GenAlg

P. Baillehache

May 10, 2020

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, repro-
duction and mutation are designed to efficiently explore all the possible gene
combination, and avoid local optimum. It is also possible to save and load

1

the GenAlg, and to record the history of birth of entities in a JSON file.

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

2

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"
#include "respublish.h"

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

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

#define GENALG_NBENTITIES 100
#define GENALG_NBELITES 20

#define GENALG_TXTOMETER_NBADNDISPLAYED 40
#define GENALG_TXTOMETER_LINE1 "Epoch #xxxxxx  KTEvent #xxxxxx  \n"
#define GENALG_TXTOMETER_FORMAT1 "Epoch #%06lu  KTEvent #%06lu\n"
#define GENALG_TXTOMETER_LINE2 "Id          Age       Val\n"
#define GENALG_TXTOMETER_LINE3 "xxxxxxxx  xxxxxx  +xxxxxx.xxxxxx\n"
#define GENALG_TXTOMETER_FORMAT3 "%08lu  %06lu  %+06.6f\n"
#define GENALG_TXTOMETER_LINE4 "-------------------------------\n"
#define GENALG_TXTOMETER_LINE5 "Diversity +xxx.xxxxx/+xxx.xxxxx \n"
#define GENALG_TXTOMETER_FORMAT5 "Diversity %+03.5f/%+03.5f \n"
#define GENALG_TXTOMETER_LINE6 "Size pool xxxxxx                \n"
#define GENALG_TXTOMETER_FORMAT6 "Size pool %06d  \n"

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

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

typedef struct GenAlg GenAlg;

typedef struct GenAlgAdn {
  // ID
  unsigned long _id;
  // ID parents
  unsigned long _idParents[2];
  // Age
  unsigned long _age;
  // Adn for floating point value
  VecFloat* _adnF;
  // Delta Adn during mutation for floating point value
  VecFloat* _deltaAdnF;
  // Adn for integer value
  VecLong* _adnI;
  // Value
  float _val;
  // Mutability of adn for floating point value
  VecFloat* _mutabilityF;
  // Mutability of adn for integer value
  VecFloat* _mutabilityI;
} 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);
```

4

```
// 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
static 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
static inline
#endif
const VecFloat* GAAdnDeltaAdnF(const GenAlgAdn* const that);

// Return the adn for integer values of the GenAlgAdn 'that'
#if BUILDMODE != 0
static 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(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
static 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
static 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
static 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
static 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
static 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
static inline
#endif
void GAAdnSetGeneI(GenAlgAdn* const that, const long iGene,
  const long gene);

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

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

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

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

// Set the mutability vectors for the GenAlgAdn 'that' to 'mutability'
#if BUILDMODE != 0
static inline
#endif
void GAAdnSetMutabilityInt(GenAlgAdn* const that,
  const VecFloat* const mutability);
#if BUILDMODE != 0
static inline
#endif
void GAAdnSetMutabilityFloat(GenAlgAdn* const that,
  const VecFloat* const mutability);

// ------------- GenAlg
```

```
// ================= 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;
  long _nbLink;
  // Flag to memorize if the links of the NeuraNet can be modified
  bool _flagMutableLink;
} GANeuraNet;

// Structures to save the history of the GenAlg
typedef struct GAHistoryBirth {
  // Epoch
  unsigned long _epoch;
  // First parent
  unsigned long _idParents[2];
  // Child
  unsigned long _idChild;
} GAHistoryBirth;
typedef struct GAHistory {
  // Set of GAHistoryBirth
  GSet _genealogy;
  // Path to the history file
  char* _path;
} GAHistory;

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;
  // Flag to memorize if there has been a KT event during last call to GAStep
  bool _flagKTEvent;
  // Flag to remember if we display info via a TextOMeter
  // about the population
  bool _flagTextOMeter;
  // DiversityThreshold
  float _diversityThreshold;
  // TextOMeter to display information about the population
  // If the TextOMEter is used, its ocntent is refreshed at each call
  // of the function GAStep();
  TextOMeter* _textOMeter;
  // Nb min of adns
  int _nbMinAdn;
  // Nb max of adns
  int _nbMaxAdn;
  // History of the GenAlg
  GAHistory _history;
  // Flag to remember if we are recording the history
  bool _flagHistory;
  // Maximum age for an entity
  unsigned long _maxAge;
} 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
static 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
static 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,
// 'nbLink' links dedicated to the convolution and 'nbBaseCellConv' bases function per cell of convolution
#if BUILDMODE != 0
```

```
static inline
#endif
void GASetTypeNeuraNetConv(GenAlg* const that, const int nbIn,
  const int nbHid, const int nbOut, const long nbBaseConv,
  const long nbBaseCellConv, const long nbLink);

// Return the GSet of the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
GSet* GAAdns(const GenAlg* const that);

// Return the max nb of adns of the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
int GAGetNbMaxAdn(const GenAlg* const that);

// Return the min nb of adns of the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
int GAGetNbMinAdn(const GenAlg* const that);

// Set the min nb of adns of the GenAlg 'that' to 'nb'
#if BUILDMODE != 0
static inline
#endif
void GASetNbMaxAdn(GenAlg* const that, const int nb);

// Set the min nb of adns of the GenAlg 'that' to 'nb'
#if BUILDMODE != 0
static inline
#endif
void GASetNbMinAdn(GenAlg* const that, const int nb);

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

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

// Set the diversity threshold of the GenAlg 'that' to 'threshold'
#if BUILDMODE != 0
static inline
#endif
void GASetDiversityThreshold(GenAlg* const that, const float threshold);

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

// Return the current epoch of the GenAlg 'that'
#if BUILDMODE != 0
```

```
static inline
#endif
unsigned long GAGetCurEpoch(const GenAlg* const that);

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

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

// Return the flag memorizing if the TextOMeter is displayed for
// the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
bool GAIsTextOMeterActive(const GenAlg* const that);

// Set the flag memorizing if the TextOMeter is displayed for
// the GenAlg 'that' to 'flag'
void GASetTextOMeterFlag(GenAlg* const that, bool flag);

// Set the link mutability flag for the NeuraNet data of
// the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
void GASetNeuraNetLinkMutability(GenAlg* const that, const bool flag);

// Get the link mutability flag for the NeuraNet data of
// the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
bool GAGetNeuraNetLinkMutability(GenAlg* const that);

// Get the flag about KTEvent at last call of GAStep for
// the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
bool GAGetFlagKTEvent(GenAlg* const that);

// Create a static GAHistory
GAHistory GAHistoryCreateStatic(void);

// Free the memory used by the GAHistory 'that'
void GAHistoryFree(GAHistory* that);

// Add a birth to the history of the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
void GAHistoryRecordBirth(GAHistory* const that, const GenAlgAdn* child,
  const unsigned int epoch);

// Set the history recording flag for the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
void GASetFlagHistory(GenAlg* const that, const bool flag);

// Set the path where the history is recorded for the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
```

```c
void GASetHistoryPath(GenAlg* const that, const char* const path);

// Get the path where the history is recorded for the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
const char* GAGetHistoryPath(GenAlg* const that);

// Get the history recording flag for the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
bool GAGetFlagHistory(const GenAlg* const that);

