the other with equal probabililty
    if (rnd() < 0.5) {
      for (long jGene = 3; jGene--;)
        VecSet(child->_adnI, iGene + jGene,
          VecGet(parentA->_adnI, iGene + jGene));
    } else {
      for (long jGene = 3; jGene--;)
        VecSet(child->_adnI, iGene + jGene,
          VecGet(parentB->_adnI, iGene + jGene));
    }
  }
  // Reset the age of the child
  child->_age = 1;
  // Set the id of the child
  child->_id = (that->_nextId)++;
}
```

```
// Set the genes of the adn at rank 'iChild' as a 50/50 mix of the
// genes of adns at ranks 'parents[0]' and 'parents[1]'
// This version is optimised to calculate the parameters of a NeuraNet
// with convolution by inheriting whole bases from parents
void GAReproductionNeuraNetConv(GenAlg* const that,
  const int* const parents, const int iChild) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (parents == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'parents' is null");
    PBErrCatch(GenAlgErr);
  }
  if (iChild < 0 || iChild >= GAGetNbAdns(that)) {
    GenAlgErr->_type = PBErrTypeInvalidArg;
    sprintf(GenAlgErr->_msg, "'child' is invalid (0<=%d<%d)",
      iChild, GAGetNbAdns(that));
    PBErrCatch(GenAlgErr);
  }
#endif
  // Get the parents and child
  GenAlgAdn* parentA = GAAdn(that, parents[0]);
  GenAlgAdn* parentB = GAAdn(that, parents[1]);
  GenAlgAdn* child = GAAdn(that, iChild);
  // For each gene of the adn for floating point value of convolution
  // base functions
  for (long iGene = 0;
    iGene < that->_NNdata._nbBaseConv * 3;
    iGene += that->_NNdata._nbBaseCellConv * 3) {
    // Get the gene from one parent or the other with equal probabililty
    if (rnd() < 0.5) {
      for (long jGene = that->_NNdata._nbBaseCellConv * 3;
        jGene--;) {
        VecSet(child->_adnF, iGene + jGene,
          VecGet(parentA->_adnF, iGene + jGene));
        VecSet(child->_deltaAdnF, iGene + jGene,
          VecGet(parentA->_deltaAdnF, iGene + jGene));
      }
    } else {
      for (long jGene = that->_NNdata._nbBaseCellConv * 3;
        jGene--;) {
        VecSet(child->_adnF, iGene + jGene,
          VecGet(parentB->_adnF, iGene + jGene));
        VecSet(child->_deltaAdnF, iGene + jGene,
          VecGet(parentB->_deltaAdnF, iGene + jGene));
      }
    }
  }
  // For each gene of the adn for floating point value of convolution
  // base functions
  for (long iGene = that->_NNdata._nbBaseConv * 3;
    iGene < GAGetLengthAdnFloat(that); iGene += 3) {
    // Get the gene from one parent or the other with equal probabililty
    if (rnd() < 0.5) {
      for (long jGene = 3; --jGene;) {
        VecSet(child->_adnF, iGene + jGene,
          VecGet(parentA->_adnF, iGene + jGene));
        VecSet(child->_deltaAdnF, iGene + jGene,
```

```
          VecGet(parentA->_deltaAdnF, iGene + jGene));
      }
    } else {
      for (long jGene = 3; --jGene;) {
        VecSet(child->_adnF, iGene + jGene,
          VecGet(parentB->_adnF, iGene + jGene));
        VecSet(child->_deltaAdnF, iGene + jGene,
          VecGet(parentB->_deltaAdnF, iGene + jGene));
      }
    }
  }
  // Reset the age of the child
  child->_age = 1;
  // Set the id of the child
  child->_id = (that->_nextId)++;
}

// Set the genes of the adn at rank 'iChild' as a 50/50 mix of the
// genes of adns at ranks 'parents[0]' and 'parents[1]'
void GAReproductionDefault(GenAlg* const that,
  const int* const parents, const int iChild) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (parents == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'parents' is null");
    PBErrCatch(GenAlgErr);
  }
  if (iChild < 0 || iChild >= GAGetNbAdns(that)) {
    GenAlgErr->_type = PBErrTypeInvalidArg;
    sprintf(GenAlgErr->_msg, "'child' is invalid (0<=%d<%d)",
      iChild, GAGetNbAdns(that));
    PBErrCatch(GenAlgErr);
  }
#endif
  // Get the parents and child
  GenAlgAdn* parentA = GAAdn(that, parents[0]);
  GenAlgAdn* parentB = GAAdn(that, parents[1]);
  GenAlgAdn* child = GAAdn(that, iChild);
  // For each gene of the adn for floating point value
  for (long iGene = GAGetLengthAdnFloat(that); iGene--;) {
    // Get the gene from one parent or the other with equal probabililty
    if (rnd() < 0.5) {
      VecSet(child->_adnF, iGene, VecGet(parentA->_adnF, iGene));
      VecSet(child->_deltaAdnF, iGene,
        VecGet(parentA->_deltaAdnF, iGene));
    } else {
      VecSet(child->_adnF, iGene, VecGet(parentB->_adnF, iGene));
      VecSet(child->_deltaAdnF, iGene,
        VecGet(parentB->_deltaAdnF, iGene));
    }
  }
  // For each gene of the adn for int value
  for (long iGene = GAGetLengthAdnInt(that); iGene--;) {
    // Get the gene from one parent or the other with equal probabililty
    if (rnd() < 0.5)
      VecSet(child->_adnI, iGene, VecGet(parentA->_adnI, iGene));
    else
```

```
      VecSet(child->_adnI, iGene, VecGet(parentB->_adnI, iGene));
  }
  // Reset the age of the child
  child->_age = 1;
  // Set the id of the child
  child->_id = (that->_nextId)++;
}

// Router toward the appropriate Mute function according to the type
// of GenAlg
void GAMute(GenAlg* const that, const int* const parents,
  const int iChild) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (parents == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'parents' is null");
    PBErrCatch(GenAlgErr);
  }
  if (iChild < 0 || iChild >= GAGetNbAdns(that)) {
    GenAlgErr->_type = PBErrTypeInvalidArg;
    sprintf(GenAlgErr->_msg, "'child' is invalid (0<=%d<%d)",
      iChild, GAGetNbAdns(that));
    PBErrCatch(GenAlgErr);
  }
#endif
  switch (GAGetType(that)) {
    case genAlgTypeNeuraNet:
      GAMuteNeuraNet(that, parents, iChild);
      break;
    case genAlgTypeNeuraNetConv:
      GAMuteNeuraNetConv(that, parents, iChild);
      break;
    case genAlgTypeDefault:
    default:
      GAMuteDefault(that, parents, iChild);
  }
}

// Mute the genes of the entity at rank 'iChild'
// This version is optimised to calculate the parameters of a NeuraNet
// by ensuring coherence in links: outputs have at least one link
// and there is no dead link
void GAMuteNeuraNet(GenAlg* const that, const int* const parents,
  const int iChild) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (parents == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'parents' is null");
    PBErrCatch(GenAlgErr);
  }
  if (iChild < 0 || iChild >= GAGetNbAdns(that)) {
    GenAlgErr->_type = PBErrTypeInvalidArg;
```

```
    sprintf(GenAlgErr->_msg, "'child' is invalid (0<=%d<%d)",
      iChild, GAGetNbAdns(that));
    PBErrCatch(GenAlgErr);
  }
#endif
  // Get the first parent and child
  GenAlgAdn* parentA = GAAdn(that, parents[0]);
  GenAlgAdn* child = GAAdn(that, iChild);
  // Get the proba and amplitude of mutation
  float probMute = sqrt(((float)iChild) / ((float)GAGetNbAdns(that)));
  float amp = 1.0 - sqrt(1.0 / (float)(parentA->_age + 1));
  probMute /= (float)(GAGetLengthAdnInt(that));
  probMute += (float)(parentA->_age) / 10000.0;
  // Ensure the proba is not null
  if (probMute < PBMATH_EPSILON)
    probMute = PBMATH_EPSILON;
  // Declare a variable to memorize if there has been mutation
  bool hasMuted = false;
  // Declare a variable to memorize the used values amongst input and
  // hidden
  long nbMaxUsedVal = that->_NNdata._nbIn + that->_NNdata._nbHid;
  char* isUsed = PBErrMalloc(GenAlgErr, sizeof(char) * nbMaxUsedVal);
  // Loop until there has been at least one mutation
  do {
    // Reset the used values
    memset(isUsed, 0, sizeof(char) * nbMaxUsedVal);
    memset(isUsed, 1, sizeof(char) * that->_NNdata._nbIn);
    // For each gene of the adn for int value (links definitions)
    for (long iGene = 0; iGene < GAGetLengthAdnInt(that); iGene += 3) {
      // If the link mutes
      if (rnd() < probMute) {
        hasMuted= true;
        // If this link is currently inactivated
        if (GAAdnGetGeneI(child, iGene) == -1) {
          // Base function
          long iBase = (int)round((float)iGene / 3.0);
          GAAdnSetGeneI(child, iGene, iBase);
          // Input
          long min =
            VecGet(GABoundsAdnInt(that, iGene + 1), 0);
          long max =
            VecGet(GABoundsAdnInt(that, iGene + 1), 1);
          long val = min;
          // Ensure the input is a used value
          do {
            val = (long)round((float)min +
              (float)(max - min) * rnd());
          } while (isUsed[val] == 0);
          GAAdnSetGeneI(child, iGene + 1, val);
          // Output
          min = MAX(val, VecGet(GABoundsAdnInt(that, iGene + 2), 0));
          max = VecGet(GABoundsAdnInt(that, iGene + 2), 1);
          val = (long)round((float)min + (float)(max - min) * rnd());
          GAAdnSetGeneI(child, iGene + 2, val);
          if (val < nbMaxUsedVal)
            isUsed[val] = 1;
        // Else, this link is currently activated
        } else {
          // Choose between inactivation or mutation
          if (rnd() < 0.5) {
            // Inactivate the link
            GAAdnSetGeneI(child, iGene, -1);
```

```
    } else {
      // Input
      long min =
        VecGet(GABoundsAdnInt(that, iGene + 1), 0);
      long max =
        VecGet(GABoundsAdnInt(that, iGene + 1), 1);
      long val = min;
      // Ensure the input is a used value
      do {
        val = (long)round((float)min +
          (float)(max - min) * rnd());
      } while (isUsed[val] == 0);
      GAAdnSetGeneI(child, iGene + 1, val);
      // Output
      min = MAX(val, VecGet(GABoundsAdnInt(that, iGene + 2), 0));
      max = VecGet(GABoundsAdnInt(that, iGene + 2), 1);
      val = (long)round((float)min + (float)(max - min) * rnd());
      GAAdnSetGeneI(child, iGene + 2, val);
      if (val < nbMaxUsedVal)
        isUsed[val] = 1;
    }
  }
}
// Get the index of the base function
long baseFun = GAAdnGetGeneI(child, iGene);
// If the link is active
if (baseFun != -1) {
  // If the associated base function mutes
  if (rnd() < probMute) {
    long baseFunGene = baseFun * 3;
    for (long jGene = 3; jGene--;) {
      // Get the bounds
      const VecFloat2D* const bounds =
        GABoundsAdnFloat(that, baseFunGene + jGene);
      // Declare a variable to memorize the previous value
      // of the gene
      float prevVal = GAAdnGetGeneF(child, baseFunGene + jGene);
      // Apply the mutation
      GAAdnSetGeneF(child, baseFunGene + jGene,
        GAAdnGetGeneF(child, baseFunGene + jGene) +
        (VecGet(bounds, 1) - VecGet(bounds, 0)) * amp *
        (rnd() - 0.5 +
        GAAdnGetDeltaGeneF(child, baseFunGene + jGene)));
      // Keep the gene value in bounds
      while (GAAdnGetGeneF(child, baseFunGene + jGene) <
        VecGet(bounds, 0) ||
        GAAdnGetGeneF(child, baseFunGene + jGene) >
        VecGet(bounds, 1)) {
        if (GAAdnGetGeneF(child, baseFunGene + jGene) >
          VecGet(bounds, 1))
          GAAdnSetGeneF(child, baseFunGene + jGene,
            2.0 * VecGet(bounds, 1) -
            GAAdnGetGeneF(child, baseFunGene + jGene));
        else if (GAAdnGetGeneF(child, baseFunGene + jGene) <
          VecGet(bounds, 0))
          GAAdnSetGeneF(child, baseFunGene + jGene,
            2.0 * VecGet(bounds, 0) -
            GAAdnGetGeneF(child, baseFunGene + jGene));
      }
      // Update the deltaAdn
      GAAdnSetDeltaGeneF(child, baseFunGene + jGene,
        GAAdnGetGeneF(child, baseFunGene + jGene) - prevVal);
```

```
          }
        }
      }
    }
  } while (hasMuted == false);
  free(isUsed);
}

// Mute the genes of the entity at rank 'iChild'
void GAMuteDefault(GenAlg* const that, const int* const parents,
  const int iChild) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (parents == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'parents' is null");
    PBErrCatch(GenAlgErr);
  }
  if (iChild < 0 || iChild >= GAGetNbAdns(that)) {
    GenAlgErr->_type = PBErrTypeInvalidArg;
    sprintf(GenAlgErr->_msg, "'child' is invalid (0<=%d<%d)",
      iChild, GAGetNbAdns(that));
    PBErrCatch(GenAlgErr);
  }
#endif
  // Get the first parent and child
  GenAlgAdn* parentA = GAAdn(that, parents[0]);
  GenAlgAdn* child = GAAdn(that, iChild);
  // Get the proba amplitude of mutation
  float probMute = sqrt(((float)iChild) / ((float)GAGetNbAdns(that)));
  float amp = 1.0 - sqrt(1.0 / (float)(parentA->_age));
  probMute /= (float)(GAGetLengthAdnInt(that));
  probMute += (float)(parentA->_age) / 10000.0;
  if (probMute < PBMATH_EPSILON)
    probMute = PBMATH_EPSILON;
  bool hasMuted = false;
  do {
    // For each gene of the adn for floating point value
    for (long iGene = GAGetLengthAdnFloat(that); iGene--;) {
      // If this gene mutes
      if (rnd() < probMute) {
        hasMuted = true;
        // Get the bounds
        const VecFloat2D* const bounds = GABoundsAdnFloat(that, iGene);
        // Declare a variable to memorize the previous value of the gene
        float prevVal = GAAdnGetGeneF(child, iGene);
        // Apply the mutation
        GAAdnSetGeneF(child, iGene, GAAdnGetGeneF(child, iGene) +
          (VecGet(bounds, 1) - VecGet(bounds, 0)) * amp *
          (rnd() - 0.5 + GAAdnGetDeltaGeneF(child, iGene)));
        // Keep the gene value in bounds
        while (GAAdnGetGeneF(child, iGene) < VecGet(bounds, 0) ||
          GAAdnGetGeneF(child, iGene) > VecGet(bounds, 1)) {
          if (GAAdnGetGeneF(child, iGene) > VecGet(bounds, 1))
            GAAdnSetGeneF(child, iGene,
              2.0 * VecGet(bounds, 1) - GAAdnGetGeneF(child, iGene));
          else if (GAAdnGetGeneF(child, iGene) < VecGet(bounds, 0))
            GAAdnSetGeneF(child, iGene,
```

```
            2.0 * VecGet(bounds, 0) - GAAdnGetGeneF(child, iGene));
        }
        // Update the deltaAdn
        GAAdnSetDeltaGeneF(child, iGene,
          GAAdnGetGeneF(child, iGene) - prevVal);
      }
    }
    // For each gene of the adn for int value
    for (long iGene = GAGetLengthAdnInt(that); iGene--;) {
      // If this gene mutes
      if (rnd() < probMute) {
        hasMuted = true;
        // Get the bounds
        const VecLong2D* const boundsI = GABoundsAdnInt(that, iGene);
        VecFloat2D bounds = VecLongToFloat2D(boundsI);
        // Apply the mutation (as it is int value, ensure the amplitude
        // is big enough to have an effect
        float ampI = MIN(2.0,
          (float)(VecGet(&bounds, 1) - VecGet(&bounds, 0)) * amp);
        GAAdnSetGeneI(child, iGene, GAAdnGetGeneI(child, iGene) +
          (long)round(ampI * (rnd() - 0.5)));
        // Keep the gene value in bounds
        while (GAAdnGetGeneI(child, iGene) < VecGet(&bounds, 0) ||
          GAAdnGetGeneI(child, iGene) > VecGet(&bounds, 1)) {
          if (GAAdnGetGeneI(child, iGene) > VecGet(&bounds, 1))
            GAAdnSetGeneI(child, iGene,
              2 * VecGet(&bounds, 1) - GAAdnGetGeneI(child, iGene));
          else if (GAAdnGetGeneI(child, iGene) < VecGet(&bounds, 0))
            GAAdnSetGeneI(child, iGene,
              2 * VecGet(&bounds, 0) - GAAdnGetGeneI(child, iGene));
        }
      }
    }
  } while (hasMuted == false);
}

// Mute the genes of the entity at rank 'iChild'
// This version is optimised to calculate the parameters of a NeuraNet
// with convolution by muting bases function per cell
void GAMuteNeuraNetConv(GenAlg* const that, const int* const parents,
  const int iChild) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (parents == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'parents' is null");
    PBErrCatch(GenAlgErr);
  }
  if (iChild < 0 || iChild >= GAGetNbAdns(that)) {
    GenAlgErr->_type = PBErrTypeInvalidArg;
    sprintf(GenAlgErr->_msg, "'child' is invalid (0<=%d<%d)",
      iChild, GAGetNbAdns(that));
    PBErrCatch(GenAlgErr);
  }
#endif
  // Get the first parent and child
  GenAlgAdn* parentA = GAAdn(that, parents[0]);
  GenAlgAdn* child = GAAdn(that, iChild);
```

```
  // Get the proba amplitude of mutation
  float probMute = sqrt(((float)iChild) / ((float)GAGetNbAdns(that)));
  float amp = 1.0 - sqrt(1.0 / (float)(parentA->_age));
  probMute /= (float)(GAGetLengthAdnInt(that));
  probMute += (float)(parentA->_age) / 10000.0;
  if (probMute < PBMATH_EPSILON)
    probMute = PBMATH_EPSILON;
  bool hasMuted = false;
  do {
    // For each gene of the adn for floating point value
    for (long iGene = GAGetLengthAdnFloat(that); iGene--;) {
      // If this gene mutes
      if (rnd() < probMute) {
        hasMuted = true;
        // Get the bounds
        const VecFloat2D* const bounds = GABoundsAdnFloat(that, iGene);
        // Declare a variable to memorize the previous value of the gene
        float prevVal = GAAdnGetGeneF(child, iGene);
        // Apply the mutation
        GAAdnSetGeneF(child, iGene, GAAdnGetGeneF(child, iGene) +
          (VecGet(bounds, 1) - VecGet(bounds, 0)) * amp *
          (rnd() - 0.5 + GAAdnGetDeltaGeneF(child, iGene)));
        // Keep the gene value in bounds
        while (GAAdnGetGeneF(child, iGene) < VecGet(bounds, 0) ||
          GAAdnGetGeneF(child, iGene) > VecGet(bounds, 1)) {
          if (GAAdnGetGeneF(child, iGene) > VecGet(bounds, 1))
            GAAdnSetGeneF(child, iGene,
              2.0 * VecGet(bounds, 1) - GAAdnGetGeneF(child, iGene));
          else if (GAAdnGetGeneF(child, iGene) < VecGet(bounds, 0))
            GAAdnSetGeneF(child, iGene,
              2.0 * VecGet(bounds, 0) - GAAdnGetGeneF(child, iGene));
        }
        // Update the deltaAdn
        GAAdnSetDeltaGeneF(child, iGene,
          GAAdnGetGeneF(child, iGene) - prevVal);
      }
    }
  } while (hasMuted == false);
}

// Print the information about the GenAlg 'that' on the stream 'stream'
void GAPrintln(const GenAlg* const that, FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (stream == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'stream' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  fprintf(stream, "epoch:%lu\n", GAGetCurEpoch(that));
  fprintf(stream, "%d entities, %d elites\n", GAGetNbAdns(that),
    GAGetNbElites(that));
  GSetIterBackward iter = GSetIterBackwardCreateStatic(GAAdns(that));
  int iEnt = 0;
  do {
    GenAlgAdn* ent = GSetIterGet(&iter);
    fprintf(stream, "#%d value:%f ", iEnt,
```

```
      GSetIterGetElem(&iter)->_sortVal);
    if (iEnt < GAGetNbElites(that))
      fprintf(stream, "elite ");
    GAAdnPrintln(ent, stream);
    ++iEnt;
  } while (GSetIterStep(&iter));
}

// Print a summary about the elite entities of the GenAlg 'that'
// on the stream 'stream'
void GAEliteSummaryPrintln(const GenAlg* const that,
  FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (stream == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'stream' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  GSetIterBackward iter = GSetIterBackwardCreateStatic(GAAdns(that));
  int iEnt = 0;
  GenAlgAdn* leader = GSetIterGet(&iter);
  fprintf(stream, "(age,val,div) ");
  do {
    GenAlgAdn* ent = GSetIterGet(&iter);
    fprintf(stream, "(%lu,%.3f,%.3f) ", GAAdnGetAge(ent),
      GSetIterGetElem(&iter)->_sortVal,
      GAAdnGetDiversity(ent, leader, that));
    ++iEnt;
  } while (GSetIterStep(&iter) && iEnt < GAGetNbElites(that));
  fprintf(stream, "\n");
}

// Update the norm of the range value for adans of the GenAlg 'that'
void GAUpdateNormRange(GenAlg* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  // If there are float adn
  if (GAGetLengthAdnFloat(that) > 0) {
    // Declare a vector to memorize the ranges in float gene values
    VecFloat* range = VecFloatCreate(GAGetLengthAdnFloat(that));
    // Calculate the ranges in gene values
    for (long iGene = GAGetLengthAdnFloat(that); iGene--;)
      VecSet(range, iGene,
        VecGet(GABoundsAdnFloat(that, iGene), 1) -
        VecGet(GABoundsAdnFloat(that, iGene), 0));
    // Calculate the norm of the range
    that->_normRangeFloat = VecNorm(range);
    // Free memory
    VecFree(&range);
  }
```