// Save the history of the GenAlg 'that'
// Return true if we could save the history, false else
bool GASaveHistory(const GenAlg* const that);

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

// Flush the content of the GAHistory 'that'
void GAHistoryFlush(GAHistory* that);

// Load the history into the GAHistory 'that' from the FILE 'stream'
// Return true if we could load the history, false else
bool GAHistoryLoad(GAHistory* const that, FILE* const stream);

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

// Set the maximum age for an entity of the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
void GASetMaxAge(GenAlg* const that, const unsigned long age);

// Get the maximum age for an entity of the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
unsigned long GAGetMaxAge(GenAlg* const that);

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

// =============== static 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);
      that->_mutabilityF = VecFloatCreate(lengthAdnF);
    } else {
      that->_adnF = NULL;
      that->_deltaAdnF = NULL;
      that->_mutabilityF = NULL;
    }
    if (lengthAdnI > 0) {
      that->_adnI = VecLongCreate(lengthAdnI);
      that->_mutabilityI = VecFloatCreate(lengthAdnI);
    } else {
      that->_adnI = NULL;
      that->_mutabilityI = 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));
    if ((*that)->_mutabilityF != NULL)
      VecFree(&((*that)->_mutabilityF));
    if ((*that)->_mutabilityI != NULL)
      VecFree(&((*that)->_mutabilityI));
    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(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);
    }
```

```c
  // Initialise the parent id, by default itself
  that->_idParents[0] = that->_id;
  that->_idParents[1] = that->_id;
}

// 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);
    VecSet(that->_mutabilityF, iGene, 1.0);
  }
  // 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);
    VecSet(that->_mutabilityI, iGene, 1.0);
  }
}

// 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);
    VecSet(that->_mutabilityF, iGene, 1.0);
  }
}

// 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);
    VecSet(that->_mutabilityF, iGene, 1.0);
  }
  // If the links are mutable
  if (ga->_NNdata._flagMutableLink == true) {
    // 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 = GAGetLengthAdnInt(ga); iGene--;) {
      VecSet(that->_adnI, iGene, -1);
      VecSet(that->_mutabilityI, iGene, 1.0);
    }
    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:");
  if (GAAdnAdnF(that) != NULL)
    VecFloatPrint(GAAdnAdnF(that), stream, 6);
  else
```

17

```
      fprintf(stream, "<null>");
    fprintf(stream, "\n");
    fprintf(stream, "  deltaAdnF:");
    if (GAAdnAdnF(that) != NULL)
      VecFloatPrint(GAAdnDeltaAdnF(that), stream, 6);
    else
      fprintf(stream, "<null>");
    fprintf(stream, "\n");
    fprintf(stream, "  adnI:");
    if (GAAdnAdnI(that) != NULL)
      VecPrint(GAAdnAdnI(that), stream);
    else
      fprintf(stream, "<null>");
    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);

// Refresh the content of the TextOMeter attached to the GenAlg 'that'
void GAUpdateTextOMeter(const GenAlg* const that);

// ================ 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->_flagTextOMeter = false;
  that->_diversityThreshold = PBMATH_EPSILON;
  that->_textOMeter = NULL;
  that->_nbMinAdn = nbEntities;
  that->_nbMaxAdn = nbEntities;
  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);
  that->_history = GAHistoryCreateStatic();
  that->_flagHistory = false;
  that->_maxAge = 100;
  // 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));
  if ((*that)->_textOMeter != NULL) {
    TextOMeterFree(&((*that)->_textOMeter));
  }
  GAHistoryFree(&((*that)->_history));
  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));
    that->_nextId++;
    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
  // Flush the history
  GAHistoryFlush(&(that->_history));
  // 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));
  that->_flagKTEvent = false;
  that->_curEpoch = 0;
  // If the user requested to save the history
  if (GAGetFlagHistory(that) == true) {
    // Update the history
    for (int iAdn = 0; iAdn < GAGetNbAdns(that); ++iAdn) {
      GAHistoryRecordBirth(
        &(that->_history), GAAdn(that, iAdn), GAGetCurEpoch(that));
    }
    // Save the history
    bool ret = GASaveHistory(that);
    if (ret == false) {
      GenAlgErr->_type = PBErrTypeNullPointer;
      sprintf(GenAlgErr->_msg, "Couldn't save the history");
      PBErrCatch(GenAlgErr);
    }
  }
}


// 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

  int iAdnMin = 1;
  while (iAdnMin < GAGetNbElites(that) - 1 &&
    fabs(GAAdn(that, iAdnMin)->_val - GAAdn(that, GAGetNbElites(that) - 1)->_val) > GAGetDiversityThreshold(that)) {
    ++iAdnMin;
  }

  for (int iAdn = GAGetNbAdns(that) - 1; iAdn >= iAdnMin ; --iAdn) {
    GenAlgAdn* adn = GAAdn(that, iAdn);
    GAAdnInit(adn, that);
    adn->_age = 0;
    adn->_id = (that->_nextId)++;
  }
  if (iAdnMin == 1)
    that->_nbKTEvent += 1;
//GSetSort(GAAdns(that));

  /*for (int iAdn = GAGetNbAdns(that) - 1; iAdn >= 1 ; --iAdn) {
    GenAlgAdn* adn = GAAdn(that, iAdn);
    GAAdnInit(adn, that);
    adn->_age = 0;
    adn->_id = (that->_nextId)++;
  }
  that->_nbKTEvent += 1;
  GSetSort(GAAdns(that));*/

  // Loop until the diversity of the elites is sufficient
  /*int nbKTEvent = 0;
  int nbMaxLoop =
    (int)round((float)GAGetNbAdns(that) / (float)GAGetNbElites(that));
  do {
    nbKTEvent = 0;
    // For each pair of adn

    //for (int iAdn = GAGetNbElites(that) - 1; iAdn >= 1 ; --iAdn) {

    for (int iAdn = MAX(GAGetNbElites(that) - 1, GAGetNbAdns(that) / 2); iAdn >= 1 ; --iAdn) {
      for (int jAdn = iAdn - 1; jAdn >= 0 ; --jAdn) {
        // Get the diversity of this pair
        float div = fabs(GAAdnGetVal(GAAdn(that, iAdn)) -
          GAAdnGetVal(GAAdn(that, jAdn)));
        // If it's below the diversity threshold
        if (div <= GAGetDiversityThreshold(that)) {
          GenAlgAdn* adn = GAAdn(that, iAdn);
          GAAdnInit(adn, that);
          adn->_age = 1;
          adn->_id = (that->_nextId)++;
          //GASetAdnValue(that, adn, worstValue);
          jAdn = 0;
          ++nbKTEvent;
        }
      }
    }
    // If their has been KTEvent
    if (nbKTEvent > 0) {
      // We need to sort the adns
      //GSetSort(GAAdns(that));
      // Update the flag about KTEvent
```

```
        that->_flagKTEvent = true;
      }
      --nbMaxLoop;
    } while (nbKTEvent > 0 && nbMaxLoop > 0);