```
  // If there are int adn
  if (GAGetLengthAdnInt(that) > 0) {
    // Declare a vector to memorize the ranges in int gene values
    VecFloat* range = VecFloatCreate(GAGetLengthAdnInt(that));
    // Calculate the ranges in gene values
    for (long iGene = GAGetLengthAdnInt(that); iGene--;)
      VecSet(range, iGene,
        VecGet(GABoundsAdnInt(that, iGene), 1) -
        VecGet(GABoundsAdnInt(that, iGene), 0));
    // Calculate the norm of the range
    that->_normRangeInt = VecNorm(range);
    // Free memory
    VecFree(&range);
  }
}


// Get the diversity value of 'adnA' against 'adnB'
// The diversity is equal to
float GAAdnGetDiversity(const GenAlgAdn* const adnA,
  const GenAlgAdn* const adnB, const GenAlg* const ga) {
#if BUILDMODE == 0
  if (adnA == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'adnA' is null");
    PBErrCatch(GenAlgErr);
  }
  if (adnB == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'adnB' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  // Declare a variable to memorize the result
  float diversity = 0.0;
  // If there are adn for floating point values
  if (GAAdnAdnF(adnA) != NULL && GAAdnAdnF(adnB) != NULL) {
    // Get the difference in adn with the first entity
    VecFloat* diff =
      VecGetOp(GAAdnAdnF(adnA), 1.0, GAAdnAdnF(adnB), -1.0);
    // Calculate the diversity
    diversity += VecNorm(diff) / ga->_normRangeFloat;
    // Free memory
    VecFree(&diff);
  }
  // If there are adn for int values
  if (GAAdnAdnI(adnA) != NULL && GAAdnAdnI(adnB) != NULL) {
    // Get the difference in adn with the first entity
    VecLong* diffI =
      VecGetOp(GAAdnAdnI(adnA), 1, GAAdnAdnI(adnB), -1);
    VecFloat* diff = VecLongToFloat(diffI);
    // Calculate the diversity
    diversity += VecNorm(diff) / ga->_normRangeInt;
    // Free memory
    VecFree(&diffI);
    VecFree(&diff);
  }
  // Correct diversity if there was both float and int adns
  if (GAAdnAdnF(adnA) != NULL && GAAdnAdnF(adnB) != NULL &&
    GAAdnAdnI(adnA) != NULL && GAAdnAdnI(adnB) != NULL)
    diversity /= 2.0;
  // Return the result
```

```
  return diversity;
}

// Function which return the JSON encoding of 'that'
JSONNode* GAAdnEncodeAsJSON(const GenAlgAdn* const that,
  const float elo) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // Create the JSON structure
  JSONNode* json = JSONCreate();
  // Declare a buffer to convert value into string
  char val[100];
  // Encode the id
  sprintf(val, "%lu", that->_id);
  JSONAddProp(json, "_id", val);
  // Encode the age
  sprintf(val, "%lu", that->_age);
  JSONAddProp(json, "_age", val);
  // Encode the elo
  sprintf(val, "%f", elo);
  JSONAddProp(json, "_elo", val);
  // Encode the value
  sprintf(val, "%f", that->_val);
  JSONAddProp(json, "_val", val);
  // Encode the genes
  if (that->_adnF != NULL) {
    JSONAddProp(json, "_adnF", VecEncodeAsJSON(that->_adnF));
    JSONAddProp(json, "_deltaAdnF", VecEncodeAsJSON(that->_deltaAdnF));
  }
  if (that->_adnI != NULL)
    JSONAddProp(json, "_adnI", VecEncodeAsJSON(that->_adnI));
  // Return the created JSON
  return json;
}

// Function which return the JSON encoding of 'that'
JSONNode* GAEncodeAsJSON(const GenAlg* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // Create the JSON structure
  JSONNode* json = JSONCreate();
  // Declare a buffer to convert value into string
  char val[100];
  // Encode the type
  sprintf(val, "%d", GAGetType(that));
  JSONAddProp(json, "_type", val);
  switch (GAGetType(that)) {
    case genAlgTypeNeuraNet:
    case genAlgTypeNeuraNetConv:
      sprintf(val, "%d", that->_NNdata._nbIn);
      JSONAddProp(json, "NN_nbIn", val);
      sprintf(val, "%d", that->_NNdata._nbHid);
```

```
      JSONAddProp(json, "NN_nbHid", val);
      sprintf(val, "%d", that->_NNdata._nbOut);
      JSONAddProp(json, "NN_nbOut", val);
      sprintf(val, "%ld", that->_NNdata._nbBaseConv);
      JSONAddProp(json, "NN_nbBaseConv", val);
      sprintf(val, "%ld", that->_NNdata._nbBaseCellConv);
      JSONAddProp(json, "NN_nbBaseCellConv", val);
      break;
    default:
      break;
  }
  // Encode the nb adns
  sprintf(val, "%d", GAGetNbAdns(that));
  JSONAddProp(json, "_nbAdns", val);
  // Encode the nb elites
  sprintf(val, "%d", GAGetNbElites(that));
  JSONAddProp(json, "_nbElites", val);
  // Encode the length adn float
  sprintf(val, "%ld", GAGetLengthAdnFloat(that));
  JSONAddProp(json, "_lengthAdnF", val);
  // Encode the length adn int
  sprintf(val, "%ld", GAGetLengthAdnInt(that));
  JSONAddProp(json, "_lengthAdnI", val);
  // Encode the epoch
  sprintf(val, "%lu", GAGetCurEpoch(that));
  JSONAddProp(json, "_curEpoch", val);
  // Encode the next id
  sprintf(val, "%lu", that->_nextId);
  JSONAddProp(json, "_nextId", val);
  // Encode the bounds
  JSONArrayStruct setBoundFloat = JSONArrayStructCreateStatic();
  if (GAGetLengthAdnFloat(that) > 0) {
    for (long iBound = 0; iBound < GAGetLengthAdnFloat(that); ++iBound)
      JSONArrayStructAdd(&setBoundFloat,
        VecEncodeAsJSON((VecFloat*)GABoundsAdnFloat(that, iBound)));
    JSONAddProp(json, "_boundFloat", &setBoundFloat);
  }
  JSONArrayStruct setBoundInt = JSONArrayStructCreateStatic();
  if (GAGetLengthAdnInt(that) > 0) {
    for (long iBound = 0; iBound < GAGetLengthAdnInt(that); ++iBound)
      JSONArrayStructAdd(&setBoundInt,
        VecEncodeAsJSON((VecLong*)GABoundsAdnInt(that, iBound)));
    JSONAddProp(json, "_boundInt", &setBoundInt);
  }
  // Save the adns
  JSONArrayStruct setAdn = JSONArrayStructCreateStatic();
  for (int iEnt = 0; iEnt < GAGetNbAdns(that); ++iEnt) {
    GenAlgAdn* ent = GSetElemData(GSetElement(GAAdns(that), iEnt));
    float sortVal = GSetElemGetSortVal(GSetElement(GAAdns(that), iEnt));
    JSONArrayStructAdd(&setAdn, GAAdnEncodeAsJSON(ent, sortVal));
  }
  JSONAddProp(json, "_adns", &setAdn);
  // Save the best adn
  JSONAddProp(json, "_bestAdn",
    GAAdnEncodeAsJSON(GABestAdn(that), 0.0));
  // Free memory
  JSONArrayStructFlush(&setBoundFloat);
  JSONArrayStructFlush(&setBoundInt);
  JSONArrayStructFlush(&setAdn);
  // Return the created JSON
  return json;
}
```

```
// Function which decode from JSON encoding 'json' to 'that'
bool GAAdnDecodeAsJSON(GenAlgAdn** that, const JSONNode* const json) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (json == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'json' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // If 'that' is already allocated
  if (*that != NULL)
    // Free memory
    GenAlgAdnFree(that);
  // Get the id from the JSON
  JSONNode* prop = JSONProperty(json, "_id");
  if (prop == NULL) {
    return false;
  }
  unsigned long id = strtoul(JSONLabel(JSONValue(prop, 0)), NULL, 10);
  // Get the lengthAdnF from the JSON
  long lengthAdnF = 0;
  prop = JSONProperty(json, "_adnF");
  if (prop != NULL) {
    JSONNode* subprop = JSONProperty(prop, "_dim");
    lengthAdnF = atol(JSONLabel(JSONValue(subprop, 0)));
  }
  // Get the lengthAdnI from the JSON
  long lengthAdnI = 0;
  prop = JSONProperty(json, "_adnI");
  if (prop != NULL) {
    JSONNode* subprop = JSONProperty(prop, "_dim");
    lengthAdnI = atol(JSONLabel(JSONValue(subprop, 0)));
  }
  // Allocate memory
  *that = GenAlgAdnCreate(id, lengthAdnF, lengthAdnI);
  // Get the age from the JSON
  prop = JSONProperty(json, "_age");
  if (prop == NULL) {
    return false;
  }
  (*that)->_age = strtoul(JSONLabel(JSONValue(prop, 0)), NULL, 10);
  // Get the adnF from the JSON
  prop = JSONProperty(json, "_adnF");
  if (prop != NULL) {
    if (!VecDecodeAsJSON(&((*that)->_adnF), prop)) {
      return false;
    }
    prop = JSONProperty(json, "_deltaAdnF");
    if (prop == NULL) {
      return false;
    }
    if (!VecDecodeAsJSON(&((*that)->_deltaAdnF), prop)) {
      return false;
    }
  }
  // Get the adnI from the JSON
```

```
  prop = JSONProperty(json, "_adnI");
  if (prop != NULL)
    if (!VecDecodeAsJSON(&((*that)->_adnI), prop)) {
      return false;
    }
  // Get the value
  prop = JSONProperty(json, "_val");
  if (prop == NULL) {
    return false;
  }
  (*that)->_val = atof(JSONLabel(JSONValue(prop, 0)));
  // Return the success code
  return true;
}

// Function which decode from JSON encoding 'json' to 'that'
bool GADecodeAsJSON(GenAlg** that, const JSONNode* const json) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (json == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'json' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // If 'that' is already allocated
  if (*that != NULL)
    // Free memory
    GenAlgFree(that);
  // Decode the nb adns
  JSONNode* prop = JSONProperty(json, "_nbAdns");
  if (prop == NULL) {
    return false;
  }
  int nbAdns = atoi(JSONLabel(JSONValue(prop, 0)));
  // Decode the nb elites
  prop = JSONProperty(json, "_nbElites");
  if (prop == NULL) {
    return false;
  }
  int nbElites = atoi(JSONLabel(JSONValue(prop, 0)));
  // Decode the length adn float
  prop = JSONProperty(json, "_lengthAdnF");
  if (prop == NULL) {
    return false;
  }
  long lengthAdnF = atol(JSONLabel(JSONValue(prop, 0)));
  // Decode the length adn int
  prop = JSONProperty(json, "_lengthAdnI");
  if (prop == NULL) {
    return false;
  }
  long lengthAdnI = atol(JSONLabel(JSONValue(prop, 0)));
  // Allocate memory
  *that = GenAlgCreate(nbAdns, nbElites, lengthAdnF, lengthAdnI);
  // Decode the type
  prop = JSONProperty(json, "_type");
  if (prop == NULL) {
```

```
  return false;
}
int type = atoi(JSONLabel(JSONValue(prop, 0)));
switch (type) {
  case genAlgTypeNeuraNet:
  case genAlgTypeNeuraNetConv:
    prop = JSONProperty(json, "NN_nbIn");
    if (prop == NULL) {
      return false;
    }
    int nbIn = atoi(JSONLabel(JSONValue(prop, 0)));
    prop = JSONProperty(json, "NN_nbOut");
    if (prop == NULL) {
      return false;
    }
    int nbOut = atoi(JSONLabel(JSONValue(prop, 0)));
    prop = JSONProperty(json, "NN_nbHid");
    if (prop == NULL) {
      return false;
    }
    int nbHid = atoi(JSONLabel(JSONValue(prop, 0)));
    prop = JSONProperty(json, "NN_nbBaseConv");
    if (prop == NULL) {
      return false;
    }
    int nbBaseConv = atoi(JSONLabel(JSONValue(prop, 0)));
    prop = JSONProperty(json, "NN_nbBaseCellConv");
    if (prop == NULL) {
      return false;
    }
    int nbBaseCellConv = atoi(JSONLabel(JSONValue(prop, 0)));
    GASetTypeNeuraNetConv(*that, nbIn, nbHid, nbOut, nbBaseConv,
      nbBaseCellConv);
    break;
  default:
    break;
}
// Decode the epoch
prop = JSONProperty(json, "_curEpoch");
if (prop == NULL) {
  return false;
}
(*that)->_curEpoch =
  strtoul(JSONLabel(JSONValue(prop, 0)), NULL, 10);
// Decode the next id
prop = JSONProperty(json, "_nextId");
if (prop == NULL) {
  return false;
}
(*that)->_nextId = strtoul(JSONLabel(JSONValue(prop, 0)), NULL, 10);
// Decode the bounds
prop = JSONProperty(json, "_boundFloat");
if (prop != NULL) {
  if (JSONGetNbValue(prop) != GAGetLengthAdnFloat(*that))
    return false;
  for (long iBound = 0; iBound < GAGetLengthAdnFloat(*that); ++iBound) {
    JSONNode* val = JSONValue(prop, iBound);
    VecFloat2D* b = NULL;
    if (!VecDecodeAsJSON((VecFloat**)&b, val)) {
      return false;
    }
    GASetBoundsAdnFloat(*that, iBound, b);
```

```
      VecFree((VecFloat**)&b);
    }
  }
  prop = JSONProperty(json, "_boundInt");
  if (prop != NULL) {
    if (JSONGetNbValue(prop) != GAGetLengthAdnInt(*that))
      return false;
    for (long iBound = 0; iBound < GAGetLengthAdnInt(*that); ++iBound) {
      JSONNode* val = JSONValue(prop, iBound);
      VecLong2D* b = NULL;
      if (!VecDecodeAsJSON((VecLong**)&b, val)) {
        return false;
      }
      GASetBoundsAdnInt(*that, iBound, b);
      VecFree((VecLong**)&b);
    }
  }
  // Upadte the norm of the range values
  GAUpdateNormRange(*that);
  // Decode the adns
  prop = JSONProperty(json, "_adns");
  if (prop == NULL) {
    return false;
  }
  if (JSONGetNbValue(prop) != GAGetNbAdns(*that))
    return false;
  for (int iEnt = 0; iEnt < GAGetNbAdns(*that); ++iEnt) {
    JSONNode* val = JSONValue(prop, iEnt);
    if (!GAAdnDecodeAsJSON(
      (GenAlgAdn**)&(GSetElement(GAAdns(*that), iEnt)->_data), val)) {
      return false;
    }
  }
  // Decode the best adn
  prop = JSONProperty(json, "_bestAdn");
  if (prop == NULL) {
    return false;
  }
  if (!GAAdnDecodeAsJSON((GenAlgAdn**)&((*that)->_bestAdn), prop)) {
    return false;
  }

  // Return the success code
  return true;
}

// Load the GenAlg 'that' from the stream 'stream'
// If the GenAlg is already allocated, it is freed before loading
// Return true in case of success, else false
bool GALoad(GenAlg** that, FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (stream == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'stream' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
```

```
  // Declare a json to load the encoded data
  JSONNode* json = JSONCreate();
  // Load the whole encoded data
  if (!JSONLoad(json, stream)) {
    return false;
  }
  // Decode the data from the JSON
  if (!GADecodeAsJSON(that, json)) {
    return false;
  }
  // Free the memory used by the JSON
  JSONFree(&json);
  // Return the success code
  return true;
}

// Save the GenAlg 'that' to the stream 'stream'
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true in case of success, else false
bool GASave(const GenAlg* const that, FILE* const stream,
  const bool compact) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (stream == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'stream' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  // Get the JSON encoding
  JSONNode* json = GAEncodeAsJSON(that);
  // Save the JSON
  if (!JSONSave(json, stream, compact)) {
    return false;
  }
  // Free memory
  JSONFree(&json);
  // Return success code
  return true;
}
```

3.2 genalg-inline.c

```
// ============ GENALG-INLINE.C ================

// ------------- GenAlgAdn

// ================ Functions implementation ====================

// Return the adn for floating point values of the GenAlgAdn 'that'
#if BUILDMODE != 0
inline
#endif
const VecFloat* GAAdnAdnF(const GenAlgAdn* const that) {
```

```
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  return that->_adnF;
}

// Return the delta of adn for floating point values of the
// GenAlgAdn 'that'
#if BUILDMODE != 0
inline
#endif
const VecFloat* GAAdnDeltaAdnF(const GenAlgAdn* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  return that->_deltaAdnF;
}

// Return the adn for integer values of the GenAlgAdn 'that'
#if BUILDMODE != 0
inline
#endif
VecLong* GAAdnAdnI(const GenAlgAdn* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  return that->_adnI;
}

// Get the 'iGene'-th gene of the adn for floating point values of the
// GenAlgAdn 'that'
#if BUILDMODE != 0
inline
#endif
float GAAdnGetGeneF(const GenAlgAdn* const that, const long iGene) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  return VecGet(that->_adnF, iGene);
}

// Get the delta of the 'iGene'-th gene of the adn for floating point
// values of the GenAlgAdn 'that'
#if BUILDMODE != 0
inline
#endif
```

```
float GAAdnGetDeltaGeneF(const GenAlgAdn* const that, const long iGene) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  return VecGet(that->_deltaAdnF, iGene);
}

// Get the 'iGene'-th gene of the adn for int values of the
// GenAlgAdn 'that'
#if BUILDMODE != 0
inline
#endif
int GAAdnGetGeneI(const GenAlgAdn* const that, const long iGene) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  return VecGet(that->_adnI, iGene);
}

// Set the 'iGene'-th gene of the adn for floating point values of the
// GenAlgAdn 'that' to 'gene'
#if BUILDMODE != 0
inline
#endif
void GAAdnSetGeneF(GenAlgAdn* const that, const long iGene,
  const float gene) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  VecSet(that->_adnF, iGene, gene);
}

// Set the delta of the 'iGene'-th gene of the adn for floating point
// values of the GenAlgAdn 'that' to 'delta'
#if BUILDMODE != 0
inline
#endif
void GAAdnSetDeltaGeneF(GenAlgAdn* const that, const long iGene,
  const float delta) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  VecSet(that->_deltaAdnF, iGene, delta);
}

// Set the 'iGene'-th gene of the adn for int values of the
```

```
// GenAlgAdn 'that'to 'gene'
#if BUILDMODE != 0
inline
#endif
void GAAdnSetGeneI(GenAlgAdn* const that, const long iGene,
  const long gene) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  VecSet(that->_adnI, iGene, gene);
}

// Get the id of the GenAlgAdn 'that'
#if BUILDMODE != 0
inline
#endif
unsigned long int GAAdnGetId(const GenAlgAdn* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  return that->_id;
}

// Get the age of the GenAlgAdn 'that'
#if BUILDMODE != 0
inline
#endif
unsigned long int GAAdnGetAge(const GenAlgAdn* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  return that->_age;
}

// Get the value of the GenAlgAdn 'that'
#if BUILDMODE != 0
inline
#endif
float GAAdnGetVal(const GenAlgAdn* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  return that->_val;
}

// Return true if the GenAlgAdn 'that' is new, i.e. is age equals 1
```

```
// Return false
#if BUILDMODE != 0
inline
#endif
bool GAAdnIsNew(const GenAlgAdn* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  return (that->_age == 1);
}

// Copy the GenAlgAdn 'tho' into the GenAlgAdn 'that'
#if BUILDMODE != 0
inline
#endif
void GAAdnCopy(GenAlgAdn* const that, const GenAlgAdn* const tho) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (tho == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'tho' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  that->_id = tho->_id;
  that->_age = tho->_age;
  that->_val = tho->_val;
  if (tho->_adnF != NULL)
    VecCopy(that->_adnF, tho->_adnF);
  else
    VecFree(&(that->_adnF));
  if (tho->_deltaAdnF != NULL)
    VecCopy(that->_deltaAdnF, tho->_deltaAdnF);
  else
    VecFree(&(that->_deltaAdnF));
  if (tho->_adnI != NULL)
    VecCopy(that->_adnI, tho->_adnI);
  else
    VecFree(&(that->_adnI));
}

// ------------- GenAlg

// ================ Functions implementation ====================

// Get the type of the GenAlg 'that'
#if BUILDMODE != 0
inline
#endif
GenAlgType GAGetType(const GenAlg* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
```

```
    PBErrCatch(GenAlgErr);
  }
#endif
  return that->_type;
}

// Set the type of the GenAlg 'that' to genAlgTypeNeuraNet, the GenAlg
// will be used with a NeuraNet having 'nbIn' inputs, 'nbHid' hidden
// values and 'nbOut' outputs
#if BUILDMODE != 0
inline
#endif
void GASetTypeNeuraNet(GenAlg* const that, const int nbIn,
  const int nbHid, const int nbOut) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  if (GAGetLengthAdnFloat(that) != GAGetLengthAdnInt(that)) {
    GenAlgErr->_type = PBErrTypeInvalidArg;
    sprintf(GenAlgErr->_msg, "Must have the same nb of bases and links");
    PBErrCatch(GenAlgErr);
  }
  that->_type = genAlgTypeNeuraNet;
  that->_NNdata._nbIn = nbIn;
  that->_NNdata._nbHid = nbHid;
  that->_NNdata._nbOut = nbOut;
  that->_NNdata._nbBaseConv = 0;
}

// Set the type of the GenAlg 'that' to genAlgTypeNeuraNetConv,
// the GenAlg will be used with a NeuraNet having 'nbIn' inputs,
// 'nbHid' hidden values, 'nbOut' outputs, 'nbBaseConv' bases function
// dedicated to the convolution and 'nbBaseCellConv' bases function per cell of convolution
#if BUILDMODE != 0
inline
#endif
void GASetTypeNeuraNetConv(GenAlg* const that, const int nbIn,
  const int nbHid, const int nbOut, const long nbBaseConv,
  const long nbBaseCellConv) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  that->_type = genAlgTypeNeuraNetConv;
  that->_NNdata._nbIn = nbIn;
  that->_NNdata._nbHid = nbHid;
  that->_NNdata._nbOut = nbOut;
  that->_NNdata._nbBaseConv = nbBaseConv;
  that->_NNdata._nbBaseCellConv = nbBaseCellConv;
}

// Return the GSet of the GenAlg 'that'
#if BUILDMODE != 0
inline
#endif
```

```
GSet* GAAdns(const GenAlg* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  return that->_adns;
}

// Return the nb of entities of the GenAlg 'that'
#if BUILDMODE != 0
inline
#endif
int GAGetNbAdns(const GenAlg* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  return GSetNbElem(that->_adns);
}