    // Calculate a threshold for the age of the best elite
    //float thresholdAgeBest = GAAdnGetAge(GAAdn(that, 0)) / (float)GAGetNbElites(that);
    // Get the diversity relatively to the best of all
    float div = GAAdnGetVal(GABestAdn(that)) - GAAdnGetVal(GAAdn(that, 0));
    // If it's below the diversity threshold or it's older than
    // the threshold
    if ((div >= -PBMATH_EPSILON && div <= GAGetDiversityThreshold(that)) ||
      GAAdnGetAge(GAAdn(that, 0)) > (unsigned int)GAGetNbElites(that)) {
      GenAlgAdn* adn = GAAdn(that, 0);
      GAAdnInit(adn, that);
      adn->_age = 1;
      adn->_id = (that->_nextId)++;
      //GASetAdnValue(that, adn, worstValue);
      // We need to sort the adns
      GSetSort(GAAdns(that));
      // Memorize the total number of KTEvent
      that->_nbKTEvent += 1;
      // Update the flag about KTEvent
      //that->_flagKTEvent = true;
    }
    if (that->_flagKTEvent == true) {
      // We need to sort the adns
      GSetSort(GAAdns(that));
    }*/

}

// 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
  // Reset the flag about KTEvent
  that->_flagKTEvent = false;
  // Selection, Reproduction, Mutation
  // Ensure the set of adns is sorted
  GSetSort(GAAdns(that));
  // Variable to memorize if there has been improvement
  bool flagImprov = false;
  // Update the best adn if necessary
  if (that->_curEpoch == 1) {
    GAAdnCopy(that->_bestAdn, GAAdn(that, 0));
    that->_bestAdn->_age = that->_curEpoch + 1;
  } else {
    if (GAAdnGetVal(GAAdn(that, 0)) > GAAdnGetVal(GABestAdn(that))) {
      GAAdnCopy(that->_bestAdn, GAAdn(that, 0));
      that->_bestAdn->_age = that->_curEpoch + 1;
      flagImprov = true;
    } else {
      // If the boss is too old
      if (GAAdnGetAge(GAAdn(that, 0)) > GAGetMaxAge(that)) {
        // Kill the boss !
```

```
      GAAdn(that, 0)->_age = 0;
      GAAdnInit(GAAdn(that, 0), that);
      GAAdn(that, 0)->_age = 0;
      GAAdn(that, 0)->_id = (that->_nextId)++;
    }
    // Check for the diversity level
    float diversity = GAGetDiversity(that);
    if (diversity < GAGetDiversityThreshold(that)) {
      GAKTEvent(that);
    }
  }
}
// Refresh the TextOMeter if necessary
if (that->_flagTextOMeter) {
  GAUpdateTextOMeter(that);
}
// Resize the population according to the improvement
if (that->_curEpoch > 1) {
  if (flagImprov) {
    GASetNbEntities(that,
      MAX(GAGetNbMinAdn(that), GAGetNbAdns(that) / 2));
  } else {
    GASetNbEntities(that,
      MIN(GAGetNbMaxAdn(that), 2 * GAGetNbAdns(that)));
  }
}

// For each adn which is an elite
for (int iAdn = 0; iAdn < GAGetNbElites(that); ++iAdn) {
  // Increment age
  ++(GAAdn(that, iAdn)->_age);
  // Update the parents
  GAAdn(that, iAdn)->_idParents[0] = GAAdnGetId(GAAdn(that, iAdn));
  GAAdn(that, iAdn)->_idParents[1] = GAAdnGetId(GAAdn(that, iAdn));
}
// For each adn which is not an elite
for (int iAdn = GAGetNbElites(that); iAdn < GAGetNbAdns(that);
  ++iAdn) {
  // Declare a variable to memorize the parents
  int parents[2];
  // 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);
// If the user requested to save the history
if (GAGetFlagHistory(that) == true) {
  // Update the history
  for (int iAdn = 0; iAdn < GAGetNbAdns(that); ++iAdn) {
    GAHistoryRecordBirth(
      &(that->_history), GAAdn(that, iAdn), GAGetCurEpoch(that));
  }
}
}

// 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)) - 1;
  } 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];
  }

  // TEST
  parents[1] = parents[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);
  }
  // Update the parent id in the new child
  GenAlgAdn* child = GAAdn(that, iChild);
  child->_idParents[0] = GAAdnGetId(GAAdn(that, parents[0]));
  child->_idParents[1] = GAAdnGetId(GAAdn(that, parents[1]));
}

// 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 = sqrt(1.0 / (float)(parentA->_age + 1));
  probMute /= (float)(GAGetLengthAdnInt(that));
  probMute += (float)(parentA->_age) / 10000;
  // 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 (that->_NNdata._flagMutableLink == true && 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) {
      hasMuted = true;
      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 = sqrt(1.0 / (float)(parentA->_age));
  probMute /= (float)(MAX(GAGetLengthAdnInt(that),
    GAGetLengthAdnFloat(that)));
  probMute += (float)(parentA->_age) / 10000;
  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 = sqrt(1.0 / (float)(parentA->_age));
  probMute /= (float)(that->_NNdata._nbLink);
  probMute += (float)(parentA->_age) / 10000;
  if (probMute < PBMATH_EPSILON)
    probMute = PBMATH_EPSILON;
  bool hasMuted = false;
  int nbTry = 0;
  do {
    // For each gene of the adn for floating point value
    for (long iGene = GAGetLengthAdnFloat(that); iGene--;) {
      // If this gene mutes
      if (rnd() < probMute * VecGet(parentA->_mutabilityF, iGene)) {
        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);
      }
    }
    ++nbTry;
  } while (hasMuted == false && nbTry < 10);
}
```

```
// 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:
      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, "%d", that->_NNdata._flagMutableLink);
      JSONAddProp(json, "NN_flagMutablelink", val);
      break;
    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, "%d", that->_NNdata._flagMutableLink);
      JSONAddProp(json, "NN_flagMutablelink", val);
      sprintf(val, "%ld", that->_NNdata._nbBaseConv);
      JSONAddProp(json, "NN_nbBaseConv", val);
      sprintf(val, "%ld", that->_NNdata._nbBaseCellConv);
      JSONAddProp(json, "NN_nbBaseCellConv", val);
      sprintf(val, "%ld", that->_NNdata._nbLink);
      JSONAddProp(json, "NN_nbLink", 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(JSONLblVal(prop), NULL, 10);
  // Get the lengthAdnF from the JSON
  long lengthAdnF = 0;
```

39