// Return the nb of elites of the GenAlg 'that'
#if BUILDMODE != 0
inline
#endif
int GAGetNbElites(const GenAlg* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  return that->_nbElites;
}

// Return the current epoch of the GenAlg 'that'
#if BUILDMODE != 0
inline
#endif
unsigned long int GAGetCurEpoch(const GenAlg* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  return that->_curEpoch;
}

// Return the number of KTEvent of the GenAlg 'that'
#if BUILDMODE != 0
inline
#endif
unsigned long int GAGetNbKTEvent(const GenAlg* const that) {
#if BUILDMODE == 0
```

```
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  return that->_nbKTEvent;
}


// Get the length of adn for floating point value
#if BUILDMODE != 0
inline
#endif
long GAGetLengthAdnFloat(const GenAlg* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  return that->_lengthAdnF;
}

// Get the length of adn for integer value
#if BUILDMODE != 0
inline
#endif
long GAGetLengthAdnInt(const GenAlg* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  return that->_lengthAdnI;
}

// Set the bounds for the 'iGene'-th gene of adn for floating point
// values to a copy of 'bounds'
#if BUILDMODE != 0
inline
#endif
void GASetBoundsAdnFloat(GenAlg* const that, const long iGene,
  const VecFloat2D* const bounds) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (bounds == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'bounds' is null");
    PBErrCatch(GenAlgErr);
  }
  if (iGene < 0 || iGene >= that->_lengthAdnF) {
    GenAlgErr->_type = PBErrTypeInvalidArg;
    sprintf(GenAlgErr->_msg, "'iGene' is invalid (0<=%ld<%ld)",
      iGene, that->_lengthAdnF);
```

```
    PBErrCatch(GenAlgErr);
  }
#endif
  VecCopy(that->_boundsF + iGene, bounds);
  GAUpdateNormRange(that);
}

// Set the bounds for the 'iGene'-th gene of adn for integer values
// to a copy of 'bounds'
#if BUILDMODE != 0
inline
#endif
void GASetBoundsAdnInt(GenAlg* const that, const long iGene,
  const VecLong2D* const bounds) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (bounds == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'bounds' is null");
    PBErrCatch(GenAlgErr);
  }
  if (iGene < 0 || iGene >= that->_lengthAdnI) {
    GenAlgErr->_type = PBErrTypeInvalidArg;
    sprintf(GenAlgErr->_msg, "'iGene' is invalid (0<=%ld<%ld)",
      iGene, that->_lengthAdnI);
    PBErrCatch(GenAlgErr);
  }
#endif
  VecCopy(that->_boundsI + iGene, bounds);
  GAUpdateNormRange(that);
}

// Get the bounds for the 'iGene'-th gene of adn for floating point
// values
#if BUILDMODE != 0
inline
#endif
const VecFloat2D* GABoundsAdnFloat(const GenAlg* const that,
  const long iGene) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (iGene < 0 || iGene >= that->_lengthAdnF) {
    GenAlgErr->_type = PBErrTypeInvalidArg;
    sprintf(GenAlgErr->_msg, "'iGene' is invalid (0<=%ld<%ld)",
      iGene, that->_lengthAdnF);
    PBErrCatch(GenAlgErr);
  }
#endif
  return that->_boundsF + iGene;
}

// Get the bounds for the 'iGene'-th gene of adn for integer values
#if BUILDMODE != 0
inline
```

```
#endif
const VecLong2D* GABoundsAdnInt(const GenAlg* const that,
  const long iGene) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (iGene < 0 || iGene >= that->_lengthAdnI) {
    GenAlgErr->_type = PBErrTypeInvalidArg;
    sprintf(GenAlgErr->_msg, "'iGene' is invalid (0<=%ld<%ld)",
      iGene, that->_lengthAdnI);
    PBErrCatch(GenAlgErr);
  }
#endif
  return that->_boundsI + iGene;
}

// Get the GenAlgAdn of the GenAlg 'that' currently at rank 'iRank'
// (0 is the best adn)
#if BUILDMODE != 0
inline
#endif
GenAlgAdn* GAAdn(const GenAlg* const that, const int iRank) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (iRank < 0 || iRank >= GAGetNbAdns(that)) {
    GenAlgErr->_type = PBErrTypeInvalidArg;
    sprintf(GenAlgErr->_msg, "'iRank' is invalid (0<=%d<%d)",
      iRank, GAGetNbAdns(that));
    PBErrCatch(GenAlgErr);
  }
#endif
  return (GenAlgAdn*)GSetGet(that->_adns,
    GSetNbElem(that->_adns) - iRank - 1);
}

// Set the value of the GenAlgAdn 'adn' of the GenAlg 'that' to 'val'
#if BUILDMODE != 0
inline
#endif
void GASetAdnValue(GenAlg* const that, GenAlgAdn* const adn,
  const float val) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (adn == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'adn' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  // Set the value
  adn->_val = val;
```

```
  GSetElemSetSortVal((GSetElem*)GSetFirstElem(GAAdns(that), adn), val);
}

// Get the diversity of the GenAlg 'that'
#if BUILDMODE != 0
inline
#endif
float GAGetDiversity(const GenAlg* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  float diversity =
    GAAdn(that, 0)->_val - GAAdn(that, GAGetNbElites(that) - 1)->_val;
  return diversity;
}

// Return the best adn of the GenAlg 'that'
#if BUILDMODE != 0
inline
#endif
const GenAlgAdn* GABestAdn(const GenAlg* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  return that->_bestAdn;
}
```

4 Makefile

```
# Build mode
# 0: development (max safety, no optimisation)
# 1: release (min safety, optimisation)
# 2: fast and furious (no safety, optimisation)
BUILD_MODE?=0

all: main

# Makefile definitions
MAKEFILE_INC=../PBMake/Makefile.inc
include $(MAKEFILE_INC)

# Rules to make the executable
repo=genalg
$($(repo)_EXENAME): \
$($(repo)_EXENAME).o \
$($(repo)_EXE_DEP) \
$($(repo)_DEP)
$(COMPILER) `echo "$($(repo)_EXE_DEP) $($(repo)_EXENAME).o" | tr ' ' '\n' | sort -u` $(LINK_ARG) $($(repo)_LINK_ARG)

$($(repo)_EXENAME).o: \
```

```
$($(repo)_DIR)/$($(repo)_EXENAME).c \
$($(repo)_INC_H_EXE) \
$($(repo)_EXE_DEP)
$(COMPILER) $(BUILD_ARG) $($(repo)_BUILD_ARG) `echo "$($(repo)_INC_DIR)" | tr ' ' '\n' | sort -u` -c $($(repo)_DIR)/$
```

5 Unit tests

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include "genalg.h"

#define RANDOMSEED 2

void UnitTestGenAlgAdnCreateFree() {
  unsigned long int id = 1;
  int lengthAdnF = 2;
  int lengthAdnI = 3;
  GenAlgAdn* ent = GenAlgAdnCreate(id, lengthAdnF, lengthAdnI);
  if (ent->_age != 1 ||
    ent->_id != id ||
    VecGetDim(ent->_adnF) != lengthAdnF ||
    VecGetDim(ent->_deltaAdnF) != lengthAdnF ||
    VecGetDim(ent->_adnI) != lengthAdnI) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GenAlgAdnCreate failed");
    PBErrCatch(GenAlgErr);
  }
  GenAlgAdnFree(&ent);
  if (ent != NULL) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GenAlgAdnFree failed");
    PBErrCatch(GenAlgErr);
  }
  printf("UnitTestGenAlgAdnCreateFree OK\n");
}

void UnitTestGenAlgAdnGetSet() {
  unsigned long int id = 1;
  int lengthAdnF = 2;
  int lengthAdnI = 3;
  GenAlgAdn* ent = GenAlgAdnCreate(id, lengthAdnF, lengthAdnI);
  if (GAAdnAdnF(ent) != ent->_adnF) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GAAdnAdnF failed");
    PBErrCatch(GenAlgErr);
  }
  if (GAAdnDeltaAdnF(ent) != ent->_deltaAdnF) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GAAdnDeltaAdnF failed");
    PBErrCatch(GenAlgErr);
  }
  if (GAAdnAdnI(ent) != ent->_adnI) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
```

```
  sprintf(GenAlgErr->_msg, "GAAdnAdnI failed");
  PBErrCatch(GenAlgErr);
}
GAAdnSetGeneF(ent, 0, 1.0);
if (ISEQUALF(VecGet(ent->_adnF, 0), 1.0) == false) {
  GenAlgErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenAlgErr->_msg, "GAAdnSetGeneF failed");
  PBErrCatch(GenAlgErr);
}
if (ISEQUALF(GAAdnGetGeneF(ent, 0), 1.0) == false) {
  GenAlgErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenAlgErr->_msg, "GAAdnGetGeneF failed");
  PBErrCatch(GenAlgErr);
}
GAAdnSetDeltaGeneF(ent, 0, 2.0);
if (ISEQUALF(VecGet(ent->_deltaAdnF, 0), 2.0) == false) {
  GenAlgErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenAlgErr->_msg, "GAAdnSetDeltaGeneF failed");
  PBErrCatch(GenAlgErr);
}
if (ISEQUALF(GAAdnGetDeltaGeneF(ent, 0), 2.0) == false) {
  GenAlgErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenAlgErr->_msg, "GAAdnGetDeltaGeneF failed");
  PBErrCatch(GenAlgErr);
}
GAAdnSetGeneI(ent, 0, 3);
if (VecGet(ent->_adnI, 0) != 3) {
  GenAlgErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenAlgErr->_msg, "GAAdnSetGeneI failed");
  PBErrCatch(GenAlgErr);
}
if (GAAdnGetGeneI(ent, 0) != 3) {
  GenAlgErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenAlgErr->_msg, "GAAdnGetGeneI failed");
  PBErrCatch(GenAlgErr);
}
if (GAAdnGetAge(ent) != 1) {
  GenAlgErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenAlgErr->_msg, "GAAdnGetAge failed");
  PBErrCatch(GenAlgErr);
}
ent->_val = 2.0;
if (ISEQUALF(GAAdnGetVal(ent), 2.0) == false) {
  GenAlgErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenAlgErr->_msg, "GAAdnGetVal failed");
  PBErrCatch(GenAlgErr);
}
if (GAAdnGetId(ent) != id) {
  GenAlgErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenAlgErr->_msg, "GAAdnGetId failed");
  PBErrCatch(GenAlgErr);
}
if (GAAdnIsNew(ent) != true) {
  GenAlgErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenAlgErr->_msg, "GAAdnIsNew failed");
  PBErrCatch(GenAlgErr);
}
ent->_age = 2;
if (GAAdnIsNew(ent) != false) {
  GenAlgErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenAlgErr->_msg, "GAAdnIsNew failed");
  PBErrCatch(GenAlgErr);
```

```
  }
  GenAlgAdnFree(&ent);
  printf("UnitTestGenAlgAdnGetSet OK\n");
}

void UnitTestGenAlgAdnInit() {
  srandom(5);
  unsigned long int id = 1;
  int lengthAdnF = 2;
  int lengthAdnI = 2;
  GenAlgAdn* ent = GenAlgAdnCreate(id, lengthAdnF, lengthAdnI);
  GenAlg* ga = GenAlgCreate(GENALG_NBENTITIES, GENALG_NBELITES,
    lengthAdnF, lengthAdnI);
  VecFloat2D boundsF = VecFloatCreateStatic2D();
  VecLong2D boundsI = VecLongCreateStatic2D();
  VecSet(&boundsF, 0, -1.0); VecSet(&boundsF, 1, 1.0);
  VecSet(&boundsI, 0, 1); VecSet(&boundsI, 1, 10);
  GASetBoundsAdnFloat(ga, 0, &boundsF);
  GASetBoundsAdnFloat(ga, 1, &boundsF);
  GASetBoundsAdnInt(ga, 0, &boundsI);
  GASetBoundsAdnInt(ga, 1, &boundsI);
  GAAdnInit(ent, ga);
  if (ISEQUALF(VecGet(ent->_adnF, 0), -0.907064) == false ||
    ISEQUALF(VecGet(ent->_adnF, 1), -0.450509) == false ||
    VecGet(ent->_adnI, 0) != 2 ||
    VecGet(ent->_adnI, 1) != 10) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GAAdnInit failed");
    PBErrCatch(GenAlgErr);
  }
  GenAlgFree(&ga);
  GenAlgAdnFree(&ent);
  printf("UnitTestGenAlgAdnInit OK\n");
}

void UnitTestGenAlgAdn() {
  UnitTestGenAlgAdnCreateFree();
  UnitTestGenAlgAdnGetSet();
  UnitTestGenAlgAdnInit();
  printf("UnitTestGenAlgAdn OK\n");
}

void UnitTestGenAlgCreateFree() {
  int lengthAdnF = 2;
  int lengthAdnI = 3;
  GenAlg* ga = GenAlgCreate(GENALG_NBENTITIES, GENALG_NBELITES,
    lengthAdnF, lengthAdnI);
  if (ga->_type != genAlgTypeDefault ||
    ga->_curEpoch != 0 ||
    ga->_nbKTEvent != 0 ||
    ga->_nextId != GENALG_NBENTITIES ||
    ga->_nbElites != GENALG_NBELITES ||
    ga->_lengthAdnF != lengthAdnF ||
    ga->_lengthAdnI != lengthAdnI ||
    GSetNbElem(GAAdns(ga)) != GENALG_NBENTITIES) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GenAlgCreate failed");
    PBErrCatch(GenAlgErr);
  }
  GenAlgFree(&ga);
  if (ga != NULL) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
```

```
    sprintf(GenAlgErr->_msg, "GenAlgFree failed");
    PBErrCatch(GenAlgErr);
  }
  printf("UnitTestGenAlgCreateFree OK\n");
}

void UnitTestGenAlgGetSet() {
  int lengthAdnF = 2;
  int lengthAdnI = 3;
  GenAlg* ga = GenAlgCreate(GENALG_NBENTITIES, GENALG_NBELITES,
    lengthAdnF, lengthAdnI);
  if (GAGetType(ga) != ga->_type) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GAGetType failed");
    PBErrCatch(GenAlgErr);
  }
  if (GAAdns(ga) != ga->_adns) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GAEloRank failed");
    PBErrCatch(GenAlgErr);
  }
  if (GAGetNbAdns(ga) != GENALG_NBENTITIES) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GAGetNbAdns failed");
    PBErrCatch(GenAlgErr);
  }
  if (GAGetNbElites(ga) != GENALG_NBELITES) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GAGetNbElites failed");
    PBErrCatch(GenAlgErr);
  }
  if (GAGetCurEpoch(ga) != 0) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GAGetCurEpoch failed");
    PBErrCatch(GenAlgErr);
  }
  if (GAGetNbKTEvent(ga) != 0) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GAGetNbKTEvent failed");
    PBErrCatch(GenAlgErr);
  }
  GASetNbEntities(ga, 10);
  if (GAGetNbAdns(ga) != 10 ||
    GAGetNbElites(ga) != 9 ||
    GSetNbElem(GAAdns(ga)) != 10) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GASetNbEntities failed");
    PBErrCatch(GenAlgErr);
  }
  GASetNbElites(ga, 20);
  if (GAGetNbAdns(ga) != 21 ||
    GAGetNbElites(ga) != 20 ||
    GSetNbElem(GAAdns(ga)) != 21) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GASetNbElites failed");
    PBErrCatch(GenAlgErr);
  }
  if (GAGetLengthAdnFloat(ga) != lengthAdnF) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GAGetLengthAdnFloat failed");
    PBErrCatch(GenAlgErr);
  }
```

```
  if (GAGetLengthAdnInt(ga) != lengthAdnI) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GAGetLengthAdnInt failed");
    PBErrCatch(GenAlgErr);
  }
  if (GABoundsAdnFloat(ga, 1) != ga->_boundsF + 1) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GABoundsAdnFloat failed");
    PBErrCatch(GenAlgErr);
  }
  VecFloat2D boundsF = VecFloatCreateStatic2D();
  VecSet(&boundsF, 0, -1.0); VecSet(&boundsF, 1, 1.0);
  GASetBoundsAdnFloat(ga, 1, &boundsF);
  if (VecIsEqual(GABoundsAdnFloat(ga, 1), &boundsF) == false) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GASetBoundsAdnFloat failed");
    PBErrCatch(GenAlgErr);
  }
  VecLong2D boundsS = VecLongCreateStatic2D();
  VecSet(&boundsS, 0, -1); VecSet(&boundsS, 1, 1);
  GASetBoundsAdnInt(ga, 1, &boundsS);
  if (VecIsEqual(GABoundsAdnInt(ga, 1), &boundsS) == false) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GASetBoundsAdnInt failed");
    PBErrCatch(GenAlgErr);
  }
  if (GABoundsAdnInt(ga, 1) != ga->_boundsI + 1) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GABoundsAdnInt failed");
    PBErrCatch(GenAlgErr);
  }
  GASetAdnValue(ga, GAAdn(ga, 0), 1.0);
  if (ISEQUALF(GAAdn(ga, 0)->_val, 1.0) == false ||
    ISEQUALF(ga->_adns->_tail->_sortVal, 1.0) == false) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GASetAdnValue failed");
    PBErrCatch(GenAlgErr);
  }
  GenAlgFree(&ga);
  ga = GenAlgCreate(GENALG_NBENTITIES, GENALG_NBELITES, 3, 3);
  GASetTypeNeuraNet(ga, 1, 2, 3);
  if (GAGetType(ga) != genAlgTypeNeuraNet ||
    ga->_NNdata._nbIn != 1 ||
    ga->_NNdata._nbHid != 2 ||
    ga->_NNdata._nbOut != 3) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GASetTypeNeuraNet failed");
    PBErrCatch(GenAlgErr);
  }
  GenAlgFree(&ga);
  printf("UnitTestGenAlgGetSet OK\n");
}

void UnitTestGenAlgInit() {
  srandom(5);
  int lengthAdnF = 2;
  int lengthAdnI = 2;
  GenAlg* ga = GenAlgCreate(GENALG_NBENTITIES, GENALG_NBELITES,
    lengthAdnF, lengthAdnI);
  VecFloat2D boundsF = VecFloatCreateStatic2D();
  VecLong2D boundsI = VecLongCreateStatic2D();
  VecSet(&boundsF, 0, -1.0); VecSet(&boundsF, 1, 1.0);
```

```
  VecSet(&boundsI, 0, 1); VecSet(&boundsI, 1, 10);
  GASetBoundsAdnFloat(ga, 0, &boundsF);
  GASetBoundsAdnFloat(ga, 1, &boundsF);
  GASetBoundsAdnInt(ga, 0, &boundsI);
  GASetBoundsAdnInt(ga, 1, &boundsI);
  GAInit(ga);
  GenAlgAdn* ent = (GenAlgAdn*)(GAAdns(ga)->_head->_data);
  if (ISEQUALF(VecGet(ent->_adnF, 0), -0.907064) == false ||
    ISEQUALF(VecGet(ent->_adnF, 1), -0.450509) == false ||
    VecGet(ent->_adnI, 0) != 2 ||
    VecGet(ent->_adnI, 1) != 10) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GAInit failed");
    PBErrCatch(GenAlgErr);
  }
  GenAlgFree(&ga);
  printf("UnitTestGenAlgInit OK\n");
}

void UnitTestGenAlgPrint() {
  srandom(5);
  int lengthAdnF = 2;
  int lengthAdnI = 2;
  GenAlg* ga = GenAlgCreate(3, 2, lengthAdnF, lengthAdnI);
  VecFloat2D boundsF = VecFloatCreateStatic2D();
  VecLong2D boundsI = VecLongCreateStatic2D();
  VecSet(&boundsF, 0, -1.0); VecSet(&boundsF, 1, 1.0);
  VecSet(&boundsI, 0, 1); VecSet(&boundsI, 1, 10);
  GASetBoundsAdnFloat(ga, 0, &boundsF);
  GASetBoundsAdnFloat(ga, 1, &boundsF);
  GASetBoundsAdnInt(ga, 0, &boundsI);
  GASetBoundsAdnInt(ga, 1, &boundsI);
  GAInit(ga);
  GAPrintln(ga, stdout);
  GAEliteSummaryPrintln(ga, stdout);
  GenAlgFree(&ga);
  printf("UnitTestGenAlgInit OK\n");
}

void UnitTestGenAlgGetDiversity() {
  srandom(5);
  int lengthAdnF = 2;
  int lengthAdnI = 2;
  GenAlg* ga = GenAlgCreate(GENALG_NBENTITIES, GENALG_NBELITES,
    lengthAdnF, lengthAdnI);
  VecFloat2D boundsF = VecFloatCreateStatic2D();
  VecLong2D boundsI = VecLongCreateStatic2D();
  VecSet(&boundsF, 0, -1.0); VecSet(&boundsF, 1, 1.0);
  VecSet(&boundsI, 0, 1); VecSet(&boundsI, 1, 10);
  GASetBoundsAdnFloat(ga, 0, &boundsF);
  GASetBoundsAdnFloat(ga, 1, &boundsF);
  GASetBoundsAdnInt(ga, 0, &boundsI);
  GASetBoundsAdnInt(ga, 1, &boundsI);
  GASetNbElites(ga, 2);
  GASetNbEntities(ga, 3);
  GAInit(ga);
  if (ISEQUALF(GAGetDiversity(ga), 0.0) == false) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GAGetDiversity failed");
    PBErrCatch(GenAlgErr);
  }
  VecCopy(GAAdn(ga, 1)->_adnF, GAAdn(ga, 0)->_adnF);
```

```
  VecCopy(GAAdn(ga, 1)->_adnI, GAAdn(ga, 0)->_adnI);
  if (ISEQUALF(GAGetDiversity(ga), 0.0) == false) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GAGetDiversity failed");
    PBErrCatch(GenAlgErr);
  }
  GenAlgFree(&ga);
  printf("UnitTestGenAlgGetDiversity OK\n");
}

void UnitTestGenAlgStep() {
  srandom(2);
  int lengthAdnF = 2;
  int lengthAdnI = 2;
  GenAlg* ga = GenAlgCreate(3, 2, lengthAdnF, lengthAdnI);
  VecFloat2D boundsF = VecFloatCreateStatic2D();
  VecLong2D boundsI = VecLongCreateStatic2D();
  VecSet(&boundsF, 0, -1.0); VecSet(&boundsF, 1, 1.0);
  VecSet(&boundsI, 0, 1); VecSet(&boundsI, 1, 10);
  GASetBoundsAdnFloat(ga, 0, &boundsF);
  GASetBoundsAdnFloat(ga, 1, &boundsF);
  GASetBoundsAdnInt(ga, 0, &boundsI);
  GASetBoundsAdnInt(ga, 1, &boundsI);
  GAInit(ga);
  for (int i = 3; i--;)
    GASetAdnValue(ga, GAAdn(ga, i), 3.0 - (float)i);
  printf("Before Step:\n");
  GAPrintln(ga, stdout);
  GenAlgAdn* child = GAAdn(ga, 2);
  GAStep(ga);
  printf("After Step:\n");
  GAPrintln(ga, stdout);
  if (ga->_nextId != 4 || GAAdnGetId(child) != 3 ||
    GAAdnGetAge(child) != 1 ||
    ISEQUALF(GAAdnGetGeneF(child, 0), -0.156076) == false ||
    ISEQUALF(GAAdnGetGeneF(child, 1), 0.174965) == false ||
    ISEQUALF(GAAdnGetDeltaGeneF(child, 0), 0.0) == false ||
    ISEQUALF(GAAdnGetDeltaGeneF(child, 1), 0.0) == false ||
    GAAdnGetGeneI(child, 0) != 4 ||
    GAAdnGetGeneI(child, 1) != 7 ||
    GAAdn(ga, 2) != child ||
    GAAdnGetAge(GAAdn(ga, 0)) != 2 ||
    GAAdnGetAge(GAAdn(ga, 1)) != 2 ||
    GAAdnGetId(GAAdn(ga, 0)) != 0 ||
    GAAdnGetId(GAAdn(ga, 1)) != 1) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GAStep failed");
    PBErrCatch(GenAlgErr);
  }
  GenAlgFree(&ga);
  printf("UnitTestGenAlgStep OK\n");
}

void UnitTestGenAlgLoadSave() {
  srandom(5);
  int lengthAdnF = 2;
  int lengthAdnI = 2;
  GenAlg* ga = GenAlgCreate(3, 2, lengthAdnF, lengthAdnI);
  VecFloat2D boundsF = VecFloatCreateStatic2D();
  VecLong2D boundsI = VecLongCreateStatic2D();
  VecSet(&boundsF, 0, -1.0); VecSet(&boundsF, 1, 1.0);
  VecSet(&boundsI, 0, 1); VecSet(&boundsI, 1, 10);
```

```
GASetBoundsAdnFloat(ga, 0, &boundsF);
GASetBoundsAdnFloat(ga, 1, &boundsF);
GASetBoundsAdnInt(ga, 0, &boundsI);
GASetBoundsAdnInt(ga, 1, &boundsI);
GAInit(ga);
GAStep(ga);
GSet* rank = GSetCreate();
for (int i = 3; i--;)
  GSetAddSort(rank, GAAdn(ga, i), 3.0 - (float)i);
FILE* stream = fopen("./UnitTestGenAlgLoadSave.txt", "w");
if (GASave(ga, stream, false) == false) {
  GenAlgErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenAlgErr->_msg, "GASave failed");
  PBErrCatch(GenAlgErr);
}
fclose(stream);
stream = fopen("./UnitTestGenAlgLoadSave.txt", "r");
GenAlg* gaLoad = NULL;
if (GALoad(&gaLoad, stream) == false) {
  GenAlgErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenAlgErr->_msg, "GALoad failed");
  PBErrCatch(GenAlgErr);
}
fclose(stream);
if (ga->_nextId != gaLoad->_nextId ||
  ga->_curEpoch != gaLoad->_curEpoch ||
  ga->_nbElites != gaLoad->_nbElites ||
  ga->_type != genAlgTypeDefault ||
  ga->_lengthAdnF != gaLoad->_lengthAdnF ||
  ga->_lengthAdnI != gaLoad->_lengthAdnI ||
  VecIsEqual(ga->_boundsF, gaLoad->_boundsF) == false ||
  VecIsEqual(ga->_boundsF + 1, gaLoad->_boundsF + 1) == false ||
  VecIsEqual(ga->_boundsI, gaLoad->_boundsI) == false ||
  VecIsEqual(ga->_boundsI + 1, gaLoad->_boundsI + 1) == false ||
  GAAdnGetId(GAAdn(ga, 0)) != GAAdnGetId(GAAdn(gaLoad, 0)) ||
  GAAdnGetId(GAAdn(ga, 1)) != GAAdnGetId(GAAdn(gaLoad, 1)) ||
  GAAdnGetId(GAAdn(ga, 2)) != GAAdnGetId(GAAdn(gaLoad, 2)) ||
  GAAdnGetAge(GAAdn(ga, 0)) != GAAdnGetAge(GAAdn(gaLoad, 0)) ||
  GAAdnGetAge(GAAdn(ga, 1)) != GAAdnGetAge(GAAdn(gaLoad, 1)) ||
  GAAdnGetAge(GAAdn(ga, 2)) != GAAdnGetAge(GAAdn(gaLoad, 2)) ||
  VecIsEqual(GAAdn(ga, 0)->_adnF,
    GAAdn(gaLoad, 0)->_adnF) == false ||
  VecIsEqual(GAAdn(ga, 0)->_deltaAdnF,
    GAAdn(gaLoad, 0)->_deltaAdnF) == false ||
  VecIsEqual(GAAdn(ga, 0)->_adnI,
    GAAdn(gaLoad, 0)->_adnI) == false ||
  VecIsEqual(GAAdn(ga, 1)->_adnF,
    GAAdn(gaLoad, 1)->_adnF) == false ||
  VecIsEqual(GAAdn(ga, 1)->_deltaAdnF,
    GAAdn(gaLoad, 1)->_deltaAdnF) == false ||
  VecIsEqual(GAAdn(ga, 1)->_adnI,
    GAAdn(gaLoad, 1)->_adnI) == false ||
  VecIsEqual(GAAdn(ga, 2)->_adnF,
    GAAdn(gaLoad, 2)->_adnF) == false ||
  VecIsEqual(GAAdn(ga, 2)->_deltaAdnF,
    GAAdn(gaLoad, 2)->_deltaAdnF) == false ||
  VecIsEqual(GAAdn(ga, 2)->_adnI,
    GAAdn(gaLoad, 2)->_adnI) == false) {
  GenAlgErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenAlgErr->_msg, "UnitTestGenAlgLoadSave failed");
  PBErrCatch(GenAlgErr);
}
```

```
  GSetFree(&rank);
  GenAlgFree(&ga);
  GenAlgFree(&gaLoad);
  printf("UnitTestGenAlgLoadSave OK\n");
}

float ftarget(float x) {
  return -0.5 * fastpow(x, 3) + 0.314 * fastpow(x, 2) - 0.7777 * x + 0.1;
}

float evaluate(const VecFloat* adnF, const VecLong* adnI) {
  float delta = 0.02;
  int nb = (int)round(4.0 / delta);
  float res = 0.0;
  float x = -2.0;
  for (int i = 0; i < nb; ++i, x += delta) {
    float y = 0.0;
    for (int j = 4; j--;)
      y += VecGet(adnF, j) * fastpow(x, VecGet(adnI, j));
    res += fabs(ftarget(x) - y);
  }
  return res / (float)nb;
}

void UnitTestGenAlgTest() {
  srandom(5);
  int lengthAdnF = 4;
  int lengthAdnI = lengthAdnF;
  GenAlg* ga = GenAlgCreate(GENALG_NBENTITIES, GENALG_NBELITES,
    lengthAdnF, lengthAdnI);
  VecFloat2D boundsF = VecFloatCreateStatic2D();
  VecLong2D boundsI = VecLongCreateStatic2D();
  VecSet(&boundsF, 0, -1.0); VecSet(&boundsF, 1, 1.0);
  VecSet(&boundsI, 0, 0); VecSet(&boundsI, 1, 4);
  for (int i = lengthAdnF; i--;) {
    GASetBoundsAdnFloat(ga, i, &boundsF);
    GASetBoundsAdnInt(ga, i, &boundsI);
  }
  GAInit(ga);
  //GASetDiversityThreshold(ga, 0.0);
float best = 1.0;
//int step = 0;
  do {
//float ev = evaluate(GABestAdnF(ga), GABestAdnI(ga));
//printf("%lu %f %f\n",GAGetCurEpoch(ga), ev, GAGetDiversity(ga));
    for (int iEnt = GAGetNbAdns(ga); iEnt--;)
      if (GAAdnIsNew(GAAdn(ga, iEnt)))
        GASetAdnValue(ga, GAAdn(ga, iEnt),
          -1.0 * evaluate(GAAdnAdnF(GAAdn(ga, iEnt)),
          GAAdnAdnI(GAAdn(ga, iEnt))));
    GAStep(ga);
float ev = evaluate(GABestAdnF(ga), GABestAdnI(ga));
//if (step == 10){
//  printf("%d %f %f\n",GAGetCurEpoch(ga), ev, GAGetDiversity(ga));
//  step = 0;
//} else step++;
if (best - ev > PBMATH_EPSILON) {
  best = ev;
  printf("%lu %f ", GAGetCurEpoch(ga), best);
  VecFloatPrint(GABestAdnF(ga), stdout, 6);
  printf(" ");
  VecPrint(GABestAdnI(ga), stdout);
```

```
  printf("\n");
}
  } while (GAGetCurEpoch(ga) < 20000 ||
    evaluate(GABestAdnF(ga), GABestAdnI(ga)) < PBMATH_EPSILON);
  printf("target: -0.5*x^3 + 0.314*x^2 - 0.7777*x + 0.1\n");
  printf("approx: \n");
  GAAdnPrintln(GABestAdn(ga), stdout);
  printf("error: %f\n", evaluate(GABestAdnF(ga), GABestAdnI(ga)));
  GenAlgFree(&ga);
  printf("UnitTestGenAlgTest OK\n");
}

void UnitTestGenAlgPerf() {
  int nbRun = 500;
  unsigned long int nbMaxEpoch = 2000;
  float maxEv = 0.0;
  float bestEv = 0.0;
  float sumEv = 0.0;
  float avgEv = 0.0;
  for (int iRun = 0; iRun < nbRun; ++iRun) {
    srandom(time(NULL));
    int lengthAdnF = 4;
    int lengthAdnI = lengthAdnF;
    GenAlg* ga = GenAlgCreate(GENALG_NBENTITIES, GENALG_NBELITES,
      lengthAdnF, lengthAdnI);
    VecFloat2D boundsF = VecFloatCreateStatic2D();
    VecLong2D boundsI = VecLongCreateStatic2D();
    VecSet(&boundsF, 0, -1.0); VecSet(&boundsF, 1, 1.0);
    VecSet(&boundsI, 0, 0); VecSet(&boundsI, 1, 4);
    for (int i = lengthAdnF; i--;) {
      GASetBoundsAdnFloat(ga, i, &boundsF);
      GASetBoundsAdnInt(ga, i, &boundsI);
    }
    GAInit(ga);
    float ev = 0.0;
    do {
      for (int iEnt = GAGetNbAdns(ga); iEnt--;)
        if (GAAdnIsNew(GAAdn(ga, iEnt)))
          GASetAdnValue(ga, GAAdn(ga, iEnt),
            -1.0 * evaluate(GAAdnAdnF(GAAdn(ga, iEnt)),
            GAAdnAdnI(GAAdn(ga, iEnt))));
      GAStep(ga);
      ev = evaluate(GABestAdnF(ga), GABestAdnI(ga));
    } while (GAGetCurEpoch(ga) < nbMaxEpoch || ev < PBMATH_EPSILON);
    sumEv += ev;
    if (iRun == 0 || bestEv > ev)
      bestEv = ev;
    if (iRun == 0 || maxEv < ev)
      maxEv = ev;
//avgEv = sumEv / (float)iRun;
//printf("best: %f, worst: %f, avg: %f, %lu\n", bestEv, maxEv, avgEv, ga->_nbKTEvent);
    GenAlgFree(&ga);
  }
  avgEv = sumEv / (float)nbRun;
  printf("in %d runs, %lu epochs, best: %f, worst: %f, avg: %f\n",
    nbRun, nbMaxEpoch, bestEv, maxEv, avgEv);
  printf("UnitTestGenAlgPerf OK\n");
}

void UnitTestGenAlg() {
  UnitTestGenAlgCreateFree();
  UnitTestGenAlgGetSet();
```

```
  UnitTestGenAlgInit();
  UnitTestGenAlgPrint();
  UnitTestGenAlgGetDiversity();
  UnitTestGenAlgStep();
  UnitTestGenAlgLoadSave();
  UnitTestGenAlgTest();
  UnitTestGenAlgPerf();
  printf("UnitTestGenAlg OK\n");
}

void UnitTestAll() {
  UnitTestGenAlgAdn();
  UnitTestGenAlg();
  printf("UnitTestAll OK\n");
}

int main() {
  UnitTestAll();
  //UnitTestGenAlgPerf();
  // Return success code
  return 0;
}
```