```
    prop = JSONProperty(json, "_adnF");
    if (prop != NULL) {
      JSONNode* subprop = JSONProperty(prop, "_dim");
      lengthAdnF = atol(JSONLblVal(subprop));
    }
    // Get the lengthAdnI from the JSON
    long lengthAdnI = 0;
    prop = JSONProperty(json, "_adnI");
    if (prop != NULL) {
      JSONNode* subprop = JSONProperty(prop, "_dim");
      lengthAdnI = atol(JSONLblVal(subprop));
    }
    // 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(JSONLblVal(prop), 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(JSONLblVal(prop));
    // 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(JSONLblVal(prop));
  // Decode the nb elites
  prop = JSONProperty(json, "_nbElites");
  if (prop == NULL) {
    return false;
  }
  int nbElites = atoi(JSONLblVal(prop));
  // Decode the length adn float
  prop = JSONProperty(json, "_lengthAdnF");
  if (prop == NULL) {
    return false;
  }
  long lengthAdnF = atol(JSONLblVal(prop));
  // Decode the length adn int
  prop = JSONProperty(json, "_lengthAdnI");
  if (prop == NULL) {
    return false;
  }
  long lengthAdnI = atol(JSONLblVal(prop));
  // Allocate memory
  *that = GenAlgCreate(nbAdns, nbElites, lengthAdnF, lengthAdnI);
  // Decode the type
  prop = JSONProperty(json, "_type");
  if (prop == NULL) {
    return false;
  }
  int type = atoi(JSONLblVal(prop));
  int nbIn = 0;
  int nbOut = 0;
  int nbHid = 0;
  bool flagMutableLink = true;
  switch (type) {
    case genAlgTypeNeuraNet:
      prop = JSONProperty(json, "NN_nbIn");
      if (prop == NULL) {
        return false;
      }
      nbIn = atoi(JSONLblVal(prop));
      prop = JSONProperty(json, "NN_nbOut");
      if (prop == NULL) {
        return false;
      }
      nbOut = atoi(JSONLblVal(prop));
      prop = JSONProperty(json, "NN_nbHid");
      if (prop == NULL) {
        return false;
      }
      nbHid = atoi(JSONLblVal(prop));
      GASetTypeNeuraNet(*that, nbIn, nbHid, nbOut);
      prop = JSONProperty(json, "NN_flagMutableLink");
      if (prop == NULL) {
```

41

```
      return false;
    }
    flagMutableLink = atoi(JSONLblVal(prop));
    GASetNeuraNetLinkMutability(*that, flagMutableLink);
    break;
  case genAlgTypeNeuraNetConv:
    prop = JSONProperty(json, "NN_nbIn");
    if (prop == NULL) {
      return false;
    }
    nbIn = atoi(JSONLblVal(prop));
    prop = JSONProperty(json, "NN_nbOut");
    if (prop == NULL) {
      return false;
    }
    nbOut = atoi(JSONLblVal(prop));
    prop = JSONProperty(json, "NN_nbHid");
    if (prop == NULL) {
      return false;
    }
    nbHid = atoi(JSONLblVal(prop));
    prop = JSONProperty(json, "NN_nbBaseConv");
    if (prop == NULL) {
      return false;
    }
    long nbBaseConv = atol(JSONLblVal(prop));
    prop = JSONProperty(json, "NN_nbBaseCellConv");
    if (prop == NULL) {
      return false;
    }
    long nbBaseCellConv = atol(JSONLblVal(prop));
    prop = JSONProperty(json, "NN_nbLink");
    if (prop == NULL) {
      return false;
    }
    long nbLink = atol(JSONLblVal(prop));
    GASetTypeNeuraNetConv(*that, nbIn, nbHid, nbOut, nbBaseConv,
      nbBaseCellConv, nbLink);
    prop = JSONProperty(json, "NN_flagMutableLink");
    if (prop == NULL) {
      return false;
    }
    flagMutableLink = atoi(JSONLblVal(prop));
    GASetNeuraNetLinkMutability(*that, flagMutableLink);
    break;
  default:
    break;
}
// Decode the epoch
prop = JSONProperty(json, "_curEpoch");
if (prop == NULL) {
  return false;
}
(*that)->_curEpoch =
  strtoul(JSONLblVal(prop), NULL, 10);
// Decode the next id
prop = JSONProperty(json, "_nextId");
if (prop == NULL) {
  return false;
}
(*that)->_nextId = strtoul(JSONLblVal(prop), 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;
}


// Set the flag memorizing if the TextOMeter is displayed for
// the GenAlg 'that' to 'flag'
void GASetTextOMeterFlag(GenAlg* const that, bool flag) {
#if BUILDMODE == 0
  if (that == NULL) {
```

```
      GenAlgErr->_type = PBErrTypeNullPointer;
      sprintf(GenAlgErr->_msg, "'that' is null");
      PBErrCatch(GenAlgErr);
  }
#endif
  // If the requested flag is different from the current flag;
  if (that->_flagTextOMeter != flag) {
    if (flag && that->_textOMeter == NULL) {
      char title[] = "GenAlg";
      int width = strlen(GENALG_TXTOMETER_LINE1) + 1;
      int height = 10 +
        MIN(GENALG_TXTOMETER_NBADNDISPLAYED, GAGetNbMaxAdn(that));
      that->_textOMeter = TextOMeterCreate(title, width, height);
    }
    if (!flag && that->_textOMeter != NULL) {
      TextOMeterFree(&(that->_textOMeter));
    }
    that->_flagTextOMeter = flag;
  }
}


// Refresh the content of the TextOMeter attached to the GenAlg 'that'
void GAUpdateTextOMeter(const GenAlg* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (that->_textOMeter == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that->_textOMeter' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  // Clear the TextOMeter
  TextOMeterClear(that->_textOMeter);
  // Declare a variable to print the content of the TextOMeter
  char str[50];
  // Print the content of the TextOMeter
  // Epoch #xxxxxx  KTEvent #xxxxxx
  sprintf(str, GENALG_TXTOMETER_FORMAT1,
    GAGetCurEpoch(that), GAGetNbKTEvent(that));
  TextOMeterPrint(that->_textOMeter, str);
  // Diversity +xxxxxx.xxxxxx
  sprintf(str, GENALG_TXTOMETER_FORMAT5, GAGetDiversity(that),
    GAGetDiversityThreshold(that));
  TextOMeterPrint(that->_textOMeter, str);
  // Nb adns xxxxxx
  sprintf(str, GENALG_TXTOMETER_FORMAT6, GAGetNbAdns(that));
  TextOMeterPrint(that->_textOMeter, str);
  //
  sprintf(str, "\n");
  TextOMeterPrint(that->_textOMeter, str);
  // Id      Age     Val
  sprintf(str, GENALG_TXTOMETER_LINE2);
  TextOMeterPrint(that->_textOMeter, str);
  // xxxxxx  xxxxxx  +xxxxxx.xxxx
  sprintf(str, GENALG_TXTOMETER_FORMAT3,
    GAAdnGetId(GABestAdn(that)), GAAdnGetAge(GABestAdn(that)),
    GAAdnGetVal(GABestAdn(that)));
  TextOMeterPrint(that->_textOMeter, str);
```

```c
  // ......................
  sprintf(str, GENALG_TXTOMETER_LINE4);
  TextOMeterPrint(that->_textOMeter, str);
  // xxxxxx  xxxxxx  +xxxxxx.xxxx
  for (int iRank = 0; iRank < GAGetNbElites(that); ++iRank) {
    sprintf(str, GENALG_TXTOMETER_FORMAT3,
      GAAdnGetId(GAAdn(that, iRank)), GAAdnGetAge(GAAdn(that, iRank)),
      GAAdnGetVal(GAAdn(that, iRank)));
    TextOMeterPrint(that->_textOMeter, str);
  }
  // ......................
  sprintf(str, GENALG_TXTOMETER_LINE4);
  TextOMeterPrint(that->_textOMeter, str);
  // xxxxxx  xxxxxx  +xxxxxx.xxxx
  int maxRank = MIN(GENALG_TXTOMETER_NBADNDISPLAYED, GAGetNbAdns(that));
  for (int iRank = GAGetNbElites(that); iRank < maxRank; ++iRank) {
    sprintf(str, GENALG_TXTOMETER_FORMAT3,
      GAAdnGetId(GAAdn(that, iRank)), GAAdnGetAge(GAAdn(that, iRank)),
      GAAdnGetVal(GAAdn(that, iRank)));
    TextOMeterPrint(that->_textOMeter, str);
  }
  // Fill in with blank lines if necessary
  sprintf(str, "\n");
  for (int iBlank = GAGetNbAdns(that);
    iBlank < GENALG_TXTOMETER_NBADNDISPLAYED; ++iBlank) {
    TextOMeterPrint(that->_textOMeter, str);
  }
  // If there are more adns than availalbe space in the TextOMeter
  if (GAGetNbAdns(that)> GENALG_TXTOMETER_NBADNDISPLAYED) {
    sprintf(str, "...");
    TextOMeterPrint(that->_textOMeter, str);
  }
  // Flush the content of the TextOMeter
  TextOMeterFlush(that->_textOMeter);
}

// Create a static GAHistory
GAHistory GAHistoryCreateStatic(void) {
  // Declare the new GAHistory
  GAHistory that;
  // Init properties
  that._genealogy = GSetCreateStatic();
  that._path = strdup("./genAlgHistory.json");
  // Return the new GAHistory
  return that;
}

// Free the memory used by the GAHistory 'that'
void GAHistoryFree(GAHistory* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  // Flush the history
  GAHistoryFlush(that);
  // Free memory
  free(that->_path);
}
```

```c
// Flush the content of the GAHistory 'that'
void GAHistoryFlush(GAHistory* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  // Loop on the genealogy
  while (GSetNbElem(&(that->_genealogy)) > 0) {
    // Pop the birth
    GAHistoryBirth* birth = GSetPop(&(that->_genealogy));
    // Free memory
    free(birth);
  }
}


// Save the history of the GenAlg 'that'
bool GASaveHistory(const GenAlg* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  // Get the JSON encoding
  JSONNode* json = GAHistoryEncodeAsJSON(&(that->_history));
  // Open the stream
  FILE* stream = fopen(that->_history._path, "w");
  // Save the JSON
  bool compact = true;
  if (!JSONSave(json, stream, compact)) {
    return false;
  }
  // Close the stream
  fclose(stream);
  // Free memory
  JSONFree(&json);
  // Return the success code
  return true;
}


// Function which return the JSON encoding of the GAHistory 'that'
JSONNode* GAHistoryEncodeAsJSON(const GAHistory* 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];

  // Array of birth
  JSONArrayStruct genealogy = JSONArrayStructCreateStatic();
  // Loop on the births
```

47

```
    GSetIterForward iter =
      GSetIterForwardCreateStatic(
        &(that->_genealogy));
    do {
      // Get the birth
      GAHistoryBirth* birth = GSetIterGet(&iter);
      // Encode the birth
      JSONNode* birthJson = JSONCreate();
      sprintf(val, "%ld", birth->_epoch);
      JSONAddProp(birthJson, "_epoch", val);
      sprintf(val, "%ld", birth->_idParents[0]);
      JSONAddProp(birthJson, "_father", val);
      sprintf(val, "%ld", birth->_idParents[1]);
      JSONAddProp(birthJson, "_mother", val);
      sprintf(val, "%ld", birth->_idChild);
      JSONAddProp(birthJson, "_id", val);
      // Add the birth to the array
      JSONArrayStructAdd(&genealogy, birthJson);
    } while (GSetIterStep(&iter));
    // Add the genealogy
    JSONAddProp(json, "_genealogy", &genealogy);
    // Flush the temporary node for the genealogy
    JSONArrayStructFlush(&genealogy);
    // Return the created JSON
    return json;
}

// Load the history into the GAHistory 'that' from the FILE 'stream'
// Return true if we could load the history, false else
bool GAHistoryLoad(GAHistory* 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
  // 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 (!GAHistoryDecodeAsJSON(that, json)) {
    return false;
  }
  // Free the memory used by the JSON
  JSONFree(&json);
  // Return the success code
  return true;
}

// Function which decode from JSON encoding 'json' to GAHistory 'that'
bool GAHistoryDecodeAsJSON(GAHistory* const 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
  // Flush the history
  GAHistoryFlush(that);
  // Decode the genealogy
  JSONNode* propGen = JSONProperty(json, "_genealogy");
  if (propGen == NULL) {
    return false;
  }
  // Loop on the births
  for (int i = 0; i < JSONGetNbValue(propGen); ++i) {
    JSONNode* val = JSONValue(propGen, i);
    // Decode the epoch
    JSONNode* prop = JSONProperty(val, "_epoch");
    if (prop == NULL) {
      return false;
    }
    long epoch = atol(JSONLblVal(prop));
    // Decode the father
    prop = JSONProperty(val, "_father");
    if (prop == NULL) {
      return false;
    }
    long father = atol(JSONLblVal(prop));
    // Decode the epoch
    prop = JSONProperty(val, "_mother");
    if (prop == NULL) {
      return false;
    }
    long mother = atol(JSONLblVal(prop));
    // Decode the id
    prop = JSONProperty(val, "_id");
    if (prop == NULL) {
      return false;
    }
    GenAlgAdn adn;
    adn._id = atol(JSONLblVal(prop));
    adn._idParents[0] = father;
    adn._idParents[1] = mother;
    // Add the birth to history
    GAHistoryRecordBirth(that, &adn, epoch);
  }
  // Return the success code
  return true;
}
```

3.2   genalg-inline.c

```
// ============ GENALG-static inline.C ===============
```

```
// ------------- GenAlgAdn

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

// Return the adn for floating point values of the GenAlgAdn 'that'
#if BUILDMODE != 0
static 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
static 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
static 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
static 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
static 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
static 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
static 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
static 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
static 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
static 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
static 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
static 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
static 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
static 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));
}

// Set the mutability vectors for the GenAlgAdn 'that' to 'mutability'
#if BUILDMODE != 0
static inline
#endif
```

```
void GAAdnSetMutabilityInt(GenAlgAdn* const that,
  const VecFloat* const mutability) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (that->_mutabilityI == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that->_mutabilityI' is null");
    PBErrCatch(GenAlgErr);
  }
  if (mutability == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'mutability' is null");
    PBErrCatch(GenAlgErr);
  }
  if (VecGetDim(mutability) != VecGetDim(GAAdnAdnF(that))) {
    GenAlgErr->_type = PBErrTypeInvalidArg;
    sprintf(GenAlgErr->_msg, "'mutability''s dim is invalid (%ld==%ld)",
      VecGetDim(mutability), VecGetDim(GAAdnAdnI(that)));
    PBErrCatch(GenAlgErr);
  }
#endif
  VecCopy(that->_mutabilityI, mutability);
}

#if BUILDMODE != 0
static inline
#endif
void GAAdnSetMutabilityFloat(GenAlgAdn* const that,
  const VecFloat* const mutability) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (that->_mutabilityF == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that->_mutabilityF' is null");
    PBErrCatch(GenAlgErr);
  }
  if (mutability == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'mutability' is null");
    PBErrCatch(GenAlgErr);
  }
  if (VecGetDim(mutability) != VecGetDim(GAAdnAdnF(that))) {
    GenAlgErr->_type = PBErrTypeInvalidArg;
    sprintf(GenAlgErr->_msg, "'mutability''s dim is invalid (%ld==%ld)",
      VecGetDim(mutability), VecGetDim(GAAdnAdnF(that)));
    PBErrCatch(GenAlgErr);
  }
#endif
  VecCopy(that->_mutabilityF, mutability);
}

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

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

```c
// Get the type of the GenAlg 'that'
#if BUILDMODE != 0
static 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
static 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;
  that->_NNdata._flagMutableLink = true;
}

// 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,
// 'nbLink' links dedicated to the convolution and 'nbBaseCellConv' bases function per cell of convolution
#if BUILDMODE != 0
static inline
#endif
void GASetTypeNeuraNetConv(GenAlg* const that, const int nbIn,
  const int nbHid, const int nbOut, const long nbBaseConv,
  const long nbBaseCellConv, const long nbLink) {
#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;
    that->_NNdata._nbLink = nbLink;
    that->_NNdata._flagMutableLink = true;
}

// Return the GSet of the GenAlg 'that'
#if BUILDMODE != 0
static 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
static 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
static 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
static 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
static 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;
}

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

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

// Set the min nb of adns of the GenAlg 'that' to 'nb'
#if BUILDMODE != 0
static inline
#endif
void GASetNbMaxAdn(GenAlg* const that, const int nb) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
```

```
  }
#endif
  that->_nbMaxAdn = MAX(nb, GAGetNbElites(that) + 1);
  if (GAGetNbMinAdn(that) > that->_nbMaxAdn)
    GASetNbMinAdn(that, nb);
}

// Set the min nb of adns of the GenAlg 'that' to 'nb'
#if BUILDMODE != 0
static inline
#endif
void GASetNbMinAdn(GenAlg* const that, const int nb) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  that->_nbMinAdn = MAX(nb, GAGetNbElites(that) + 1);
  if (GAGetNbMaxAdn(that) < that->_nbMinAdn)
    GASetNbMaxAdn(that, nb);
}

// Get the length of adn for floating point value
#if BUILDMODE != 0
static 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
static 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
static inline
#endif
void GASetBoundsAdnFloat(GenAlg* const that, const long iGene,
  const VecFloat2D* const bounds) {
#if BUILDMODE == 0
```

```c
  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 (VecGet(bounds, 0) >= VecGet(bounds, 1)) {
    GenAlgErr->_type = PBErrTypeInvalidArg;
    sprintf(GenAlgErr->_msg, "'bounds' is invalid (%f<%f)",
      VecGet(bounds, 0), VecGet(bounds, 1));
    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
static 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 (VecGet(bounds, 0) >= VecGet(bounds, 1)) {
    GenAlgErr->_type = PBErrTypeInvalidArg;
    sprintf(GenAlgErr->_msg, "'bounds' is invalid (%ld<%ld)",
      VecGet(bounds, 0), VecGet(bounds, 1));
    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
static 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
static 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
static 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 < -1 || iRank >= GAGetNbAdns(that)) {
    GenAlgErr->_type = PBErrTypeInvalidArg;
    sprintf(GenAlgErr->_msg, "'iRank' is invalid (0<=%d<%d)",
      iRank, GAGetNbAdns(that));
    PBErrCatch(GenAlgErr);
```

```
  }
#endif
  if (iRank == -1)
    return (GenAlgAdn*)GABestAdn(that);
  else
    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
static 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
static 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 = fabs(
      GAAdn(that, GAGetNbElites(that) - 2)->_val - GAAdn(that, GAGetNbElites(that) - 1)->_val);
  return diversity;
}

// Get the diversity threshold of the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
float GAGetDiversityThreshold(const GenAlg* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
```

```
  return that->_diversityThreshold;
}

// Set the diversity threshold of the GenAlg 'that' to 'threshold'
#if BUILDMODE != 0
static inline
#endif
void GASetDiversityThreshold(GenAlg* const that, const float threshold) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  that->_diversityThreshold = threshold;
}

// Return the best adn of the GenAlg 'that'
#if BUILDMODE != 0
static 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;
}

// Return the flag memorizing if the TextOMeter is displayed for
// the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
bool GAIsTextOMeterActive(const GenAlg* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  return that->_flagTextOMeter;
}

// Set the link mutability flag for the NeuraNet data of
// the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
void GASetNeuraNetLinkMutability(GenAlg* const that, const bool flag) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
```

```c
  that->_NNdata._flagMutableLink = flag;
}

// Get the link mutability flag for the NeuraNet data of
// the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
bool GAGetNeuraNetLinkMutability(GenAlg* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  return that->_NNdata._flagMutableLink;
}

// Get the flag about KTEvent at last call of GAStep for
// the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
bool GAGetFlagKTEvent(GenAlg* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  return that->_flagKTEvent;
}

// Add a birth to the history of the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
void GAHistoryRecordBirth(GAHistory* const that, const GenAlgAdn* child,
  const unsigned int epoch) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (child == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'child' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  // Create the GAHistoryBirth
  GAHistoryBirth* birth = PBErrMalloc(GenAlgErr, sizeof(GAHistory));
  birth->_epoch = epoch;
  birth->_idParents[0] = child->_idParents[0];
  birth->_idParents[1] = child->_idParents[1];
  birth->_idChild = GAAdnGetId(child);
  // Add the birth to the genealogy of 'that'
  GSetAppend(&(that->_genealogy), birth);
}
```

```
// Set the history recording flag for the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
void GASetFlagHistory(GenAlg* const that, const bool flag) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  // Set the flag
  that->_flagHistory = flag;
}

// Get the history recording flag for the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
bool GAGetFlagHistory(const GenAlg* const that){
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  // Return the flag
  return that->_flagHistory;
}

// Set the path where the history is recorded for the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
void GASetHistoryPath(GenAlg* const that, const char* const path) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
  if (path == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'path' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  // Set the path
  free(that->_history._path);
  that->_history._path = strdup(path);
}

// Get the path where the history is recorded for the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
const char* GAGetHistoryPath(GenAlg* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
```

```
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  // Return the path
  return that->_history._path;
}


// Set the maximum age for an entity of the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
void GASetMaxAge(GenAlg* const that, const unsigned long age) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  // Set the maximum age
  that->_maxAge = age;
}


// Get the maximum age for an entity of the GenAlg 'that'
#if BUILDMODE != 0
static inline
#endif
unsigned long GAGetMaxAge(GenAlg* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "'that' is null");
    PBErrCatch(GenAlgErr);
  }
#endif
  // Return the maximumAge
  return that->_maxAge;
}
```

4 Makefile

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


all: pbmake_wget main


# Automatic installation of the repository PBMake in the parent folder
pbmake_wget:
if [ ! -d ../PBMake ]; then wget https://github.com/BayashiPascal/PBMake/archive/master.zip; unzip master.zip; rm -f


# 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

```c
#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 ||
      ga->_flagTextOMeter != false ||
      ga->_nbMinAdn != GENALG_NBENTITIES ||
      ga->_nbMaxAdn != GENALG_NBENTITIES ||
      ISEQUALF(ga->_diversityThreshold, PBMATH_EPSILON) != true ||
      ga->_textOMeter != NULL ||
      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);
  }
  if (ISEQUALF(GAGetDiversityThreshold(ga),
    ga->_diversityThreshold) != true) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GAGetDiversityThreshold failed");
```

```
    PBErrCatch(GenAlgErr);
}
GASetDiversityThreshold(ga, 2.0);
if (ISEQUALF(GAGetDiversityThreshold(ga), 2.0) != true) {
  GenAlgErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenAlgErr->_msg, "GASetDiversityThrehsold 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);
  }
  if (GAGetNbMaxAdn(ga) != ga->_nbMaxAdn) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GAGetNbMaxAdn failed");
    PBErrCatch(GenAlgErr);
  }
  if (GAGetNbMinAdn(ga) != ga->_nbMinAdn) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GAGetNbMinAdn failed");
    PBErrCatch(GenAlgErr);
  }
  GASetNbMaxAdn(ga, 100);
  if (GAGetNbMaxAdn(ga) != 100) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GASetNbMaxAdn failed");
    PBErrCatch(GenAlgErr);
  }
  GASetNbMinAdn(ga, 100);
  if (GAGetNbMinAdn(ga) != 100) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GASetNbMinAdn 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);
  }
  GASetNeuraNetLinkMutability(ga, true);
  if (ga->_NNdata._flagMutableLink != true) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GASetNeuraNetLinkMutability failed");
    PBErrCatch(GenAlgErr);
  }
  if (GAGetNeuraNetLinkMutability(ga) != true) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GAGetNeuraNetLinkMutability failed");
    PBErrCatch(GenAlgErr);
  }
  GASetNeuraNetLinkMutability(ga, false);
  if (ga->_NNdata._flagMutableLink != false) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GASetNeuraNetLinkMutability failed");
    PBErrCatch(GenAlgErr);
  }
  if (GAGetNeuraNetLinkMutability(ga) != false) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "GAGetNeuraNetLinkMutability failed");
    PBErrCatch(GenAlgErr);
  }
  GenAlgFree(&ga);
  printf("UnitTestGenAlgGetSet OK\n");
```

71

```
}

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), 0.285933) == 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) != 10 ||
      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(0);
  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);
  }
  GASetFlagHistory(ga, true);
  GASetHistoryPath(ga, "./history.json");
  GAInit(ga);
  GASetTextOMeterFlag(ga, true);
  GASetNbMinAdn(ga, GENALG_NBELITES * 2);
  GASetNbMaxAdn(ga, GENALG_NBENTITIES);
  GASetMaxAge(ga, 1000);
  float best = 1.0;
  unsigned long nbMaxEpoch = 2000;
  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);
    // Slow down the process to have time to read the TextOMeter
    unsigned int microseconds = 10000;
    usleep(microseconds);
    //sleep(1);
    // Display info if there is improvment
    float ev = evaluate(GABestAdnF(ga), GABestAdnI(ga));
    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) < nbMaxEpoch && best > PBMATH_EPSILON);
  // Save the history
  bool ret = GASaveHistory(ga);
  if (ret == false) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "Couldn't save the history");
    PBErrCatch(GenAlgErr);
  }
  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 = 10;
  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);
    GASetNbMinAdn(ga, GENALG_NBELITES * 2);
    GASetNbMaxAdn(ga, GENALG_NBENTITIES);
    GASetMaxAge(ga, 1000);
    float ev = 0.0;
    do {
      for (int iEnt = GAGetNbAdns(ga); iEnt--;)
```

76

```
        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 + 1);
    printf("best: %f, worst: %f, avg: %f, ktevent: %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 UnitTestGenAlgHistory() {
  srandom(0);
  int lengthAdnF = 4;
  int lengthAdnI = lengthAdnF;
  GenAlg* ga = GenAlgCreate(8, 4, 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);
  }
  GASetFlagHistory(ga, true);
  GASetHistoryPath(ga, "./UnitTestGenAlgHistory.json");
  GAInit(ga);
  GASetNbMinAdn(ga, 8);
  GASetNbMaxAdn(ga, 16);
  float best = 1.0;
  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);
    // Display info if there is improvment
    float ev = evaluate(GABestAdnF(ga), GABestAdnI(ga));
    if (best - ev > PBMATH_EPSILON) {
      best = ev;
    }
  } while (GAGetCurEpoch(ga) < 10 && best > PBMATH_EPSILON);
  // Save the history
  bool ret = GASaveHistory(ga);
  if (ret == false) {
    GenAlgErr->_type = PBErrTypeNullPointer;
    sprintf(GenAlgErr->_msg, "Couldn't save the history");
    PBErrCatch(GenAlgErr);
  }
```

77

```
        GAHistory history = GAHistoryCreateStatic();
        FILE* stream = fopen(GAGetHistoryPath(ga), "r");
        ret = GAHistoryLoad(&history, stream);
        if (ret == false) {
          GenAlgErr->_type = PBErrTypeUnitTestFailed;
          sprintf(GenAlgErr->_msg, "GAHistoryLoad failed");
          PBErrCatch(GenAlgErr);
        }
        fclose(stream);
        GSetIterForward iterA =
          GSetIterForwardCreateStatic(&(ga->_history._genealogy));
        GSetIterForward iterB =
          GSetIterForwardCreateStatic(&(history._genealogy));
        do {
          GAHistoryBirth* birthA = GSetIterGet(&iterA);
          GAHistoryBirth* birthB = GSetIterGet(&iterB);
          if (birthA->_epoch != birthB->_epoch &&
            birthA->_idParents[0] != birthB->_idParents[0] &&
            birthA->_idParents[1] != birthB->_idParents[1] &&
            birthA->_idChild != birthB->_idChild) {
            GenAlgErr->_type = PBErrTypeUnitTestFailed;
            sprintf(GenAlgErr->_msg, "GAHistoryLoad/Save failed");
            PBErrCatch(GenAlgErr);
          }

        } while (GSetIterStep(&iterA) && GSetIterStep(&iterB));
        GAHistoryFree(&history);
        GenAlgFree(&ga);
        printf("UnitTestGenAlgHistory OK\n");
}

void UnitTestGenAlg() {
        UnitTestGenAlgCreateFree();
        UnitTestGenAlgGetSet();
        UnitTestGenAlgInit();
        UnitTestGenAlgPrint();
        UnitTestGenAlgGetDiversity();
        UnitTestGenAlgStep();
        UnitTestGenAlgLoadSave();
        UnitTestGenAlgTest();
        UnitTestGenAlgPerf();
        UnitTestGenAlgHistory();
        printf("UnitTestGenAlg OK\n");
}

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

int main() {
        UnitTestAll();
        // 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,-0.303387>
  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,-0.303387>
  deltaAdnF:<0.000000,0.000000>
  adnI:<2,7>
#2 value:1.000000 id:3 age:1
  adnF:<0.285933,0.174965>
  deltaAdnF:<0.000000,0.000000>
  adnI:<4,10>
UnitTestGenAlgStep OK
UnitTestGenAlgLoadSave OK
2 0.203473 <-0.535477,-0.411679,-0.262673,0.499542> <3,1,1,2>
3 0.197736 <-0.535477,-0.411679,-0.327801,0.499542> <3,1,1,2>
4 0.197539 <-0.535477,-0.411679,-0.327801,0.499378> <3,1,1,2>
```

```
6 0.124057 <-0.535477,-0.448725,-0.183439,0.411522> <3,1,1,2>
8 0.108416 <-0.535477,-0.448725,-0.183439,0.390133> <3,1,1,2>
9 0.090007 <-0.535477,-0.448725,-0.224898,0.390133> <3,1,1,2>
11 0.079211 <-0.535477,-0.448725,-0.272853,0.390133> <3,1,1,2>
13 0.069481 <-0.535477,-0.526717,-0.183439,0.376734> <3,1,1,2>
22 0.066352 <-0.535477,-0.526717,-0.183439,0.365770> <3,1,1,2>
26 0.064598 <-0.535477,-0.514556,-0.183439,0.365770> <3,1,1,2>
39 0.063751 <-0.535477,-0.514556,-0.183439,0.360534> <3,1,1,2>
45 0.063558 <-0.535477,-0.510800,-0.183439,0.360534> <3,1,1,2>
71 0.059508 <-0.481343,-0.610945,-0.204030,0.360534> <3,1,1,2>
88 0.059488 <-0.475158,-0.610945,-0.223058,0.360534> <3,1,1,2>
110 0.059426 <-0.475158,-0.609679,-0.223058,0.360534> <3,1,1,2>
129 0.059311 <-0.475158,-0.609679,-0.223058,0.359554> <3,1,1,2>
143 0.059266 <-0.475158,-0.610226,-0.223058,0.359554> <3,1,1,2>
162 0.059216 <-0.475381,-0.610226,-0.223058,0.359554> <3,1,1,2>
196 0.059174 <-0.475381,-0.610226,-0.223058,0.358559> <3,1,1,2>
200 0.059102 <-0.476718,-0.610226,-0.223058,0.358559> <3,1,1,2>
211 0.059010 <-0.476718,-0.606485,-0.223058,0.358559> <3,1,1,2>
220 0.058928 <-0.476718,-0.525324,-0.305455,0.358559> <3,1,1,2>
294 0.058909 <-0.476718,-0.525324,-0.305455,0.357701> <3,1,1,2>
298 0.058837 <-0.477703,-0.525324,-0.305455,0.357701> <3,1,1,2>
335 0.058753 <-0.477703,-0.525324,-0.305455,0.357355> <3,1,1,2>
373 0.058729 <-0.477563,-0.525324,-0.305455,0.357355> <3,1,1,2>
391 0.058663 <-0.477563,-0.525324,-0.305455,0.356900> <3,1,1,2>
406 0.058650 <-0.477563,-0.525324,-0.305455,0.356558> <3,1,1,2>
427 0.058636 <-0.477563,-0.525324,-0.305727,0.356558> <3,1,1,2>
559 0.058618 <-0.477742,-0.525324,-0.305727,0.356558> <3,1,1,2>
583 0.058606 <-0.477742,-0.525324,-0.305727,0.356412> <3,1,1,2>
1682 0.058583 <-0.478980,-0.525324,-0.305727,0.355061> <3,1,1,2>
1695 0.058543 <-0.478980,-0.525324,-0.305727,0.353494> <3,1,1,2>
1698 0.058484 <-0.478980,-0.525324,-0.305727,0.353918> <3,1,1,2>
1768 0.058461 <-0.480723,-0.525324,-0.300932,0.353918> <3,1,1,2>
target: -0.5*x^3 + 0.314*x^2 - 0.7777*x + 0.1
approx:
id:209232 age:1768
  adnF:<-0.480723,-0.525324,-0.300932,0.353918>
  deltaAdnF:<-0.001743,0.081161,0.004794,0.000424>
  adnI:<3,1,1,2>
error: 0.058461
UnitTestGenAlgTest OK
best: 0.000144, worst: 0.000144, avg: 0.000144, ktevent: 550
best: 0.000144, worst: 0.058473, avg: 0.029308, ktevent: 582
best: 0.000144, worst: 0.058473, avg: 0.019630, ktevent: 553
best: 0.000144, worst: 0.058473, avg: 0.014791, ktevent: 553
best: 0.000144, worst: 0.058503, avg: 0.023533, ktevent: 605
best: 0.000144, worst: 0.058503, avg: 0.029362, ktevent: 605
best: 0.000144, worst: 0.058503, avg: 0.033517, ktevent: 586
best: 0.000144, worst: 0.058503, avg: 0.036633, ktevent: 586
best: 0.000144, worst: 0.058503, avg: 0.032589, ktevent: 548
best: 0.000144, worst: 0.058503, avg: 0.029354, ktevent: 548
in 10 runs, 2000 epochs, best: 0.000144, worst: 0.058503, avg: 0.029354
UnitTestGenAlgPerf OK
UnitTestGenAlgHistory OK
UnitTestGenAlg OK
UnitTestAll OK
```

UnitTestGenAlgLoadSave.txt:

```
{
  "_type":"0",
  "_nbAdns":"3",
```

```
"_nbElites":"2",
"_lengthAdnF":"2",
"_lengthAdnI":"2",
"_curEpoch":"1",
"_nextId":"6",
"_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":"5",
    "_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"]
    }
  },
  {
    "_id":"4",
    "_age":"1",
    "_elo":"0.000000",
    "_val":"0.000000",
    "_adnF":{
      "_dim":"2",
      "_val":["-0.456306","-0.273161"]
    },
    "_deltaAdnF":{
      "_dim":"2",
      "_val":["0.000000","0.000000"]
    },
    "_adnI":{
      "_dim":"2",
      "_val":["3","5"]
    }
  },
```
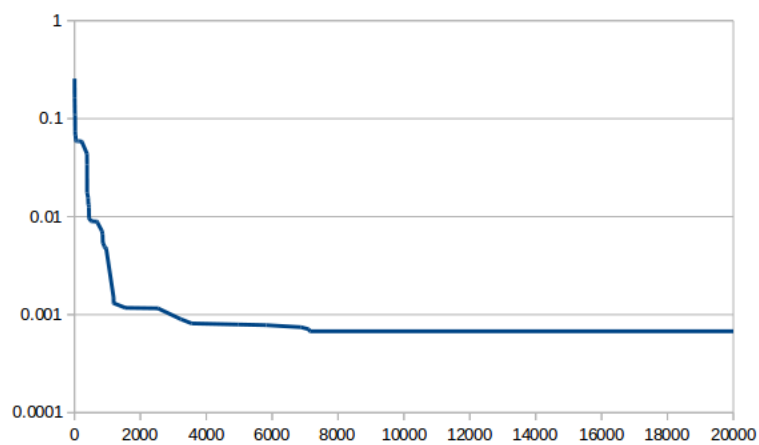
```
{
  "_id":"0",
  "_age":"2",
  "_elo":"0.000000",
  "_val":"0.000000",
  "_adnF":{
    "_dim":"2",
    "_val":["0.788004","-0.003504"]
  },
  "_deltaAdnF":{
    "_dim":"2",
    "_val":["0.000000","0.000000"]
  },
  "_adnI":{
    "_dim":"2",
    "_val":["3","1"]
  }
}
],
"_bestAdn":{
  "_id":"0",
  "_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:

UnitTestGenAlgHistory.json:

{"_genealogy":[{"_epoch":"0","_father":"0","_mother":"0","_id":"0"},{"_epoch":"0","_father":"1","_mother":"1","_id":"