6 Unit tests output

```
UnitTestGenAlgAdnCreateFree OK
UnitTestGenAlgAdnGetSet OK
UnitTestGenAlgAdnInit OK
UnitTestGenAlgAdn OK
UnitTestGenAlgCreateFree OK
UnitTestGenAlgGetSet OK
UnitTestGenAlgInit OK
epoch:0
3 entities, 2 elites
#0 value:0.000000 elite id:0 age:1
  adnF:<0.788004,-0.003504>
  deltaAdnF:<0.000000,0.000000>
  adnI:<3,1>
#1 value:0.000000 elite id:1 age:1
  adnF:<-0.840711,-0.704622>
  deltaAdnF:<0.000000,0.000000>
  adnI:<5,4>
#2 value:0.000000 id:2 age:1
  adnF:<-0.907064,-0.450509>
  deltaAdnF:<0.000000,0.000000>
  adnI:<2,10>
(age,val,div) (1,0.000,0.000) (1,0.000,0.455)
UnitTestGenAlgInit OK
UnitTestGenAlgGetDiversity OK
Before Step:
epoch:0
3 entities, 2 elites
#0 value:3.000000 elite id:0 age:1
  adnF:<0.285933,0.174965>
  deltaAdnF:<0.000000,0.000000>
  adnI:<4,10>
#1 value:2.000000 elite id:1 age:1
  adnF:<-0.156076,-0.303386>
```

```
  deltaAdnF:<0.000000,0.000000>
  adnI:<2,7>
#2 value:1.000000 id:2 age:1
  adnF:<0.619353,0.401953>
  deltaAdnF:<0.000000,0.000000>
  adnI:<2,2>
After Step:
epoch:1
3 entities, 2 elites
#0 value:3.000000 elite id:0 age:2
  adnF:<0.285933,0.174965>
  deltaAdnF:<0.000000,0.000000>
  adnI:<4,10>
#1 value:2.000000 elite id:1 age:2
  adnF:<-0.156076,-0.303386>
  deltaAdnF:<0.000000,0.000000>
  adnI:<2,7>
#2 value:1.000000 id:3 age:1
  adnF:<-0.156076,0.174965>
  deltaAdnF:<0.000000,0.000000>
  adnI:<4,7>
UnitTestGenAlgStep OK
UnitTestGenAlgLoadSave OK
2 0.256155 <0.568557,-0.743738,0.349810,-0.728776> <1,1,2,3>
5 0.164266 <-0.540080,-0.789917,0.214752,0.432405> <1,3,3,2>
10 0.128932 <0.410541,-0.899703,0.349810,-0.596805> <1,1,2,3>
12 0.111252 <0.315120,-0.899703,0.349810,-0.596805> <1,1,2,3>
13 0.110277 <0.315120,-0.881148,0.349810,-0.596805> <1,1,2,3>
17 0.108536 <0.348351,-0.896048,0.349810,-0.572533> <1,1,2,3>
20 0.074050 <0.195528,-0.875461,0.341126,-0.540044> <1,1,2,3>
24 0.067420 <0.195528,-0.863918,0.349810,-0.540044> <1,1,2,3>
33 0.067350 <0.195528,-0.863918,0.349810,-0.539760> <1,1,2,3>
34 0.066606 <-0.511536,-0.834290,0.349810,0.022997> <3,1,2,3>
39 0.061147 <-0.511536,-0.799226,0.349810,0.022997> <3,1,2,3>
46 0.059587 <-0.511536,-0.815583,0.349810,0.022997> <3,1,2,3>
49 0.059188 <-0.511536,-0.799226,0.358309,0.022997> <3,1,2,3>
69 0.059030 <-0.511536,-0.803219,0.358309,0.022997> <3,1,2,3>
112 0.058988 <-0.018471,-0.503888,-0.770280,0.370470> <2,3,1,2>
116 0.058977 <-0.018471,-0.511468,-0.746625,0.370470> <2,3,1,2>
120 0.058934 <-0.018471,-0.503888,-0.769498,0.370470> <2,3,1,2>
169 0.058876 <-0.018471,-0.503888,-0.767888,0.370470> <2,3,1,2>
221 0.058276 <-0.760091,-0.503888,0.027957,0.370470> <1,3,0,2>
379 0.043868 <-0.487453,0.329138,0.091891,-0.759397> <3,2,0,1>
381 0.033849 <-0.506735,0.329138,0.091891,-0.797105> <3,2,0,1>
383 0.017690 <-0.487453,0.329138,0.091891,-0.797105> <3,2,0,1>
412 0.015545 <-0.489918,0.329138,0.084616,-0.792611> <3,2,0,1>
419 0.013435 <-0.489918,0.325125,0.093707,-0.792611> <3,2,0,1>
432 0.013191 <-0.490301,0.325125,0.093707,-0.792611> <3,2,0,1>
436 0.012435 <-0.491399,0.325125,0.084616,-0.792611> <3,2,0,1>
441 0.009605 <-0.491399,0.316610,0.093230,-0.792611> <3,2,0,1>
512 0.008978 <-0.491399,0.316610,0.099342,-0.792519> <3,2,0,1>
685 0.008914 <-0.491399,0.316610,0.099342,-0.795866> <3,2,0,1>
698 0.008669 <-0.491399,0.315271,0.099342,-0.795866> <3,2,0,1>
831 0.007160 <-0.505292,0.315271,0.099342,-0.761168> <3,2,0,1>
852 0.006604 <-0.505292,0.310593,0.099342,-0.766974> <3,2,0,1>
860 0.005412 <-0.505292,0.315271,0.099342,-0.766974> <3,2,0,1>
912 0.004970 <-0.504548,0.315271,0.099342,-0.766468> <3,2,0,1>
914 0.004855 <-0.504548,0.315271,0.099342,-0.766974> <3,2,0,1>
961 0.004743 <-0.504548,0.313669,0.099342,-0.766974> <3,2,0,1>
1187 0.001497 <0.314424,-0.774460,0.099975,-0.501049> <2,1,0,3>
1188 0.001310 <0.314424,-0.774837,0.099975,-0.501049> <2,1,0,3>
1549 0.001174 <0.314424,-0.776076,0.099975,-0.501049> <2,1,0,3>
```

```
2531 0.001162 <0.314424,-0.776008,0.099975,-0.501049> <2,1,0,3>
3233 0.000898 <0.314424,-0.776008,0.099975,-0.500637> <2,1,0,3>
3559 0.000814 <0.314424,-0.776008,0.099241,-0.500637> <2,1,0,3>
4960 0.000794 <0.314238,-0.776008,0.099241,-0.500637> <2,1,0,3>
5821 0.000782 <0.314238,-0.776008,0.100058,-0.500637> <2,1,0,3>
6877 0.000745 <0.314238,-0.776125,0.100058,-0.500637> <2,1,0,3>
7095 0.000712 <0.314238,-0.776125,0.099856,-0.500637> <2,1,0,3>
7157 0.000677 <0.314054,-0.776125,0.099856,-0.500637> <2,1,0,3>
target: -0.5*x^3 + 0.314*x^2 - 0.7777*x + 0.1
approx:
id:1167158 age:1
  adnF:<0.314021,-0.776125,0.099987,-0.500637>
  deltaAdnF:<-0.000033,0.048377,-0.012747,0.000411>
  adnI:<2,1,0,3>
error: 0.000674
UnitTestGenAlgTest OK
in 500 runs, 2000 epochs, best: 0.000285, worst: 0.007826, avg: 0.002240
UnitTestGenAlgPerf OK
UnitTestGenAlg OK
UnitTestAll OK
```

UnitTestGenAlgLoadSave.txt:

```
{
  "_type":"0",
  "_nbAdns":"3",
  "_nbElites":"2",
  "_lengthAdnF":"2",
  "_lengthAdnI":"2",
  "_curEpoch":"1",
  "_nextId":"5",
  "_boundFloat":[
    {
      "_dim":"2",
      "_val":["-1.000000","1.000000"]
    },
    {
      "_dim":"2",
      "_val":["-1.000000","1.000000"]
    }
  ],
  "_boundInt":[
    {
      "_dim":"2",
      "_val":["1","10"]
    },
    {
      "_dim":"2",
      "_val":["1","10"]
    }
  ],
  "_adns":[
    {
      "_id":"4",
      "_age":"1",
      "_elo":"0.000000",
      "_val":"0.000000",
      "_adnF":{
        "_dim":"2",
        "_val":["0.541568","-0.654469"]
      },
```

```
    "_deltaAdnF":{
      "_dim":"2",
      "_val":["0.000000","0.000000"]
    },
    "_adnI":{
      "_dim":"2",
      "_val":["1","2"]
    }
  },
  {
    "_id":"3",
    "_age":"1",
    "_elo":"0.000000",
    "_val":"0.000000",
    "_adnF":{
      "_dim":"2",
      "_val":["-0.199914","-0.983663"]
    },
    "_deltaAdnF":{
      "_dim":"2",
      "_val":["0.000000","0.000000"]
    },
    "_adnI":{
      "_dim":"2",
      "_val":["3","5"]
    }
  },
  {
    "_id":"0",
    "_age":"1",
    "_elo":"0.000000",
    "_val":"0.000000",
    "_adnF":{
      "_dim":"2",
      "_val":["0.788004","-0.003504"]
    },
    "_deltaAdnF":{
      "_dim":"2",
      "_val":["0.000000","0.000000"]
    },
    "_adnI":{
      "_dim":"2",
      "_val":["3","1"]
    }
  }
],
"_bestAdn":{
  "_id":"0",
  "_age":"1",
  "_elo":"0.000000",
  "_val":"0.000000",
  "_adnF":{
    "_dim":"2",
    "_val":["0.788004","-0.003504"]
  },
  "_deltaAdnF":{
    "_dim":"2",
    "_val":["0.000000","0.000000"]
  },
  "_adnI":{
    "_dim":"2",
    "_val":["3","1"]
```

```
    }
  }
}
```

eval() of best genes over epoch: