BEAST - Bioinspired Evolutionary Agent Simulation Toolkit Reference Manual

Generated by Doxygen 1.3.2

Sun Feb 1 21:25:33 2004

Contents

Chapter 1

BEAST - Bioinspired Evolutionary Agent Simulation Toolkit Main Page

1.1 Introduction

Welcome to the **BEAST** documentation. Here you will find references for nearly all the classes in the API, along with a few tutorials. Some parts of **BEAST** are separated into their own modules to make navigating the documentation a little easier.

For help navigating the interface, look at **A short user's guide** (includes howto for batch mode).

1.2 Modules

- **Simulation Framework** - The main classes of the API - such as WorldObject, Animat, World, and Simulation.
- **Sensors and Sensor Functions** - **BEAST** has a powerful sensor library, allowing a range of realistic sensors to be easily implemented.
- **Biosystems-Related Classes** - Genetic Algorithms and Neural Networks form an integral part of the simulation environment.
- **Bacteria simulation classes** - An extension for studying bacterial self-organisation and pattern formation.
- **Utilities and Helper Functions** - Extensions to the STL and other utilities used throughout **BEAST**.
- **Serialisation Utilities** - A sub-library enabling you to easily write loading and saving code for your derived classes.

1.3 Tutorials

- **Tutorial 0: A note on the code**

- **Tutorial 1: Building your first Animat**
- **Tutorial 2: Adding Objects and Interactive Animats**
- **Tutorial 3: Introducing the Genetic Algorithm**
- **Tutorial 4: More Advanced Simulations** - Unfinished tutorial on more advanced simulations.

1.4 Example code

The examples and some of the library code are extensively annotated, for ease of access these are included here.

1.4.1 Library code

- **Genetic Algorithm source code**
- **FeedForwardNet source code**
- **DynamicalNet source code**
- **Bacterium source code**

1.4.2 Example code

- **Braitenberg source code**
- **Shrew source code**
- **Mouse source code**
- **Predator/Prey source code**
- **Bacteria example source code**

1.5 Links

The following sites may be useful when developing with **BEAST**.

- `http://www.comp.leeds.ac.uk/ar23/BEAST/index.php` - The **BEAST** homepage
- `http://www.sgi.com/tech/stl/` - SGI's excellent STL reference
- `http://www.cppreference.com/` - A pretty good C++ reference
- `http://groups.google.com/` - Google Groups - without doubt the best place to find answers to C++ problems

Chapter 2

BEAST - Bioinspired Evolutionary Agent Simulation Toolkit Module Index

2.1 BEAST - Bioinspired Evolutionary Agent Simulation Toolkit Modules

Here is a list of all modules:

Chapter 3

BEAST - Bioinspired Evolutionary Agent Simulation Toolkit Namespace Index

3.1 BEAST - Bioinspired Evolutionary Agent Simulation Toolkit Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 4

BEAST - Bioinspired Evolutionary Agent Simulation Toolkit Hierarchical Index

4.1 BEAST - Bioinspired Evolutionary Agent Simulation Toolkit Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 5

BEAST - Bioinspired Evolutionary Agent Simulation Toolkit Compound Index

5.1 BEAST - Bioinspired Evolutionary Agent Simulation Toolkit Compound List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 6

BEAST - Bioinspired Evolutionary Agent Simulation Toolkit File Index

6.1 BEAST - Bioinspired Evolutionary Agent Simulation Toolkit File List

Here is a list of all documented files with brief descriptions:

Chapter 7

BEAST - Bioinspired Evolutionary Agent Simulation Toolkit Page Index

7.1 BEAST - Bioinspired Evolutionary Agent Simulation Toolkit Related Pages

Here is a list of all related documentation pages:

Chapter 8

BEAST - Bioinspired Evolutionary Agent Simulation Toolkit Module Documentation

8.1 Simulation Framework

These classes wrap the interfaces of the GeneticAlgorithm and simulation environment to provide an easy way to set up a range of common simulation types.

Compounds

- class **Animat**

  Animats can move around and interact with other objects in the world.

- class **AnimatMonitor**
- class **Collisions**
- class **Drawable**
- class **Group**

  A simple class which creates and maintains a vector of objects of the specified type and adds them to the world each round.

- class **Population**

  This class is derived from **Group** *and adds a managed GA which is automatically run on the whole* **Population** *every epoch.*

- class **SimObject**

  An abstract base class for the **Population** *template, allowing populations with different templated types to be represented in* **Simulation**.

- class **Simulation**

  The basic **Simulation** *framework which must be derived to set up simulations.*

- class **Trail**
- class **Vector2D**

A class for representing two-dimensional vectors and coordinates.

- class **Wall**

  This is a handy class for putting the most common type of obstacle - walls - into the world.

- class **World**

  This is where it all happens: **World** *contains pointers to every object in the simulation environment and allows those objects to interact with each other, and then be displayed.*

- class **WorldObject**

  The base class for everything that makes a difference in the world, including Animats, Sensors and all types of scenery and interactive object.

Enumerations

- enum **AnimatPartType** { **ANIMAT_BODY**, **ANIMAT_CENTRE**, **ANIMAT_ARROW**, **ANIMAT_WHEEL** }

  Enumeration type for the different coloured parts of the Animat.

- enum **SimPrintStyleType** {

  SIM_PRINT_STATUS, **SIM_PRINT_ASSESSMENT**, **SIM_PRINT_GENERATION**, **SIM_PRINT_RUN**,

  SIM_PRINT_COMPLETE }

  Used in Simulation::ToString to specify what is to be output.

- enum **WorldDisplayType** {

  DISPLAY_NONE = 0, **DISPLAY_ANIMATS** = 1, **DISPLAY_WORLDOBJECTS** = 2, **DISPLAY_TRAILS** = 4,

  DISPLAY_SENSORS = 8, **DISPLAY_COLLISIONS** = 16, **DISPLAY_MONITOR** = 32, **DISPLAY_ALL** = 65535 }

  An enumeration type for specifying which elements of the world are to be displayed.

Functions

- ostream & **operator**<< (ostream &out, const Drawable &d)
- istream & **operator**>> (istream &in, Drawable &d)
- double **deg2rad** (double)

  Converts degrees to radians.

- double **rad2deg** (double)

  Converts radians to degrees.

- Vector2D **operator** * (double l, const Vector2D &v)

  Returns a vector multiplied by a double.

- Vector2D **PolarVector** (double l, double a)

  Creates a **Vector2D** *object using old PolarVector syntax.*

- void **SetColour** (**AnimatPartType**, const float *)

  Sets the colour of the specified.

- void **SetColour** (**AnimatPartType**, float, float, float, float a=1.0f)

  Sets the specified part of the **Animat** *to the specified colour.*

- void **SetColour** (float, float, float, float=1.0f)
- void **SetColour** (const float *)
- std::ostream & **operator<<** (std::ostream &out, const SimObject &obj)

  Output operator for all objects derived from **SimObject**.

- std::istream & **operator>>** (std::istream &in, SimObject &obj)

  Input operator for all objects derived from **SimObject**.

- **Group** (int s=0)

  Sets the **Group** *up with a number of objects of the native type.*

- virtual void **AddToWorld** ()

  Adds the contents of the **Group** *to the* **World**.

- virtual void **Serialise** (std::ostream &) const
- virtual void **Unserialise** (std::istream &)
- bool **operator==** (const Vector2D &) const

  Returns true if this vector is equal to other.

- bool **operator!=** (const Vector2D &) const

  Returns true if this vector is not equal to other.

- Vector2D **operator** * (double) const

  Returns this vector multiplied by l.

- Vector2D & **operator** *= (double)

  Multiplies this vector's length by the specified value.

- Vector2D **operator+** (const Vector2D &) const

  Returns this vector plus other.

- Vector2D & **operator+=** (const Vector2D &)

  Adds other to this vector.

- Vector2D **operator-** (const Vector2D &) const

  Returns this vector minus other.

- Vector2D & **operator-=** (const Vector2D &)

  Subtracts other from this vector.

- Vector2D **operator-** () const

  Negates both values of the vector.

- void **normalise** ()

  Converts the vector into a unit vector with the same angle.

- void **rotate** (double)

  Rotates the vector by the specified number of radians.

- Vector2D **rotation** (double) const

  Returns the vector rotated by the specified number of radians.

- void **SetPolarCoordinates** (double, double)

  Sets the vector up using polar coordinates.

- void **SetLength** (double)

  Sets the length of the vector.

- void **SetAngle** (double)

  Sets the angle of the vector.

- void **SetCartesian** (double, double)

  Quick way of setting X and Y of vector at the same time.

- Vector2D **GetReciprocal** () const

  Returns the opposite vector.

- Vector2D **GetNormalised** () const

  Returns a unit vector with the same angle as the current vector.

- double **dot** (const Vector2D &) const

  Returns the dot product of the vector with other.

- double **GetLength** () const

  Returns the length of the vector, if possible use GetLengthSquared instead.

- double **GetLengthSquared** () const

  Returns the square of the vector's length, useful for quicker comparisons.

- double **GetAngle** () const

  Returns the angle of the vector in radians.

- double **GetGradient** () const

  Returns the gradient of the vector.

- Vector2D **GetPerpendicular** () const

  Returns the perpendicular to the vector/.

- void **Serialise** (std::ostream &) const

  Writes a **Vector2D** *to an output stream.*

- void **Unserialise** (std::istream &)

  Reads a **Vector2D** *from an input stream.*

- std::ostream & **operator<<** (std::ostream &out, const Vector2D &v)

  Output operator for **Vector2D**.

- std::istream & **operator>>** (std::istream &in, Vector2D &v)

  Input operator for **Vector2D**.

- Vector2D **Centre** ()

  Returns the centre coordinates.

- Vector2D **RandomLocation** () const

  Returns a random coordinate.

- void **SetColour** (float r, float g, float b)
- virtual void **AddToWorld** ()

  Adds either the whole population, or the currently selected team to the world.

- virtual void **BeginAssessment** ()

  This method is called at the beginning of the assessment and sets up teams if required.

- virtual void **EndAssessment** ()

  This method is called at the end of the assessment and causes each individual's fitness score to be stored.

- virtual void **BeginGeneration** ()

  This method is called at the beginning of the generation and.

- virtual void **EndGeneration** ()

  Performs GA management, by copying the **Population** *into the GA, running the GA on the population and then retrieving the new generation from the GA.*

- virtual void **BeginRun** ()

  This method is called at the beginning of the run and ensures that the contents of the population has been reset.

- virtual void **EndRun** ()

  This method is called at the end of the run and currently doesn't do anything at all.

- virtual void **Serialise** (std::ostream &) const

  Copies the **Population** *to a stream.*

- virtual void **Unserialise** (std::istream &)

  Converts the data produced by Serialise back into a population.

Variables

- const double **ANIMAT_RADIUS** = 5.0

  Animat's default radius.

- const double **ANIMAT_MAX_SPEED** = 100.0

  Animat's default maximum speed.

- const double **ANIMAT_MIN_SPEED** = -50.0

  Animat's default minimum speed.

- const double **ANIMAT_MAX_ROTATE** = TWOPI

  The default max rotation/frame.

- const double **ANIMAT_DRAG** = 50.0

  An arbitrary friction value.

- const double **ANIMAT_ACCEL** = 5000.0

  AN arbitrary acceleration value.

- const double **ANIMAT_TIMESTEP** = 0.05

  The default time step.

- const int **ANIMAT_PARTS** = 4

  The number of different colours.

- const int **MONITOR_BARHEIGHT** = 25
- const unsigned int **MAX_COLLISIONS** = 200
- const double **DRAWABLE_RADIUS** = 50.0

  The default diameter for drawables.

- const unsigned int **TRAIL_LENGTH** = 30
- const double **WORLD_WIDTH** = 800.0
- const double **WORLD_HEIGHT** = 600.0

8.1.1 Detailed Description

These classes wrap the interfaces of the GeneticAlgorithm and simulation environment to provide an easy way to set up a range of common simulation types.

They also provide an interface to the GUI.

8.1.2 Enumeration Type Documentation

8.1.2.1 enum BEAST::SimPrintStyleType

Used in **Simulation::ToString** to specify what is to be output.

Enumeration values:

SIM_PRINT_STATUS One-line description of simulation position.

SIM_PRINT_ASSESSMENT Output at end of assessment.

SIM_PRINT_GENERATION Output at end of generation.

SIM_PRINT_RUN Output at end of run.

SIM_PRINT_COMPLETE Output at completion of simulation.

8.1.2.2 enum BEAST::WorldDisplayType

An enumeration type for specifying which elements of the world are to be displayed.

See also:
World::Toggle

Enumeration values:
DISPLAY_NONE Nothing is displayed at all.
DISPLAY_ANIMATS Animats are displayed.
DISPLAY_WORLDOBJECTS WorldObjects are displayed.
DISPLAY_TRAILS Trails are displayed.
DISPLAY_SENSORS Sensors are displayed.
DISPLAY_COLLISIONS Collisions are displayed.
DISPLAY_MONITOR The monitor is displayed.
DISPLAY_ALL Everything is displayed.

8.1.3 Function Documentation

8.1.3.1 template<class _Ind, class _MutFunc> void BEAST::Population< _Ind, _MutFunc >::EndGeneration () [virtual, inherited]

Performs GA management, by copying the **Population** into the GA, running the GA on the population and then retrieving the new generation from the GA.

See also:
GeneticAlgorithm
GeneticAlgorithm::Generate

Reimplemented from **BEAST::SimObject** (p. 191).

8.1.3.2 template<class _ObjType> BEAST::Group< _ObjType >::Group (int *s* = 0) [inherited]

Sets the **Group** up with a number of objects of the native type.

Parameters:
s The number of objects to be created.

8.1.3.3 Vector2D PolarVector (double *l*, double *a*) [inline]

Creates a **Vector2D** object using old PolarVector syntax.

Parameters:
l The length of the vector
a The angle of the vector anticlockwise from due east.

Deprecated
Probably of no use to anyone unless they have a strange fascination with polar coordinates.

8.1.3.4 template<class _Ind, class _MutFunc> void BEAST::Population< _Ind, _MutFunc >::Serialise (std::ostream & *out*) const [virtual, inherited]

Copies the **Population** to a stream.

Note that only the GA data and the genotypes of the population are stored.

See also:
Population::Unserialise

Reimplemented from **BEAST::Group< _Ind >** (p. ??).

8.1.3.5 template<class _ObjType> void BEAST::Group< _ObjType >::Serialise (std::ostream & *out*) const [virtual, inherited]

Todo
Finish!

Reimplemented from **BEAST::SimObject** (p. 191).

Reimplemented in **BEAST::Population< _Ind, _MutFunc >** (p. 26).

8.1.3.6 void BEAST::Vector2D::SetPolarCoordinates (double *l*, double *a*) [inline, inherited]

Sets the vector up using polar coordinates.

Parameters:
l The vector's new length.
a The vector's new angle.

8.1.3.7 template<class _Ind, class _MutFunc> void BEAST::Population< _Ind, _MutFunc >::Unserialise (std::istream & *in*) [virtual, inherited]

Converts the data produced by Serialise back into a population.

See also:
Population::Serialise

Reimplemented from **BEAST::Group< _Ind >** (p. ??).

8.1.3.8 template<class _ObjType> void BEAST::Group< _ObjType >::Unserialise (std::istream & *in*) [virtual, inherited]

Todo
Add general object unserialiser

Todo
Rather than removing only objects from this group, all objects with the same type as this group are removed.

Reimplemented from **BEAST::SimObject** (p. 191).

Reimplemented in **BEAST::Population< _Ind, _MutFunc >** (p. 26).

8.2 Bacteria simulation classes

This collection of classes and functors allow the user to implement a variety of simulations involving bacterial chemotaxis, fractal and vortex formations and simple swarming.

Compounds

- class **Bacterium**
- class **Distribution**

  Implements a grid which stores spatial density information to a specified resolution, e.g.

- struct **EvalDensity**
- struct **EvalGradient**
- struct **Gaussian2D**

  Plots a two-dimensional Gaussian function in a distribution or distribution kernel.

- struct **GaussianNoise**

  Plots normally distributed noise in a distribution.

- struct **GaussianRing2D**

  Plots a two dimensional Gaussian ring.

- struct **LimitDistribution**
- struct **Ring2D**

  Plots a two dimensional ring.

- struct **ScaleGradient**
- struct **UniformNoise**

  Plots uniform noise in a distribution.

- struct **ZeroDistribution**

Typedefs

- typedef float **DistReal**

  For speed, Distributions use floats but this typedef makes it possible to switch to doubles if higher accuracy is required.

Functions

- Sensor * **GradientSensor** ()
- Sensor * **DistributionSensor** ()

8.2.1 Detailed Description

This collection of classes and functors allow the user to implement a variety of simulations involving bacterial chemotaxis, fractal and vortex formations and simple swarming.

Note that while some sensors have been included, they are only there for the purpose of letting regular Animats interact with distributions - bacteria have their own streamlined method.

8.3 Utilities and Helper Functions

Compounds

- class **auto_indexed_pointer_property**
- class **auto_indexed_property**
- class **auto_pointer_property**
- class **auto_property**
- struct **bound_mem_fun_t**

  A functor which creates a unary function from a unary member function, binding an instance of the class to which the function belongs.

- class **call_on_mem_t**

  Allows us to bind functors so that they work on particular members of classes, useful for using for_each on maps.

- struct **creator**

  A functor for use with the for_each algorithm which can perform creation of objects when called on a container of pointers.

- struct **deleter**

  A functor for use with the for_each algorithm which can perform deletion of objects when called on a container of pointers.

- class **pointer_property**
- class **property**

  Class wrapper for a member variable which allows member data to be exposed with invisible get/set semantics.

- struct **Random**

  Function object version of randval.

Defines

- #define **M1** 259200
- #define **IA1** 7141
- #define **IC1** 54773
- #define **RM1** (1.0/M1)
- #define **M2** 134456
- #define **IA2** 8121
- #define **IC2** 28411
- #define **RM2** (1.0/M2)
- #define **M3** 243000
- #define **IA3** 4561
- #define **IC3** 51349

Enumerations

- enum **ColourType** {

  COLOUR_BLACK, **COLOUR_WHITE**, **COLOUR_GREEN**, **COLOUR_BLUE**,

  COLOUR_RED, **COLOUR_PURPLE**, **COLOUR_DARK_PURPLE**, **COLOUR_YELLOW**,

  COLOUR_LILAC, **COLOUR_BROWN**, **COLOUR_LIGHT_GREY**, **COLOUR_DARK_GREY**,

  COLOUR_MID_GREY, **COLOUR_ORANGE**, **COLOUR_PINK**, **COLOUR_SELECTION** }

  An enumeration type for colours.

Functions

- const float * **random_colour** ()

  Returns a random colour, all set for input to glColour4fv.

- float **ran1** (int *idum)
- int **rseed** (int *s, bool verbose)
- template<typename Real> Real **randval** (Real limit)

  Returns a (near) uniform distributed random number in the range 0..limit, as a Real.

- int **irand** (int limit)

  Returns a random integer in [0..limit-1].

- bool **brand** (double p)
- bool **brand** (float p)
- template<> int **randval** (int limit)

  Template specialisation to stop randval from being called with ints.

- template<> bool **randval** (bool)
- template<typename Real> Real **gaussrand** (void)

  Returns a normally distributed variable with zero mean and unit variance.

- template<typename Real> Real **normrand** (Real mean, Real sd)

  Returns a random deviate from a normal distribution with specified mean and standard distribution.

- template<class _InIt, class _Ty, class _Fn1> _Ty **accumulate_fun** (_InIt _First, _InIt _Last, _Ty _Val, _Fn1 _Func)

  Version of the STL accumulate algorithm which computes a sum of all the results of a unary function _Func applied to the values between _First and _Last.

- template<class T, typename M, class OP> call_on_mem_t< T, M, OP > **call_on_mem** (M T::*m, OP op)

  Helper function for constructing **call_on_mem_t** *function objects.*

- template<class _Class, typename _Return, typename _Arg> bound_mem_fun_t< _Class, _Return, _Arg > **bound_mem_fun** (_Class &c, _Return(_Class::*memfun)(_Arg))

A helper function for constructing **bound_mem_fun_t** *objects.*

- template<class _Class, typename _Return, typename _Arg> bound_mem_fun_t< _Class, _Return, _Arg > **bound_mem_fun** (_Class *c, _Return(_Class::*memfun)(_Arg))

  A helper function for constructing **bound_mem_fun_t** *objects.*

- template<typename _Type, typename _Base> bool **IsA** (_Base *in, _Type *&out)

  A wrapper for RTTI (RunTime Type Identification) typeid, which checks if two pointers are of identical types.

- template<typename _Type, typename _Base> bool **IsKindOf** (_Base *in, _Type *&out)

  A wrapper for RTTI (RunTime Type Identification) using dynamic_cast which checks if an object is of the same type or is in herited from an object of the same type as an input pointer.

- template<typename T> T **bound** (T L, T U, T n)

  Takes a type, a lower and an upper limit and bounds the input value to those limits.

- template<typename T> void **rbound** (T L, T U, T &n)

  A version of bound which takes a reference as its argument.

- template<typename T> T **limit** (T L, T U, T n)

  Limits the input value to the specified range, clipping at either extreme.

- template<typename T> void **rlimit** (T L, T U, T &n)

  A version of limit which takes a reference as its argument.

Variables

- const float **ColourPalette** [][4]

  A global colour pallete. Could probably do with many more colours.

8.3.1 Enumeration Type Documentation

8.3.1.1 enum BEAST::ColourType

An enumeration type for colours.

Enumeration values:

COLOUR_BLACK Black.

COLOUR_WHITE White.

COLOUR_GREEN Green.

COLOUR_BLUE You get the idea.

8.3.2 Function Documentation

8.3.2.1 template<typename T> T bound (T *L*, T *U*, T *n*) [inline]

Takes a type, a lower and an upper limit and bounds the input value to those limits.

Useful for angles.

Parameters:

T The type of value.

L The lower limit.

U The upper limit.

n The input value.

Returns:

The output value.

8.3.2.2 template<typename Real> Real gaussrand (void) [inline]

Returns a normally distributed variable with zero mean and unit variance.

Recall that the absolute value will be >3 about once in 400 trials (the three-sigma rule). This is adapted from the "Numerical recipies in C" book by Press, Flannery, Teukolsky, and Vetterling, p.217

8.3.2.3 template<typename _Type, typename _Base> bool IsA (_Base * *in*, _Type *& *out*) [inline]

A wrapper for RTTI (RunTime Type Identification) typeid, which checks if two pointers are of identical types.

Parameters:

in A pointer to the object being tested.

out A pointer of the type required, which will be set to point to the same object as the first parameter if it turns out they are of identical types.

Returns:

True if a match is made.

See also:

IsKindOf

8.3.2.4 template<typename _Type, typename _Base> bool IsKindOf (_Base * *in*, _Type *& *out*) [inline]

A wrapper for RTTI (RunTime Type Identification) using dynamic_cast which checks if an object is of the same type or is in herited from an object of the same type as an input pointer.

Parameters:

in A pointer to the object being tested.

out A pointer of the type being tested for, which will be set to point to the same object as the first paramter if it turns out it is of the same or an inherited type.

Returns:
True if a match is made.

8.3.2.5 template<typename T> T limit (T *L*, T *U*, T *n*) [inline]

Limits the input value to the specified range, clipping at either extreme.

Parameters:
T The type of values to work on.

L The lower limit.

U The upper limit.

n The input value.

Returns:
The output value.

8.3.2.6 template<typename T> void rbound (T *L*, T *U*, T & *n*) [inline]

A version of bound which takes a reference as its argument.

Parameters:
T The type of value.

L The lower limit.

U The upper limit.

n The input value.

8.3.2.7 template<typename T> void rlimit (T *L*, T *U*, T & *n*) [inline]

A version of limit which takes a reference as its argument.

Parameters:
T The type of values to work on.

L The lower limit.

U The upper limit.

n The input value.

Returns:
The output value.

8.3.3 Variable Documentation

8.3.3.1 const float BEAST::ColourPalette[][4]

Initial value:

```
{
    {   0.0f, 0.0f, 0.0f, 1.0f  },
    {   1.0f, 1.0f, 1.0f, 1.0f  },
    {   0.2f, 0.8f, 0.2f, 1.0f  },
    {   0.2f, 0.2f, 0.8f, 1.0f  },
    {   0.8f, 0.2f, 0.2f, 1.0f  },
    {   0.5f, 0.3f, 0.7f, 1.0f  },
    {   0.2f, 0.0f, 0.4f, 1.0f  },
    {   0.8f, 0.8f, 0.2f, 1.0f  },
    {   0.8f, 0.5f, 0.9f, 1.0f  },
    {   0.4f, 0.3f, 0.1f, 1.0f  },
    {   0.8f, 0.8f, 0.8f, 1.0f  },
    {   0.3f, 0.3f, 0.3f, 1.0f  },
    {   0.5f, 0.5f, 0.5f, 1.0f  },
    {   0.9f, 0.9f, 0.1f, 1.0f  },
    {   1.0f, 0.8f, 0.8f, 1.0f  },
    {   0.5f, 0.5f, 1.0f, 0.5f  }
}
```

A global colour pallete. Could probably do with many more colours.

8.4 Biosystems-Related Classes

This collection of classes implement a range of biosystems algorithms and control systems including a multi-purpose genetic algorithm and two neural nets.

Compounds

- class **DynamicalNet**

  This class implements a fully recurrent continuous (or dynamical) neural network.

- class **EvalNearestSignal**

  Sensor *evaluation functor: returns the signal of the nearest individual.*

- class **Evolver**
- class **FeedForwardNet**

  This is an implementation of a simple two-layer feed-forward neural network.

- struct **GAVariant**

  This is a general purpose data type which takes five basic data types: int, float, double, char and bool.

- class **GeneticAlgorithm**

  The **GeneticAlgorithm** *class provides functionality to cover a range of GA methods, and may be extended to incorporate other approaches.*

- struct **MutationOperator**
- struct **MutationOperator< bool >**

  Specialised **MutationOperator** *for bool, simply NOT's its input.*

- struct **MutationOperator< GAVariant >**

  This specialised mutation operator provides the facilities of the basic **MutationOperator** *for* **GAVariant**.

- struct **NormalMutator**
- class **PSOAlgorithm**
- class **Signaller**

  A general-purpose class for modelling signallers with discrete signal and state types.

Enumerations

- enum **GASelectionType** { **GA_ROULETTE** = 0, **GA_RANK**, **GA_TOURNAMENT** }

  The different options for selection are enumerated here.

- enum **GAFltParamType** { **GA_TOURNAMENT_PARAM**, **GA_RANK_SPRESSURE**, **GA_EXPONENT** }

  Assorted float parameters, set using GeneticAlgorithm::SetParameter.

- enum **GAIntParamType** { **GA_TOURNAMENT_SIZE** }

Assorted integer parameters, set using GeneticAlgorithm::SetParameter.

- enum **GAPrintStyleType** { **GA_PARAMETERS** = 1, **GA_CURRENT** = 2, **GA_GENERATION** = 4, **GA_HISTORY** = 8 }

  Use to set the printing style when using GA's << operator.

- enum **GAFitnessMethodType** { **GA_BEST_FITNESS**, **GA_WORST_FITNESS**, **GA_MEAN_FITNESS**, **GA_TOTAL_FITNESS** }

  The method by which fitness is decided when individuals have multiple scores.

- enum **GAFitnessFixType** { **GA_IGNORE**, **GA_CLAMP**, **GA_FIX** }

  Sets the method by which fitness scores are adjusted before selection.

- enum **GAVariantType** {
  GAV_INT, **GAV_FLOAT**, **GAV_DOUBLE**, **GAV_CHAR**,
  GAV_BOOL }

  A type flag for the GAVariant data type.

Functions

- ostream & **operator**<< (ostream &out, const DynamicalNet &dnn)

  An output operator for **DynamicalNet**.

- istream & **operator**>> (istream &in, DynamicalNet &dnn)

  An input operator for **DynamicalNet**.

- ostream & **operator**<< (ostream &out, const FeedForwardNet &ffn)

  Output operator overload for the **FeedForwardNet**.

- istream & **operator**>> (istream &in, FeedForwardNet &ffn)

  Input operator overload for the **FeedForwardNet**.

- template<typename T> std::ostream & **operator**<< (std::ostream &out, const MutationOperator< T > &m)

  Output operator for the **MutationOperator** *function object.*

- template<typename T> std::istream & **operator**>> (std::istream &in, MutationOperator< T > &m)

  Input operator for the **MutationOperator** *function object.*

- template<class EVO, class MUTFUNC> std::ostream & **operator**<< (std::ostream &, const GeneticAlgorithm< EVO, MUTFUNC > &)

  The GA's output operator.

- template<class EVO, class MUTFUNC> std::istream & **operator**>> (std::istream &, const GeneticAlgorithm< EVO, MUTFUNC > &)

  The GA's input operator.

- template<class EVO, class MUTFUNC> std::istream & **operator>>** (std::istream &in, GeneticAlgorithm< EVO, MUTFUNC > &ga)
- std::ostream & **operator<<** (std::ostream &out, const GAVariant &v)
- std::istream & **operator>>** (std::istream &in, GAVariant &v)
- float **RandomNum** ()

  Returns a number between -1 and 1.

- **GeneticAlgorithm** (float crossover=0.7f, float mutation=0.01f, int popSize=0)

  Constructor: sets default parameters: elitism and subelitism 0, crossover points 1, tournament parameter 0.75, tournament size 2, rank pressure 1.5, exponent 1.0.

- std::vector< EVO * > **GetPopulationCopy** () const

  Returns duplicates of the output population which won't be deleted by the GA if it does a **CleanUp()**.

- virtual void **Generate** ()

  The generation function: this is where it all happens.

- void **ReGenerate** ()

  Calls the **Generate()** *method and copies the output population into the input.*

- void **CalcStats** ()

  Calculates some statistics used in some of the selection methods and also stored by the class for data collection.

- void **Setup** ()

  Prepares the GA for the next epoch.

- void **FixFitness** ()

  Adjusts the fitness according to.

- float **GetFitness** (EVO *i)

  Calculates the fitness score to be used by the GA from the stored fitness scores in the EVO object.

- void **SelectParentGenotype** (GENOTYPE &)

  Depending on the selection procedure chosen, a genotype is taken from the population and returned by reference.

- void **SelectProbability** (GENOTYPE &)

  Roulette and Rank Selection Having done the Setup function (above), each individual has a probability score, derived from their rank or fitness.

- void **SelectTournament** (GENOTYPE &)

  Tournament Selection The method implemented here is an amalgamation of two slightly different approaches.

- void **CrossoverGenotypes** (GENOTYPE &, GENOTYPE &)

  This method simply takes two chromosomes, mum and dad, and swaps them over at a random point along the length.

- void **MutateGenotype** (GENOTYPE &)

While crossover simulates the effect of sexual reproduction within a population, mutation artificially reproduces the effects of transcription errors in the replication of DNA.

- std::string **ToString** () const

  Returns a string containing various details about the GA's current state, depending on what has been set with SetPrintStyle.

- std::string **GetCSV** (char separator= ',') const

  Returns a string containing a simple CSV table with average and best fitness for every generation so far.

- void **CleanUp** ()

  Deletes the objects comprising the input and output populations.

- void **Serialise** (std::ostream &) const
- void **Unserialise** (std::istream &)
- void **Randomise** (int numStates, int numSignals)

  Randomises the signaller so that each possible internal state has a random signal associated with it.

- virtual void **Generate** ()
- EVO * **Fly** (EVO *)

Variables

- const double **FFN_ACTIVATION_RESPONSE** = 1

  This value decides the curve of the sigmoid function.

- const int **FFN_COLSIZE** = 6

  The width of columns in ToString output.

8.4.1 Detailed Description

This collection of classes implement a range of biosystems algorithms and control systems including a multi-purpose genetic algorithm and two neural nets.

There are also classes for using these controllers in the simulation environment.

8.4.2 Enumeration Type Documentation

8.4.2.1 enum BEAST::GAFitnessFixType

Sets the method by which fitness scores are adjusted before selection.

Enumeration values:

GA_IGNORE Leave them as they are.

GA_CLAMP Minimum fitness is clamped at 0.

GA_FIX Worst score set to 0, others scaled up.

8.4.2.2 enum BEAST::GAFitnessMethodType

The method by which fitness is decided when individuals have multiple scores.

Enumeration values:

GA_BEST_FITNESS The best fitness score is used.

GA_WORST_FITNESS The worst fitness score is used.

GA_MEAN_FITNESS The average fitness is used.

GA_TOTAL_FITNESS The total fitness is used.

8.4.2.3 enum BEAST::GAFltParamType

Assorted float parameters, set using **GeneticAlgorithm::SetParameter**.

Enumeration values:

GA_TOURNAMENT_PARAM Probability of fittest winning in tournament selection.

GA_RANK_SPRESSURE Selection pressure for rank selection, [1.0:2.0].

GA_EXPONENT An expontial modifier for rank and roulette probabilities.

8.4.2.4 enum BEAST::GAIntParamType

Assorted integer parameters, set using **GeneticAlgorithm::SetParameter**.

Enumeration values:

GA_TOURNAMENT_SIZE Number of individuals per round of tournament selection.

8.4.2.5 enum BEAST::GAPrintStyleType

Use to set the printing style when using GA's << operator.

Enumeration values:

GA_PARAMETERS Display the current parameters.

GA_CURRENT Display details of current generation.

GA_GENERATION Display all members of current generation.

GA_HISTORY Show average and best fitnesses of each generation.

8.4.2.6 enum BEAST::GASelectionType

The different options for selection are enumerated here.

Enumeration values:

GA_ROULETTE Fitness proportional selection.

GA_RANK Rank proportional selection.

GA_TOURNAMENT Tournament selection.

8.4.2.7 enum BEAST::GAVariantType

A type flag for the **GAVariant** data type.

See also:
GAVariant

8.4.3 Function Documentation

8.4.3.1 template<class EVO, class MUTFUNC> void BEAST::GeneticAlgorithm< EVO, MUTFUNC >::CleanUp () [inherited]

Deletes the objects comprising the input and output populations.

Only called if the **GeneticAlgorithm** has ownership of its data.

See also:
SetOwnsData

8.4.3.2 template<class EVO, class MUTFUNC> void BEAST::GeneticAlgorithm< EVO, MUTFUNC >::CrossoverGenotypes (GENOTYPE & *mum*, GENOTYPE & *dad*) [protected, inherited]

This method simply takes two chromosomes, mum and dad, and swaps them over at a random point along the length.

This, therefore, is the sexual part of the genetic algorithm

8.4.3.3 template<class EVO, class MUTFUNC> void BEAST::GeneticAlgorithm< EVO, MUTFUNC >::Generate () [virtual, inherited]

The generation function: this is where it all happens.

First the total, best, worst and average fitnesses are calculated with **CalcStats()**. Then fitness scaling and probability distributions are done by **Setup()**. Elitism and subelitism are dealt with and then the new population is generated using the current selection method, crossover parameters and mutation operator.

Todo
Account for odd numbered output population sizes

8.4.3.4 template<class EVO, class MUTFUNC> std::string BEAST::Genetic-Algorithm< EVO, MUTFUNC >::GetCSV (char *separator* = ',') const [inherited]

Returns a string containing a simple CSV table with average and best fitness for every generation so far.

The default separator is "," but a different one may be specified.

8.4.3.5 template<class EVO, class MUTFUNC> float BEAST::GeneticAlgorithm< EVO, MUTFUNC >::GetFitness (EVO * *i*) [protected, inherited]

Calculates the fitness score to be used by the GA from the stored fitness scores in the EVO object.

If no scores have been stored, the output from EVO::GetFitness is returned.

8.4.3.6 template<typename T> std::ostream& operator<< (std::ostream & *out*, const MutationOperator< T > & *m*)

Output operator for the **MutationOperator** function object.

Note that in order for this function to work, there must exist an output operator for the mutation operator's type.

See also:
IMPLEMENT_IOSTREAM_CAST
IMPLEMENT_IOSTREAM_BINARY_CONVERSION

8.4.3.7 ostream& operator<< (ostream & *out*, const FeedForwardNet & *ffn*)

Output operator overload for the **FeedForwardNet**.

Parameters:
out An output stream.
ffn A **FeedForwardNet** object.

Returns:
A reference to the output stream.

See also:
FeedForwardNet::Serialise

8.4.3.8 ostream& operator<< (ostream & *out*, const DynamicalNet & *dnn*)

An output operator for **DynamicalNet**.

Parameters:
out A reference to an output stream.
dnn A reference to a **DynamicalNet**.

Returns:
A reference to an output stream.

8.4.3.9 template<typename T> std::istream& operator>> (std::istream & *in*, MutationOperator< T > & *m*)

Input operator for the **MutationOperator** function object.

Note that in order for this function to work, there must exist an input operator for the mutation operator's type.

See also:
IMPLEMENT_IOSTREAM_CAST
IMPLEMENT_IOSTREAM_BINARY_CONVERSION

8.4.3.10 istream& operator>> (istream & *in*, FeedForwardNet & *ffn*)

Input operator overload for the **FeedForwardNet**.

Parameters:
in An input stream.

ffn A **FeedForwardNet** object.

Returns:
A reference to the input stream.

See also:
FeedForwardNet::Unserialise

8.4.3.11 istream& operator>> (istream & *in*, DynamicalNet & *dnn*)

An input operator for **DynamicalNet**.

Parameters:
in A reference to an input stream.

dnn A reference to a **DynamicalNet**.

Returns:
A reference to an input stream.

8.4.3.12 template<typename _State, typename _Signal, typename _Cost> void BEAST::Signaller< _State, _Signal, _Cost >::Randomise (int *numStates*, int *numSignals*) [inline, inherited]

Randomises the signaller so that each possible internal state has a random signal associated with it.

Also sets the signaller to a random state value.

8.4.3.13 template<class EVO, class MUTFUNC> void BEAST::GeneticAlgorithm< EVO, MUTFUNC >::SelectProbability (GENOTYPE & *chromo*) [protected, inherited]

Roulette and Rank Selection Having done the Setup function (above), each individual has a probability score, derived from their rank or fitness.

Imagine all the probability scores of the population as a big pie chart, printed on a roulette wheel. Now imagine numbers around the edge, starting at 0, going up to the total probability of selection. We pick a random number between 0 and 1, and call it slice. This is where our roulette ball will land.

8.4.3.14 template<class EVO, class MUTFUNC> void BEAST::GeneticAlgorithm< EVO, MUTFUNC >::SelectTournament (GENOTYPE & *chromo*) [protected, inherited]

Tournament Selection The method implemented here is an amalgamation of two slightly different approaches.

GA_TOURNAMENT_SIZE individuals are selected at random from the population. With GA_TOURNAMENT_PARAM probability, the fittest individual is picked, otherwise a random individual (perhaps still the fittest) is chosen from the tournament.

8.4.3.15 template<class EVO, class MUTFUNC> void BEAST::GeneticAlgorithm< EVO, MUTFUNC >::Setup () [protected, inherited]

Prepares the GA for the next epoch.

Output population is cleared, input population is sorted by fitness, population probabilities are set according to the selection method in use, the chromosome length is determined, etc.

TODO: perhaps make chromoLength the shortest of any given pair?

8.4.3.16 template<class EVO, class MUTFUNC> std::string BEAST::GeneticAlgorithm< EVO, MUTFUNC >::ToString () const [inherited]

Returns a string containing various details about the GA's current state, depending on what has been set with SetPrintStyle.

Options include: GA_PARAMETERS: print the current parameters GA_CURRENT: print stats for the current generation GA_GENERATION: output the current generation GA_HISTORY: print the history of average and best fitness

Todo

store time data

8.5 Sensors and Sensor Functions

Sensors are set up to provide maximum flexibility, so if you need a new sensor which isn't that different to existing ones, you will need to write a minimum of code to put that sensor together.

Compounds

- class **AreaSensor**

  Detects objects within an area specified by the size and shape of the **AreaSensor**.

- class **BeamSensor**

  BeamSensors can really be three distinct kinds of sensor: Lasers, which just detect objects a certain distance away in a straight line from the sensor's origin.

- class **EvalCount**
- class **EvalNearest**

  Keeps a tally of the nearest point passed in and returns it with GetOutput.

- class **EvalNearestAbsX**

  Returns the absolute x position of the nearest target.

- class **EvalNearestAbsY**

  Returns the absolute y position of the nearest target.

- class **EvalNearestAngle**

  Returns the normalised angle to the nearest target.

- class **EvalNearestSignal**

  Sensor *evaluation functor: returns the signal of the nearest individual.*

- class **EvalNearestXDist**

  Returns the vertical distance to the nearest target.

- class **EvalNearestYDist**

  Returns the horizontal distance to the nearest target.

- struct **MatchAdapter**

  Allows any unary predicate to be adapted for use as a matching function.

- struct **MatchComposeAnd**

  Chains any number of matching functions together such that only if all of them are true for the object being matched, **MatchComposeAnd** *will return true.*

- struct **MatchComposeOr**

  Chains any number of matching functions together such that should any of them be true for the object being matched, **MatchComposeOr** *will return true.*

- struct **MatchExact**

  Identifies exact object types, so if defined with Cheese, will return true only for Cheese, and false for Cheddar and Gruyère.

- struct **MatchKindOf**

  Identifies objects belonging to hierarchies, so if defined with Cheese, will return true for objects of type Cheese, or derived classes such as Cheddar and Gruyère.

- struct **MatchSpecific**

  Identifies one particular object and returns true only for that object.

- struct **ScaleAbs**

  Returns the absolute value of the input, as for the std::abs function.

- struct **ScaleAdapter**

  Allows any unary functor to be adapted for use as a scaling function.

- struct **ScaleCompose**

  ScaleCompose *allows the chaining of two scaling functions together, such the output of a* **ScaleCompose** *functor is the result of second(first(input)), where first and second are the arguments in ScaleCompose's constructor.*

- struct **ScaleLinear**

  A simple linear scaling function which defaults to an input scale between 0 and a defined maximum, scaling to an output range between 0 and 1.

- struct **ScaleNoise**

  ScaleNoise *adds uniform random noise to its input.*

- struct **ScaleThreshold**

  ScaleThreshold *takes values: threshold, min and max and returns min if input < threshold, or max if input >= threshold.*

- class **SelfSensor**

  The **SelfSensor** *is used to detect information about its owner.*

- class **Sensor**

  The **Sensor** *class is the base class for all the different types of sensor:* **TouchSensor**, **SelfSensor**, **AreaSensor** *and* **BeamSensor**.

- struct **SensorEvalFunction**

  Abstract base class for evaluation functors.

- struct **SensorMatchFunction**

  Abstract base class for matching functors.

- struct **SensorScaleFunction**

  Abstract base class for scaling functors.

- class **TouchSensor**

  Detects objects which are touching the sensor's owner.

Enumerations

- enum **SelfSensorType** { **SELF_SENSOR_X**, **SELF_SENSOR_Y**, **SELF_SENSOR_ANGLE**, **SELF_SENSOR_CONTROL** }

  An enumeration type for SelfSensor, used to specify which feature of the sensor's owner is to be returned by GetOutput().

Functions

- template<class T> Sensor * **ProximitySensor** (double scope, double range, double orientation)

  Creates a segment-shaped sensor with the specified scope, range and orientation which detects the distance of objects of the specified templated type.

- template<class T> Sensor * **NearestAngleSensor** ()
- template<class T> Sensor * **NearestXSensor** ()
- template<class T> Sensor * **NearestYSensor** ()
- template<class T> Sensor * **DensitySensor** (double scope, double range, double orientation)
- template<class T> Sensor * **CollisionSensor** ()
- template<class _Functor> MatchAdapter< _Functor > * **MatchAdapt** (_Functor f)

  A helper function for creating **MatchAdapter** *functors.*

- template<class T> Sensor * **NearestSignalSensor** (int highestSignal)

  Constructs and returns a pointer to a sensor which will return the signal of the nearest **Signaller** *of the specified type.*

- ScaleAdapter< _Functor > * **ScaleAdapt** (_Functor f)

  A helper function for creating **ScaleAdapter** *functors.*

Variables

- const float **SENSOR_ALPHA** = 0.1f

  Transparency value for Sensors.

- const double **BEAM_SENSOR_SCOPE** = PI/4

  The default scope for **BeamSensor**.

- const double **BEAM_SENSOR_RANGE** = 250.0

  The default range for **BeamSensor**.

- const float **BEAM_DRAW_QUALITY** = 0.1f

  The default draw quality for **BeamSensor**.

8.5.1 Detailed Description

Sensors are set up to provide maximum flexibility, so if you need a new sensor which isn't that different to existing ones, you will need to write a minimum of code to put that sensor together.

8.5.2 Enumeration Type Documentation

8.5.2.1 enum BEAST::SelfSensorType

An enumeration type for **SelfSensor**, used to specify which feature of the sensor's owner is to be returned by GetOutput().

Enumeration values:

SELF_SENSOR_X Returns the x coordinate of the owner.

SELF_SENSOR_Y Returns the y coordinate of the owner.

SELF_SENSOR_ANGLE Returns the owner's orientation.

SELF_SENSOR_CONTROL Returns a control value, specified in constructor.

8.5.3 Function Documentation

8.5.3.1 template<class T> Sensor* NearestSignalSensor (int *highestSignal*)

Constructs and returns a pointer to a sensor which will return the signal of the nearest **Signaller** of the specified type.

Parameters:

T The type of object to detect (must be derived from **Signaller**).

highestSignal The highest value a signal might take, for scaling.

8.5.3.2 template<class T> Sensor* ProximitySensor (double *scope*, double *range*, double *orientation*)

Creates a segment-shaped sensor with the specified scope, range and orientation which detects the distance of objects of the specified templated type.

Parameters:

T The type of objects to detect. The **MatchKindOf** functor is used by default.

scope The angle within which to detect objects. An angle of 0 may be specified to create a 'laser' type sensor, or an angle of TWOPI will create an all-round distance sensor.

8.6 Serialisation Utilities

Currently, the simplest way to store data is by serialisation - simply saving and loading object data as unformatted text.

Compounds

- struct **extractor**

  This is a function object which can be used for copying from an iterator when the number of input values is unknown.

- struct **ObjLoader**

  A functor for recreating templated object types using serialisation.

- struct **ObjLoaderBase**

  A simple abstract base class for **ObjLoader** *functors.*

- struct **SerialException**

  Since exceptions have an undesirable overhead, they have not been used elsewhere in the simulation environment for reasons of speed.

- struct **switcher**

  The switcher is useful when configuring bools from string data.

- class **Unserialiser**

  This class is available for unserialising objects from streams, without knowing which type of object is to be unserialised - the type is determined from the header of the stream.

Defines

- #define **IMPLEMENT_IOSTREAM_CAST**(_Ty, _Cast)

  Use this macro to create a custom output operator which simply casts _Ty to the type specified by _Cast.

- #define **IMPLEMENT_IOSTREAM_BINARY_CONVERSION**(_Ty)

  This macro is intended as a quick solution to the problem of encoding structs and classes into output streams.

- #define **IMPLEMENT_SERIALISATION**(_Name, _Parent)

  Use this macro to add basic serialisation functionality to your derived classes by simply serialising under a new name, but using the parent serialisation methods.

- #define **IMPLEMENT_LOADER**(_Name, _Type) Unserialiser::Instance().Add(_Name, new ObjLoader<_Type>());

Enumerations

- enum **SerialErrorType** {

  SERIAL_ERROR_UNKNOWN, **SERIAL_ERROR_BAD_FILE**, **SERIAL_ERROR_WRONG_TYPE**, **SERIAL_ERROR_UNKNOWN_TYPE**,

  SERIAL_ERROR_DATA_MISMATCH }

  Enumerates the different types of errors encountered in serialisation.

Functions

- std::string **add_slashes** (const std::string &str)
- std::string **strip_slashes** (const std::string &str)
- template<class _Iterator> extractor< _Iterator > **extract** (_Iterator &Iter)

  Constructs and returns an extractor of the correct type.

- template<class _InIt, class _OutIt> _InIt **copy_from** (_OutIt _First, _OutIt _Last, _InIt _Src)

  An algorithm similar to copy, but where the start and end of the target range is specified rather than the source range.

- template<class _OutIt> std::istream & **copy_from_istream** (_OutIt _First, _OutIt _Last, std::istream &in)

  Although extract and copy_from should, with the help of istream_iterator, be able to fill ranges from input streams, certain difficulties with istreams arise which make the copy_from_istream function useful.

- std::istream & **operator>>** (std::istream &in, switcher s)

  Input operator for the switcher helper object.

- template<typename _Ty, typename _Ax> std::ostream & **operator<<** (std::ostream &out, const std::vector< _Ty, _Ax > &v)

  A generic output operator for vectors, requires the vector's contents type to have its own << operator.

- template<typename _Ty, typename _Ax> std::istream & **operator>>** (std::istream &in, std::vector< _Ty, _Ax > &v)

  A generic input operator for vectors, requires the vector's contents type to have its own >> operator.

- template<typename _Ty, typename _Pr, typename _Alloc> std::ostream & **operator<<** (std::ostream &out, const std::map< std::string, _Ty, _Pr, _Alloc > &m)

  A specialised output operator for maps with key type string, which uses add_slashes to encode the string.

- template<typename _Ty, typename _Pr, typename _Alloc> std::istream & **operator>>** (std::istream &in, const std::map< std::string, _Ty, _Pr, _Alloc > &m)

  A specialised input operator for maps with key type string, which uses strip_slashes to decode the string.

- template<typename _Kty, typename _Ty, typename _Pr, typename _Alloc> std::ostream & **operator**<< (std::ostream &out, const std::map< _Kty, _Ty, _Pr, _Alloc > &m)

  A generic output operator for maps, requires the map's contents to have its own << operator.

- template<typename _Kty, typename _Ty, typename _Pr, typename _Alloc> std::istream & **operator**>> (std::istream &in, std::map< _Kty, _Ty, _Pr, _Alloc > &m)

  A generic input operator for maps, requires the map's contents to have its own >> operator.

- template<typename T1, typename T2> T1 & **stream_convert** (T2 &input)

  An inline reinterpret cast which may be helpful in outputting enumeration types to streams.

8.6.1 Detailed Description

Currently, the simplest way to store data is by serialisation - simply saving and loading object data as unformatted text.

To make it easier for you to implement serialisation for your own classes, a range of functions, classes and macros are provided.

8.6.2 Define Documentation

8.6.2.1 #define IMPLEMENT_IOSTREAM_BINARY_CONVERSION(_Ty)

Value:

```
inline std::ostream& operator<<(std::ostream& out, _Ty& i)  \
{                                                           \
    out.write(reinterpret_cast<char*>(&i), sizeof(i));      \
    return out;                                             \
}                                                           \
                                                            \
inline std::istream& operator>>(std::istream& in, _Ty& i)   \
{                                                           \
    in.read(reinterpret_cast<char*>(&i), sizeof(i));        \
    return in;                                              \
}
```

This macro is intended as a quick solution to the problem of encoding structs and classes into output streams.

A binary conversion is performed, meaning that any pointers and complex data types (e.g. vectors) will be rendered nonsensical to the input operator. Only for use on simple structs and classes containing non-pointer types. Also, output produced by these functions is unlikely to transfer between platforms due to differing number formats. This is only a temporary solution and should be replaced with special input/output operators.

8.6.2.2 #define IMPLEMENT_IOSTREAM_CAST(_Ty, _Cast)

Value:

```
inline std::ostream& operator<<(std::ostream& out, _Ty& i)  \
{                                                           \
    out << static_cast<_Cast>(i);                           \
```

```
    return out;                                                 \
}                                                               \
                                                                \
inline std::istream& operator>>(std::istream& in, _Ty& i)       \
{                                                               \
    _Cast input;                                                \
    in >> input;                                                \
    i = static_cast<_Ty>(input);                                \
    return in;                                                  \
}
```

Use this macro to create a custom output operator which simply casts _Ty to the type specified by _Cast.

Useful for converting enum types to ints which may then be input and output in the usual way.

8.6.2.3 #define IMPLEMENT_SERIALISATION(_Name, _Parent)

Value:

```
public:                                                         \
    void Serialise(std::ostream& out)const                      \
    {                                                           \
        out << _Name << "\n";                                   \
        _Parent::Serialise(out);                                \
    }                                                           \
                                                                \
    void Unserialise(std::istream& in)                          \
    {                                                           \
        std::string name;                                       \
        in >> name;                                             \
        if (name != _Name) {                                    \
            throw SerialException(SERIAL_ERROR_WRONG_TYPE, name,    \
                    std::string("This object is type ") + _Name);   \
        }                                                       \
        _Parent::Unserialise(in);                               \
    }
```

Use this macro to add basic serialisation functionality to your derived classes by simply serialising under a new name, but using the parent serialisation methods.

8.6.3 Enumeration Type Documentation

8.6.3.1 enum BEAST::SerialErrorType

Enumerates the different types of errors encountered in serialisation.

Enumeration values:

SERIAL_ERROR_UNKNOWN An unknown problem.

SERIAL_ERROR_BAD_FILE Unable to open or write to the file.

SERIAL_ERROR_WRONG_TYPE Incoming data has the wrong type label.

SERIAL_ERROR_UNKNOWN_TYPE Incoming data has an unknown type label.

SERIAL_ERROR_DATA_MISMATCH The wrong type of data seems to be coming in.

8.6.4 Function Documentation

8.6.4.1 template<class _InIt, class _OutIt> _InIt copy_from (_OutIt *_First*, _OutIt *_Last*, _InIt *_Src*)

An algorithm similar to copy, but where the start and end of the target range is specified rather than the source range.

For example, to copy from a vector of unknown length into a range of known length: `vector<int> target(30); vector<int> source(rand(100) + 30); copy_from(target.begin(), target.end(), source.begin());`

See also:
extract for another way of doing exactly the same thing.

8.6.4.2 template<class _OutIt> std::istream& copy_from_istream (_OutIt *_First*, _OutIt *_Last*, std::istream & *in*)

Although extract and copy_from should, with the help of istream_iterator, be able to fill ranges from input streams, certain difficulties with istreams arise which make the copy_from_istream function useful.

Simply specify the start and end iterators of the range to be filled and an input stream to use as the source.

8.6.4.3 template<class _Iterator> extractor<_Iterator> extract (_Iterator & *Iter*)

Constructs and returns an extractor of the correct type.

For example, to copy from a vector of unknown length into a range of known length: `vector<int> target(30); vector<int> source(rand(100) + 30); generate(target.begin(), target.end(), extract(source.begin()));`

See also:
extractor

8.6.4.4 template<typename _Kty, typename _Ty, typename _Pr, typename _Alloc> std::ostream& operator<< (std::ostream & *out*, const std::map< _Kty, _Ty, _Pr, _Alloc > & *m*) [inline]

A generic output operator for maps, requires the map's contents to have its own << operator.

The size is output, followed by each key and value, separated by spaces.

8.6.4.5 template<typename _Ty, typename _Pr, typename _Alloc> std::ostream& operator<< (std::ostream & *out*, const std::map< std::string, _Ty, _Pr, _Alloc > & *m*) [inline]

A specialised output operator for maps with key type string, which uses add_slashes to encode the string.

See also:
add_slashes

8.6.4.6 template<typename _Ty, typename _Ax> std::ostream& operator<< (std::ostream & *out*, const std::vector< _Ty, _Ax > & *v*) [inline]

A generic output operator for vectors, requires the vector's contents type to have its own << operator.

The size is output, followed by each entry, separated by spaces.

8.6.4.7 template<typename _Kty, typename _Ty, typename _Pr, typename _Alloc> std::istream& operator>> (std::istream & *in*, std::map< _Kty, _Ty, _Pr, _Alloc > & *m*) [inline]

A generic input operator for maps, requires the map's contents to have its own >> operator.

The size is read, followed by each key and value.

8.6.4.8 template<typename _Ty, typename _Ax> std::istream& operator>> (std::istream & *in*, std::vector< _Ty, _Ax > & *v*) [inline]

A generic input operator for vectors, requires the vector's contents type to have its own >> operator.

The size is input, followed by each entry.

8.6.4.9 std::istream& operator>> (std::istream & *in*, switcher *s*) [inline]

Input operator for the switcher helper object.

Note that unlike most input operators, this one does not take a reference to the switcher, since the switcher is intended for instantiation at the time of input.

8.6.4.10 template<typename T1, typename T2> T1& stream_convert (T2 & *input*) [inline]

An inline reinterpret cast which may be helpful in outputting enumeration types to streams.

Probaly better to use IMPLEMENT_IOSTREAM_CAST or ideally, write your own input/output operators for your types.

Chapter 9

BEAST - Bioinspired Evolutionary Agent Simulation Toolkit Namespace Documentation

9.1 BEAST Namespace Reference

The namespace for everything in the simulation environment.

Compounds

- class **Animat**

  Animats can move around and interact with other objects in the world.

- class **AnimatMonitor**
- class **AreaSensor**

  Detects objects within an area specified by the size and shape of the **AreaSensor**.

- class **auto_indexed_pointer_property**
- class **auto_indexed_property**
- class **auto_pointer_property**
- class **auto_property**
- class **Bacterium**
- class **BeamSensor**

  BeamSensors can really be three distinct kinds of sensor: Lasers, which just detect objects a certain distance away in a straight line from the sensor's origin.

- struct **bound_mem_fun_t**

  A functor which creates a unary function from a unary member function, binding an instance of the class to which the function belongs.

- class **call_on_mem_t**

Allows us to bind functors so that they work on particular members of classes, useful for using for_each on maps.

- class **Collisions**
- struct **creator**

  A functor for use with the for_each algorithm which can perform creation of objects when called on a container of pointers.

- struct **deleter**

  A functor for use with the for_each algorithm which can perform deletion of objects when called on a container of pointers.

- class **Distribution**

  Implements a grid which stores spatial density information to a specified resolution, e.g.

- struct **Distribution::Kernel**

  Implements diffusion and other neighbourhood operations.

- class **DNNAnimat**

  An **Animat** *with a built-in dynamical network which is automatically configured depending on the Animat's sensor and control configuration.*

- class **Drawable**
- class **DynamicalNet**

  This class implements a fully recurrent continuous (or dynamical) neural network.

- struct **DynamicalNet::Neuron**

  Unlike the **FeedForwardNet***, the* **Neuron** *in* **DynamicalNet** *is more worthy of its name, since nearly all the processing of the DNN's firing algorithm occurs here.*

- class **EvalCount**
- struct **EvalDensity**
- struct **EvalGradient**
- class **EvalNearest**

  Keeps a tally of the nearest point passed in and returns it with GetOutput.

- class **EvalNearestAbsX**

  Returns the absolute x position of the nearest target.

- class **EvalNearestAbsY**

  Returns the absolute y position of the nearest target.

- class **EvalNearestAngle**

  Returns the normalised angle to the nearest target.

- class **EvalNearestSignal**

  Sensor *evaluation functor: returns the signal of the nearest individual.*

- class **EvalNearestXDist**

  Returns the vertical distance to the nearest target.

- class **EvalNearestYDist**

  Returns the horizontal distance to the nearest target.

- class **EvoDNNAnimat**

  An evolvable version of **DNNAnimat** *with GetGenotype/SetGenotype methods already set up.*

- class **EvoFFNAnimat**

  An evolvable version of **FFNAnimat** *with GetGenotype/SetGenotype methods already set up.*

- class **Evolver**
- struct **extractor**

  This is a function object which can be used for copying from an iterator when the number of input values is unknown.

- class **FeedForwardNet**

  This is an implementation of a simple two-layer feed-forward neural network.

- struct **FeedForwardNet::Neuron**

  This member struct simply encapsulates the weighted sum function which has to be performed on the weights of each node when the net fires.

- class **FFNAnimat**

  An **Animat** *with a built-in feed-forward network which is automatically configured depending on the Animat's sensor and control configuration.*

- struct **Gaussian2D**

  Plots a two-dimensional Gaussian function in a distribution or distribution kernel.

- struct **GaussianNoise**

  Plots normally distributed noise in a distribution.

- struct **GaussianRing2D**

  Plots a two dimensional Gaussian ring.

- struct **GAVariant**

  This is a general purpose data type which takes five basic data types: int, float, double, char and bool.

- union **GAVariant::VariantData**

  Union of five data types for **GAVariant**.

- class **GeneticAlgorithm**

  The **GeneticAlgorithm** *class provides functionality to cover a range of GA methods, and may be extended to incorporate other approaches.*

- struct **GeneticAlgorithm::evo_sort**

  A little function object to enable us to sort the population by fitness.

- struct **GetSimulation**
- struct **GetSimulationBase**
- class **Group**

A simple class which creates and maintains a vector of objects of the specified type and adds them to the world each round.

- struct **LimitDistribution**
- struct **MatchAdapter**

  Allows any unary predicate to be adapted for use as a matching function.

- struct **MatchComposeAnd**

  Chains any number of matching functions together such that only if all of them are true for the object being matched, **MatchComposeAnd** *will return true.*

- struct **MatchComposeOr**

  Chains any number of matching functions together such that should any of them be true for the object being matched, **MatchComposeOr** *will return true.*

- struct **MatchExact**

  Identifies exact object types, so if defined with Cheese, will return true only for Cheese, and false for Cheddar and Gruyère.

- struct **MatchKindOf**

  Identifies objects belonging to hierarchies, so if defined with Cheese, will return true for objects of type Cheese, or derived classes such as Cheddar and Gruyère.

- struct **MatchSpecific**

  Identifies one particular object and returns true only for that object.

- struct **MutationOperator**
- struct **MutationOperator< bool >**

  Specialised **MutationOperator** *for bool, simply NOT's its input.*

- struct **MutationOperator< GAVariant >**

  This specialised mutation operator provides the facilities of the basic **MutationOperator** *for* **GAVariant**.

- struct **NormalMutator**
- struct **ObjLoader**

  A functor for recreating templated object types using serialisation.

- struct **ObjLoaderBase**

  A simple abstract base class for **ObjLoader** *functors.*

- class **pointer_property**
- class **Population**

  This class is derived from **Group** *and adds a managed GA which is automatically run on the whole* **Population** *every epoch.*

- struct **Population::Clone**

  Used by **Population** *to create a clone of an individual.*

- struct **Population::UnClone**

Merges two individuals such that the resulting individual's fitness scores will contain all the scores of both individuals.

- class **property**

  Class wrapper for a member variable which allows member data to be exposed with invisible get/set semantics.

- class **PSOAlgorithm**
- struct **Ring2D**

  Plots a two dimensional ring.

- struct **ScaleAbs**

  Returns the absolute value of the input, as for the std::abs function.

- struct **ScaleAdapter**

  Allows any unary functor to be adapted for use as a scaling function.

- struct **ScaleCompose**

  ScaleCompose *allows the chaining of two scaling functions together, such the output of a* **ScaleCompose** *functor is the result of second(first(input)), where first and second are the arguments in ScaleCompose's constructor.*

- struct **ScaleGradient**
- struct **ScaleLinear**

  A simple linear scaling function which defaults to an input scale between 0 and a defined maximum, scaling to an output range between 0 and 1.

- struct **ScaleNoise**

  ScaleNoise *adds uniform random noise to its input.*

- struct **ScaleThreshold**

  ScaleThreshold *takes values: threshold, min and max and returns min if input < threshold, or max if input >= threshold.*

- class **SelfSensor**

  The **SelfSensor** *is used to detect information about its owner.*

- class **Sensor**

  The **Sensor** *class is the base class for all the different types of sensor:* **TouchSensor**, **SelfSensor**, **AreaSensor** *and* **BeamSensor**.

- struct **SensorEvalFunction**

  Abstract base class for evaluation functors.

- struct **SensorMatchFunction**

  Abstract base class for matching functors.

- struct **SensorScaleFunction**

  Abstract base class for scaling functors.

- struct **SerialException**

Since exceptions have an undesirable overhead, they have not been used elsewhere in the simulation environment for reasons of speed.

- class **Signaller**

  A general-purpose class for modelling signallers with discrete signal and state types.

- class **SimObject**

  An abstract base class for the **Population** *template, allowing populations with different templated types to be represented in* **Simulation**.

- class **Simulation**

  The basic **Simulation** *framework which must be derived to set up simulations.*

- struct **switcher**

  The switcher is useful when configuring bools from string data.

- class **TouchSensor**

  Detects objects which are touching the sensor's owner.

- class **Trail**
- struct **UniformNoise**

  Plots uniform noise in a distribution.

- class **Unserialiser**

  This class is available for unserialising objects from streams, without knowing which type of object is to be unserialised - the type is determined from the header of the stream.

- class **Vector2D**

  A class for representing two-dimensional vectors and coordinates.

- class **Wall**

  This is a handy class for putting the most common type of obstacle - walls - into the world.

- class **World**

  This is where it all happens: **World** *contains pointers to every object in the simulation environment and allows those objects to interact with each other, and then be displayed.*

- struct **World::DisplayInfo**
- struct **World::PointerInfo**
- class **WorldObject**

  The base class for everything that makes a difference in the world, including Animats, Sensors and all types of scenery and interactive object.

- struct **ZeroDistribution**

Typedefs

- typedef float **DistReal**

  For speed, Distributions use floats but this typedef makes it possible to switch to doubles if higher accuracy is required.

Enumerations

- enum **AnimatPartType** { **ANIMAT_BODY**, **ANIMAT_CENTRE**, **ANIMAT_ARROW**, **ANIMAT_WHEEL** }

  Enumeration type for the different coloured parts of the Animat.

- enum **ColourType** {

  COLOUR_BLACK, **COLOUR_WHITE**, **COLOUR_GREEN**, **COLOUR_BLUE**,

  COLOUR_RED, **COLOUR_PURPLE**, **COLOUR_DARK_PURPLE**, **COLOUR_YELLOW**,

  COLOUR_LILAC, **COLOUR_BROWN**, **COLOUR_LIGHT_GREY**, **COLOUR_DARK_GREY**,

  COLOUR_MID_GREY, **COLOUR_ORANGE**, **COLOUR_PINK**, **COLOUR_SELECTION** }

  An enumeration type for colours.

- enum **GASelectionType** { **GA_ROULETTE** = 0, **GA_RANK**, **GA_TOURNAMENT** }

  The different options for selection are enumerated here.

- enum **GAFltParamType** { **GA_TOURNAMENT_PARAM**, **GA_RANK_SPRESSURE**, **GA_EXPONENT** }

  Assorted float parameters, set using GeneticAlgorithm::SetParameter.

- enum **GAIntParamType** { **GA_TOURNAMENT_SIZE** }

  Assorted integer parameters, set using GeneticAlgorithm::SetParameter.

- enum **GAPrintStyleType** { **GA_PARAMETERS** = 1, **GA_CURRENT** = 2, **GA_GENERATION** = 4, **GA_HISTORY** = 8 }

  Use to set the printing style when using GA's << operator.

- enum **GAFitnessMethodType** { **GA_BEST_FITNESS**, **GA_WORST_FITNESS**, **GA_MEAN_FITNESS**, **GA_TOTAL_FITNESS** }

  The method by which fitness is decided when individuals have multiple scores.

- enum **GAFitnessFixType** { **GA_IGNORE**, **GA_CLAMP**, **GA_FIX** }

  Sets the method by which fitness scores are adjusted before selection.

- enum **GAVariantType** {

  GAV_INT, **GAV_FLOAT**, **GAV_DOUBLE**, **GAV_CHAR**,

  GAV_BOOL }

  A type flag for the GAVariant data type.

- enum **SelfSensorType** { **SELF_SENSOR_X**, **SELF_SENSOR_Y**, **SELF_SENSOR_ANGLE**, **SELF_SENSOR_CONTROL** }

  An enumeration type for SelfSensor, used to specify which feature of the sensor's owner is to be returned by GetOutput().

- enum **SerialErrorType** {

  SERIAL_ERROR_UNKNOWN, **SERIAL_ERROR_BAD_FILE**, **SERIAL_ERROR_WRONG_TYPE**, **SERIAL_ERROR_UNKNOWN_TYPE**,

  SERIAL_ERROR_DATA_MISMATCH }

  Enumerates the different types of errors encountered in serialisation.

- enum **SimPrintStyleType** {

  SIM_PRINT_STATUS, **SIM_PRINT_ASSESSMENT**, **SIM_PRINT_GENERATION**, **SIM_PRINT_RUN**,

  SIM_PRINT_COMPLETE }

  Used in Simulation::ToString to specify what is to be output.

- enum **WorldDisplayType** {

  DISPLAY_NONE = 0, **DISPLAY_ANIMATS** = 1, **DISPLAY_WORLDOBJECTS** = 2, **DISPLAY_TRAILS** = 4,

  DISPLAY_SENSORS = 8, **DISPLAY_COLLISIONS** = 16, **DISPLAY_MONITOR** = 32, **DISPLAY_ALL** = 65535 }

  An enumeration type for specifying which elements of the world are to be displayed.

Functions

- **Sensor** * **GradientSensor** ()
- **Sensor** * **DistributionSensor** ()
- ostream & **operator**<< (ostream &out, const Drawable &d)
- istream & **operator**>> (istream &in, Drawable &d)
- ostream & **operator**<< (ostream &out, const **DynamicalNet** &dnn)

  An output operator for **DynamicalNet**.

- istream & **operator**>> (istream &in, **DynamicalNet** &dnn)

  An input operator for **DynamicalNet**.

- ostream & **operator**<< (ostream &out, const **FeedForwardNet** &ffn)

  Output operator overload for the **FeedForwardNet**.

- istream & **operator**>> (istream &in, **FeedForwardNet** &ffn)

  Input operator overload for the **FeedForwardNet**.

- string **add_slashes** (const string &str)

  Replaces spaces with underscores and adds backslashes to other characters which might be interpreted as white space by an input stream.

- string **strip_slashes** (const string &str)

  Removes the slashes added by add_slashes and reinstates the original string.

- ostream & **operator**<< (ostream &out, const **SimObject** &obj)

  An output operator for all classes derived from **SimObject**.

- istream & **operator**>> (istream &in, **SimObject** &obj)

An input operator for all classes derived from **SimObject**.

- const float * **random_colour** ()

  Returns a random colour, all set for input to glColour4fv.

- template<typename T> std::ostream & **operator**<< (std::ostream &out, const **MutationOperator**< T > &m)

  Output operator for the **MutationOperator** *function object.*

- template<typename T> std::istream & **operator**>> (std::istream &in, **MutationOperator**< T > &m)

  Input operator for the **MutationOperator** *function object.*

- template<class EVO, class MUTFUNC> std::ostream & **operator**<< (std::ostream &, const **GeneticAlgorithm**< EVO, MUTFUNC > &)

  The GA's output operator.

- template<class EVO, class MUTFUNC> std::istream & **operator**>> (std::istream &, const **GeneticAlgorithm**< EVO, MUTFUNC > &)

  The GA's input operator.

- template<class EVO, class MUTFUNC> std::istream & **operator**>> (std::istream &in, **GeneticAlgorithm**< EVO, MUTFUNC > &ga)
- std::ostream & **operator**<< (std::ostream &out, const **GAVariant** &v)
- std::istream & **operator**>> (std::istream &in, **GAVariant** &v)
- void **glut_start_simulation** (int &args, char *argv[], **Simulation** *pTheSim)
- template<class T> **Sensor** * **ProximitySensor** (double scope, double range, double orientation)

  Creates a segment-shaped sensor with the specified scope, range and orientation which detects the distance of objects of the specified templated type.

- template<class T> **Sensor** * **NearestAngleSensor** ()
- template<class T> **Sensor** * **NearestXSensor** ()
- template<class T> **Sensor** * **NearestYSensor** ()
- template<class T> **Sensor** * **DensitySensor** (double scope, double range, double orientation)
- template<class T> **Sensor** * **CollisionSensor** ()
- template<class _Functor> **MatchAdapter**< _Functor > * **MatchAdapt** (_Functor f)

  A helper function for creating **MatchAdapter** *functors.*

- std::string **add_slashes** (const std::string &str)
- std::string **strip_slashes** (const std::string &str)
- template<class _Iterator> **extractor**< _Iterator > **extract** (_Iterator &Iter)

  Constructs and returns an extractor of the correct type.

- template<class _InIt, class _OutIt> _InIt **copy_from** (_OutIt _First, _OutIt _Last, _InIt _Src)

  An algorithm similar to copy, but where the start and end of the target range is specified rather than the source range.

- template<class _OutIt> std::istream & **copy_from_istream** (_OutIt _First, _OutIt _Last, std::istream &in)

  Although extract and copy_from should, with the help of istream_iterator, be able to fill ranges from input streams, certain difficulties with istreams arise which make the copy_from_istream function useful.

- std::istream & **operator>>** (std::istream &in, **switcher** s)

  Input operator for the switcher helper object.

- template<typename _Ty, typename _Ax> std::ostream & **operator<<** (std::ostream &out, const std::vector< _Ty, _Ax > &v)

  A generic output operator for vectors, requires the vector's contents type to have its own << operator.

- template<typename _Ty, typename _Ax> std::istream & **operator>>** (std::istream &in, std::vector< _Ty, _Ax > &v)

  A generic input operator for vectors, requires the vector's contents type to have its own >> operator.

- template<typename _Ty, typename _Pr, typename _Alloc> std::ostream & **operator<<** (std::ostream &out, const std::map< std::string, _Ty, _Pr, _Alloc > &m)

  A specialised output operator for maps with key type string, which uses add_slashes to encode the string.

- template<typename _Ty, typename _Pr, typename _Alloc> std::istream & **operator>>** (std::istream &in, const std::map< std::string, _Ty, _Pr, _Alloc > &m)

  A specialised input operator for maps with key type string, which uses strip_slashes to decode the string.

- template<typename _Kty, typename _Ty, typename _Pr, typename _Alloc> std::ostream & **operator<<** (std::ostream &out, const std::map< _Kty, _Ty, _Pr, _Alloc > &m)

  A generic output operator for maps, requires the map's contents to have its own << operator.

- template<typename _Kty, typename _Ty, typename _Pr, typename _Alloc> std::istream & **operator>>** (std::istream &in, std::map< _Kty, _Ty, _Pr, _Alloc > &m)

  A generic input operator for maps, requires the map's contents to have its own >> operator.

- template<typename T1, typename T2> T1 & **stream_convert** (T2 &input)

  An inline reinterpret cast which may be helpful in outputting enumeration types to streams.

- template<class _InIt, class _Ty, class _Fn1> _Ty **accumulate_fun** (_InIt _First, _InIt _Last, _Ty _Val, _Fn1 _Func)

  Version of the STL accumulate algorithm which computes a sum of all the results of a unary function _Func applied to the values between _First and _Last.

- template<class T, typename M, class OP> **call_on_mem_t**< T, M, OP > **call_on_mem** (M T::*m, OP op)

  Helper function for constructing **call_on_mem_t** *function objects.*

- template<class _Class, typename _Return, typename _Arg> **bound_mem_fun_t**< _Class, _Return, _Arg > **bound_mem_fun** (_Class &c, _Return(_Class::*memfun)(_Arg))

  A helper function for constructing **bound_mem_fun_t** *objects.*

- template<class _Class, typename _Return, typename _Arg> **bound_mem_fun_t**< _Class, _Return, _Arg > **bound_mem_fun** (_Class *c, _Return(_Class::*memfun)(_Arg))

  A helper function for constructing **bound_mem_fun_t** *objects.*

- template<typename _Type, typename _Base> bool **IsA** (_Base *in, _Type *&out)

  A wrapper for RTTI (RunTime Type Identification) typeid, which checks if two pointers are of identical types.

- template<typename _Type, typename _Base> bool **IsKindOf** (_Base *in, _Type *&out)

  A wrapper for RTTI (RunTime Type Identification) using dynamic_cast which checks if an object is of the same type or is in herited from an object of the same type as an input pointer.

- template<typename T> T **bound** (T L, T U, T n)

  Takes a type, a lower and an upper limit and bounds the input value to those limits.

- template<typename T> void **rbound** (T L, T U, T &n)

  A version of bound which takes a reference as its argument.

- template<typename T> T **limit** (T L, T U, T n)

  Limits the input value to the specified range, clipping at either extreme.

- template<typename T> void **rlimit** (T L, T U, T &n)

  A version of limit which takes a reference as its argument.

- double **deg2rad** (double)

  Converts degrees to radians.

- double **rad2deg** (double)

  Converts radians to degrees.

- **Vector2D operator** * (double l, const **Vector2D** &v)

  Returns a vector multiplied by a double.

- **Vector2D PolarVector** (double l, double a)

  Creates a **Vector2D** *object using old PolarVector syntax.*

- bool **LoadPlugin** (const char *plugin, std::vector< std::string > &names, std::vector< GetSimulationBase * > &funcs)
- bool **LoadPlugin** (const char *plugin, std::map< std::string, GetSimulationBase * > &output)
- bool **UnloadPlugin** (const char *plugin)
- int **UnloadPlugins** ()
- BEAST_DLL void **SetupSimulationTable** (std::vector< std::string > &names, std::vector< GetSimulationBase * > &funcs)
- bool **ScreenGrab** (**World** &theWorld, std::string filename)
- template<class T> **Sensor** * **NearestSignalSensor** (int highestSignal)

  Constructs and returns a pointer to a sensor which will return the signal of the nearest **Signaller** *of the specified type.*

Variables

- const double **ANIMAT_RADIUS** = 5.0

  Animat's default radius.

- const double **ANIMAT_MAX_SPEED** = 100.0

  Animat's default maximum speed.

- const double **ANIMAT_MIN_SPEED** = -50.0

  Animat's default minimum speed.

- const double **ANIMAT_MAX_ROTATE** = TWOPI

  The default max rotation/frame.

- const double **ANIMAT_DRAG** = 50.0

  An arbitrary friction value.

- const double **ANIMAT_ACCEL** = 5000.0

  AN arbitrary acceleration value.

- const double **ANIMAT_TIMESTEP** = 0.05

  The default time step.

- const int **ANIMAT_PARTS** = 4

  The number of different colours.

- const int **MONITOR_BARHEIGHT** = 25
- const unsigned int **MAX_COLLISIONS** = 200
- const float **ColourPalette** [][4]

  A global colour pallete. Could probably do with many more colours.

- const double **DRAWABLE_RADIUS** = 50.0

  The default diameter for drawables.

- const double **FFN_ACTIVATION_RESPONSE** = 1

  This value decides the curve of the sigmoid function.

- const int **FFN_COLSIZE** = 6

  The width of columns in ToString output.

- const float **SENSOR_ALPHA** = 0.1f

  Transparency value for Sensors.

- const double **BEAM_SENSOR_SCOPE** = PI/4

  The default scope for **BeamSensor**.

- const double **BEAM_SENSOR_RANGE** = 250.0

  The default range for **BeamSensor**.

- const float **BEAM_DRAW_QUALITY** = 0.1f

The default draw quality for **BeamSensor**.

- const unsigned int **TRAIL_LENGTH** = 30
- const double **WORLD_WIDTH** = 800.0
- const double **WORLD_HEIGHT** = 600.0

9.1.1 Detailed Description

The namespace for everything in the simulation environment.

9.1.2 Function Documentation

9.1.2.1 string add_slashes (const string & *str*)

Replaces spaces with underscores and adds backslashes to other characters which might be interpreted as white space by an input stream.

See also:
strip_slashes

9.1.2.2 ostream& operator<< (ostream & *out*, const SimObject & *obj*) [related]

An output operator for all classes derived from **SimObject**.

Parameters:
out An output stream.
obj An object derived from **SimObject**.

Returns:
A reference to output stream.

9.1.2.3 istream& operator>> (istream & *in*, SimObject & *obj*) [related]

An input operator for all classes derived from **SimObject**.

Parameters:
in An input stream.
obj An object derived from **SimObject**.

Returns:
A reference to input stream.

9.1.2.4 string strip_slashes (const string & *str*)

Removes the slashes added by add_slashes and reinstates the original string.

See also:
add_slashes

Chapter 10

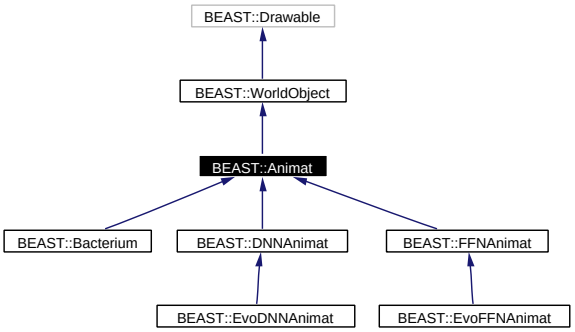
BEAST - Bioinspired Evolutionary Agent Simulation Toolkit Class Documentation

10.1 BEAST::Animat Class Reference

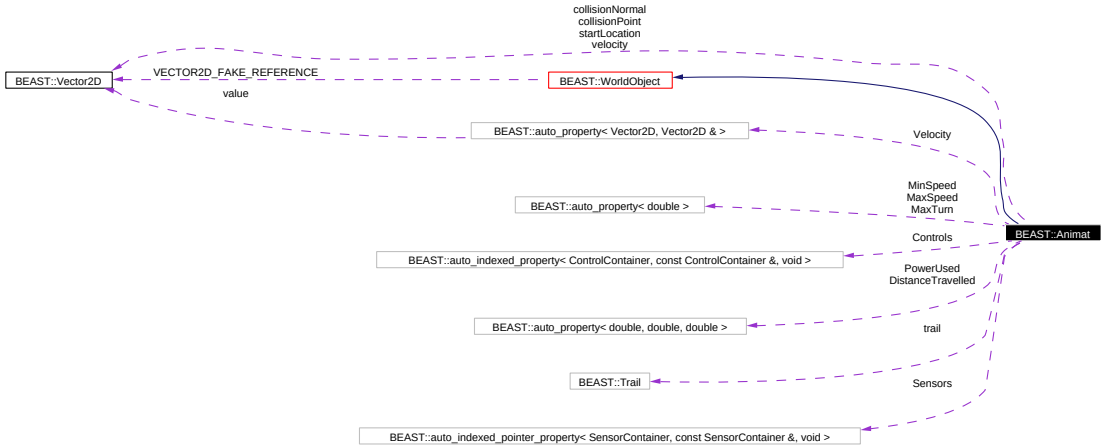
Animats can move around and interact with other objects in the world.

`#include <animat.h>`

Inheritance diagram for BEAST::Animat:



Collaboration diagram for BEAST::Animat:



Public Member Functions

- **Animat** ()

  Constructor for **Animat**, *sets most values to 0 or default.*

- virtual ~**Animat** ()

  Destructor: decrements Animat::numAnimats and deletes all the sensors which are owned by this **Animat**.

- virtual void **Init** ()

  Sets the trail colour to match the animat and calls **WorldObject::Init()**.

- void **Add** (std::string name, **Sensor** *s)

  Adds named sensors to the Animat's sensor container, and sets the owner to this **Animat**.

- void **Share** (std::string name, **Sensor** *s)

  Allows an **Animat** *to have access to a sensor without taking ownership of it.*

- virtual void **Update** ()

  Updates the animat's position according to velocity and performs updates on members (sensors and trail).

- virtual void **Control** ()

  Override this method to provide your own control method.

- virtual void **Interact** (**Animat** *)

  Processes collisions with other animats, including rudimentary physics (sticky collisions).

- virtual void **Interact** (**WorldObject** *)

  Processes collisions with non-animats, including displacement, sensors and collision events.

- virtual bool **IsTouching** (const **WorldObject** *) const

  Returns true when animat is touching other, and Sets the collision point to a point on the animat.

- virtual void **OnCollision** (**WorldObject** *r)
- void **SensorInteract** (**WorldObject** *other)

  Passes a **WorldObject** *to all the sensors in turn.*

- virtual void **Display** ()

  Displays the **Animat** *along with its sensors and trail, depending on the configuration of myWorld.*

- virtual void **Draw** ()

  Implementing Drawable - draws the **Animat** *in the correct colour.*

- **Vector2D GetVelocity** () const
- double **GetMaxSpeed** () const
- double **GetMinSpeed** () const
- double **GetMaxRotateSpeed** () const
- double **GetTimeStep** () const
- double **GetDistanceTravelled** () const
- double **GetPowerUsed** () const
- void **SetStartLocation** (const **Vector2D** &l)

  Sets the Animat's starting location.

- void **SetStartOrientation** (double o)
- virtual void **Serialise** (std::ostream &) const

  Outputs the Animat's data to a stream.

- virtual void **Unserialise** (std::istream &)

  Inputs the Animat's data from a stream.

- void **SetCollisionPoint** (const **Vector2D** &v) const
- void **SetCollisionNormal** (const **Vector2D** &v) const

Static Public Member Functions

- int **GetNumAnimats** ()

  Returns the global **Animat** *count, Animat::numAnimats.*

- void **SetTimeStep** (double t)

Public Attributes

- auto_property< **Vector2D**, **Vector2D** & > **Velocity**
- auto_property< double > **MinSpeed**
- auto_property< double > **MaxSpeed**
- auto_property< double > **MaxTurn**
- auto_property< double, double, double > **DistanceTravelled**
- auto_property< double, double, double > **PowerUsed**

Protected Types

- typedef std::map< std::string, **Sensor** * > **SensorContainer**
  A typedef for the Animat's sensor container, sensors.
- typedef SensorContainer::iterator **SensorIter**
  A typedef for the SensorContainer's iterator.
- typedef std::map< std::string, float > **ControlContainer**
  A typedef for the Animat's control container, controls.
- typedef ControlContainer::iterator **ControlIter**
  A typedef for the ControlContainer's iterator.

Protected Member Functions

- void **SetVelocity** (const **Vector2D** &pv)
  Sets the Animat's velocity.
- void **SetVelocityX** (double x)
  Sets the X component of the Animat's velocity.
- void **SetVelocityY** (double y)
  Sets the Y component of the Animat's velocity.
- void **AddVelocity** (const **Vector2D** &v)
  Adds a value (possibly negative) to the Animat's velocity.
- void **SetMaxSpeed** (double s)
  Sets the Animat's maximum speed.
- void **SetMinSpeed** (double s)
  Sets the Animat's minimum speed.
- void **SetColour** (**AnimatPartType**, const float *)
  Sets the colour of the specified.
- void **SetColour** (**AnimatPartType**, float, float, float, float a=1.0f)
  Sets the specified part of the **Animat** *to the specified colour.*
- const **SensorContainer** & **GetSensors** () const
- **SensorContainer** & **GetSensors** ()
- const **ControlContainer** & **GetControls** () const
- **ControlContainer** & **GetControls** ()

Protected Attributes

- auto_indexed_pointer_property< **SensorContainer**, const **SensorContainer** &, void > **Sensors**
- auto_indexed_property< **ControlContainer**, const **ControlContainer** &, void > **Controls**

Friends

- class **AnimatMonitor**

  To give AnimatMonitor access to sensor output.

- class **SelfSensor**

  To give **SelfSensor** *quick access to all info.*

10.1.1 Detailed Description

Animats can move around and interact with other objects in the world.

Unlike a plain **WorldObject**, Animats have a Control method which allows them to be customised with new control systems. The **Animat** class will rarely be of any use on its own, rather it should be used as a base class. Examples of classes derived from **Animat** include **FFNAnimat** and **DNNAnimat**.

See also:
Animat::Control
WorldObject
FFNAnimat
DNNAnimat

10.1.2 Constructor & Destructor Documentation

10.1.2.1 BEAST::Animat::Animat ()

Constructor for **Animat**, sets most values to 0 or default.

Sets up controls for motors, "left" and "right", stores the start location and increments the global **Animat** counter, Animat::numAnimats. Finally, gives the **Animat** a random colour.

10.1.3 Member Function Documentation

10.1.3.1 void BEAST::Animat::Add (std::string *name*, Sensor * *s*)

Adds named sensors to the Animat's sensor container, and sets the owner to this **Animat**.

Parameters:
name The name of the sensor (unique to animat).

s A pointer to the sensor.

10.1.3.2 virtual void BEAST::Animat::Control () `[inline, virtual]`

Override this method to provide your own control method.

Reimplemented in **BEAST::FFNAnimat** (p. 134), and **BEAST::DNNAnimat** (p. 100).

10.1.3.3 void BEAST::Animat::Display () [virtual]

Displays the **Animat** along with its sensors and trail, depending on the configuration of myWorld.

See also:
World::SetDispConfig
World::GetDispConfig
World::WorldDisplayType

10.1.3.4 void BEAST::Animat::Interact (WorldObject * *other*) [virtual]

Processes collisions with non-animats, including displacement, sensors and collision events.

Parameters:
other A pointer to the **WorldObject** we're interacting with.

Reimplemented from **BEAST::WorldObject** (p. 215).

Reimplemented in **BEAST::Bacterium** (p. 75).

10.1.3.5 void BEAST::Animat::Interact (Animat * *other*) [virtual]

Processes collisions with other animats, including rudimentary physics (sticky collisions).

Also calls onCollide event and sensorInteract on both animats.

Parameters:
other A pointer to the **Animat** we're interacting with.

10.1.3.6 bool BEAST::Animat::IsTouching (const WorldObject * *other*) const [virtual]

Returns true when animat is touching other, and Sets the collision point to a point on the animat.

Also Sets the colision normal via GetNearestPoint.

Parameters:
other A pointer to the **WorldObject** we're checking.

10.1.3.7 void BEAST::Animat::SensorInteract (WorldObject * *other*)

Passes a **WorldObject** to all the sensors in turn.

Parameters:
other A pointer to the **WorldObject** being sensed.

10.1.3.8 void BEAST::Animat::Serialise (std::ostream & *out*) const [virtual]

Outputs the Animat's data to a stream.

Parameters:
out Reference to an output stream.

Reimplemented from **BEAST::WorldObject** (p. 216).

Reimplemented in **BEAST::FFNAnimat** (p. 135), and **BEAST::DNNAnimat** (p. 101).

10.1.3.9 void BEAST::Animat::Share (std::string *name*, Sensor * *s*) [inline]

Allows an **Animat** to have access to a sensor without taking ownership of it.

Parameters:
name The name of the sensor (unique to this **Animat**).

s A pointer to the sensor.

See also:
Animat::Add

10.1.3.10 void BEAST::Animat::Unserialise (std::istream & *in*) [virtual]

Inputs the Animat's data from a stream.

Parameters:
in Reference to an input stream.

Reimplemented from **BEAST::WorldObject** (p. 216).

Reimplemented in **BEAST::FFNAnimat** (p. 135), and **BEAST::DNNAnimat** (p. 101).

10.1.3.11 void BEAST::Animat::Update () [virtual]

Updates the animat's position according to velocity and performs updates on members (sensors and trail).

Note that the recommended way of influencing the behaviour of **Animat** subclasses is to overload the Control method of **Animat**, rather than Update (since Control is called at an opportune moment from within Update).

Reimplemented from **BEAST::WorldObject** (p. 213).

Reimplemented in **BEAST::Bacterium** (p. 84).

The documentation for this class was generated from the following files:

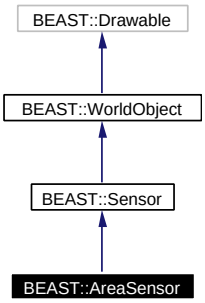
- **animat.h**
- **animat.cc**

10.2 BEAST::AreaSensor Class Reference

Detects objects within an area specified by the size and shape of the **AreaSensor**.

`#include <sensorbase.h>`

Inheritance diagram for BEAST::AreaSensor:



Collaboration diagram for BEAST::AreaSensor:

Public Member Functions

- **AreaSensor** (**Vector2D** l, double o)
- virtual void **Interact** (**WorldObject** *other)

  Checks if the **WorldObject** *is the correct type using MatchFunc, then checks if the object's centre is inside the* **AreaSensor** *and calls the EvalFunc.*

10.2.1 Detailed Description

Detects objects within an area specified by the size and shape of the **AreaSensor**.

Currenly only detects objects when their location is within the area (i.e. just touching the area won't trip the sensor).

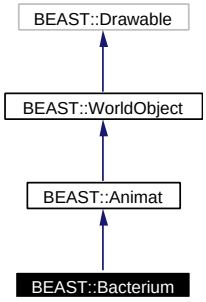
The documentation for this class was generated from the following files:

- **sensorbase.h**
- sensor.cc

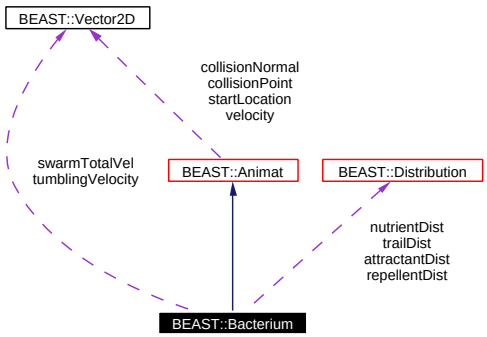
10.3 BEAST::Bacterium Class Reference

`#include <bacterium.h>`

Inheritance diagram for BEAST::Bacterium:



Collaboration diagram for BEAST::Bacterium:



Public Member Functions

- **Bacterium** ()

  Sets up the **Bacterium** *with a series of standard values which should be overridden by the constructors of inherited classes.*

- virtual ~**Bacterium** ()

  Deletes all offspring of this **Bacterium**.

- virtual void **Update** ()

  Updates the location and energy levels of the **Bacterium**, *and causes various changes in the nutrient, attractant and repellent distributions.*

- virtual void **Draw** ()

  Override the **Animat** *drawing function and return to a simple blob shape.*

- virtual void **Interact** (**WorldObject** *)

  Overloaded to stop bacteria from bothering to check for collisions with distribution objects.

- virtual void **UniInteract** (**WorldObject** *)

  Overloaded to ensure that bacteria keep track of the velocities of individuals in their local neighbourhood (determined by swarmRadius).

- void **CheckBoundary** ()

  Before we do any checks, ensure that bacteria are all within boundaries so we don't end up trying to read off the edges of the distribution.

- void **ReadDistributions** ()

  Get local information from whichever distributions are in use.

- void **UpdateDistributions** ()

  Consumes nutrient and releases attractant and repellent according to the relevant variables.

- void **ReleaseAttractant** ()

  Releases a quantity of attractant depending on the attractantRate, attractantThreshold and available energy.

- void **ReleaseRepellent** ()

  Releases a quantity of repellent depending on the repellentRate, repellentThreshold and available energy.

- **Vector2D GetSwarmVelocity** ()

  Calculates the average velocity of the individuals within swarmRadius of the current **Bacterium**.

- **Vector2D GetTumblingVelocity** ()

  Returns the current velocity when the tumbling movement system is used.

- **Vector2D GetNutrientGradient** ()

  Returns the local gradient of the nutrient distribution.

- **Vector2D GetAttractantGradient** ()

  Returns the local gradient of the attractant distribution.

- **Vector2D GetRepellentGradient** ()

  Returns the local gradient of the repellent distribution.

- void **UpdateEnergy** ()

  Accounts for the usual energy depletion and decides whether or not to sporulate or unsporulate depending on the.

- void **FinishUpdate** ()

  This should be called at the end of the Update method to update the location based on the calculated velocity.

- void **SetNextCheck** ()

  Used with the Tumbling system of movement, this method calculates the time til the next sampling of the environment.

- void **Reproduce** ()

This will cause a duplicate of the current **Bacterium** *to be added to the* **World**.

- void **GetOffspring** (std::list< **Bacterium** * > &) const

  Recursively compiles a list of all the offspring of this **Bacterium** *and its offspring - a whole family tree.*

- std::list< **Bacterium** * > **GetOffspring** () const

  Recursively compiles a list of all the offspring of this **Bacterium** *and its offspring - a whole family tree.*

- void **GetOffspring** (std::list< **Bacterium** const * > &) const

  Recursively compiles a list of all the offspring of this **Bacterium** *and its offspring - a whole family tree.*

- void **Reset** ()
- virtual std::string **ToString** () const

  Outputs essential information about the **Bacterium**, *N - current nutrient concentration, A - current attractant concentration, R - current repellent concentration, E - current energy, T - total food/energy consumed.*

- virtual void **OnClick** ()

  Outputs detailed information to the log stream (either the log window in the GUI, or standard out in batch mode).

- void **SetReproductionCost** (double r)
- void **SetEnergyRate** (double e)
- void **SetSporeEnergyRate** (double s)
- void **SetAttractantCost** (double a)
- void **SetRepellentCost** (double r)
- void **SetDeathThreshold** (double d)
- void **SetTumbleTime** (double t)
- void **SetTumbleScale** (double t)
- void **SetReproductionThreshold** (double r)
- void **SetSporulationThreshold** (double s)
- void **SetConsumptionRate** (double c)
- void **SetAttractantRate** (double a)
- void **SetRepellentRate** (double r)
- void **SetSwarmRadius** (double s)
- void **SetSwarmInfluence** (double s)
- void **SetGradientInfluence** (double g)
- void **SetNutrientResponse** (double n)
- void **SetAttractantResponse** (double a)
- void **SetRepellentResponse** (double r)
- void **SetAttractantThreshold** (double a)
- void **SetRepellentThreshold** (double r)
- void **SetSpeed** (double s)
- void **SetEnergy** (double e)
- void **SetTotalEnergy** (double e)
- void **SetNutrientDist** (**Distribution** *dist)
- void **SetAttractantDist** (**Distribution** *dist)
- void **SetRepellentDist** (**Distribution** *dist)

- void **SetTrailDist** (**Distribution** *dist)
- double **GetReproductionCost** () const
- double **GetEnergyRate** () const
- double **GetSporeEnergyRate** () const
- double **GetAttractantCost** () const
- double **GetRepellentCost** () const
- double **GetDeathThreshold** () const
- double **GetTumbleTime** () const
- double **GetTumbleScale** () const
- double **GetReproductionThreshold** () const
- double **GetSporulationThreshold** () const
- double **GetConsumptionRate** () const
- double **GetAttractantRate** () const
- double **GetRepellentRate** () const
- double **GetSwarmRadius** () const
- double **GetSwarmInfluence** () const
- double **GetGradientInfluence** () const
- double **GetNutrientResponse** () const
- double **GetAttractantResponse** () const
- double **GetRepellentResponse** () const
- double **GetAttractantThreshold** () const
- double **GetRepellentThreshold** () const
- double **GetSpeed** () const
- double **GetEnergy** () const
- double **GetTotalEnergy** () const
- bool **IsSpore** () const

Protected Attributes

- double **reproductionCost**

  Cost of splitting into two.

- double **energyRate**

  Rate of energy reduction (subtracted each frame).

- double **sporeEnergyRate**

  Rate of energy reduction when sporulated (subtracted each frame).

- double **attractantCost**

  Cost to release one unit of attractant.

- double **repellentCost**

  Cost to release one unit of repellent.

- double **deathThreshold**

  Lowest energy level before **Bacterium** *dies.*

- double **tumbleTime**

  Default time period between tumbles.

- double **tumbleScale**

  Scaling factor for tumble frequency.

- double **reproductionThreshold**

  Energy level before reproduction occurs.

- double **sporulationThreshold**

  Energy level before sporulation.

- double **consumptionRate**

  Rate of nutrient consumption.

- double **attractantRate**

  Rate of attractant release.

- double **repellentRate**

  Rate of repellent release.

- double **swarmRadius**

  Neighbourhood radius for local swarm.

- double **swarmInfluence**

  Degree of direction influence taken from swarm.

- double **gradientInfluence**

  Degree of direction influence taken from gradients.

- double **nutrientResponse**

  Degree of response to nutrient gradient.

- double **attractantResponse**

  Degree of response to attractant gradient.

- double **repellentResponse**

  Degree of response to repellent gradient.

- double **attractantThreshold**

  Level of nutrient which must be present for attractant to be released.

- double **repellentThreshold**

  Level of nutrient which must be present for repellent to be released.

- **Distribution * nutrientDist**

  Pointer to the nutrient distribution.

- **Distribution * attractantDist**

  Pointer to the attractant distribution.

- **Distribution * repellentDist**

  Pointer to the repellent distribution.

- **Distribution * trailDist**

  Pointer to the trail distribution.

- std::list< **Bacterium** * > **offspring**

  List of pointers to offspring bacteria.

- double **energy**

  Internal energy level.

- double **totalEnergy**

  Global energy total.

- double **lastNutrient**

  Last recorded nutrient level.

- double **lastAttractant**

  Last recorded attractant level.

- double **lastRepellent**

  Last recorded repellent level.

- double **currentNutrient**

  Nutrient level at current location.

- double **currentAttractant**

  Attractant level at current location.

- double **currentRepellent**

  Repellent level at current location.

- int **nextCheck**

  Frames til next nutrient sampling.

- **Vector2D tumblingVelocity**

  Tumbling velocity.

- **Vector2D swarmTotalVel**

  Total neighbourhood velocity.

- int **swarmSize**

  Number of individuals in neighbourhood.

- bool **isSpore**

  True if the bacterium has sporulated.

10.3.1 Detailed Description

A simplified model of a bacteria strain capable of self-organising through chemotaxis.

In fact, this class represents, in simulation, a large number of bacteria with a global average velocity. The **Bacterium** takes nutrients from a nutrient **Distribution** object, and releases chemoattractant and chemorepellent into two other **Distribution** objects. The precise model of the dynamics depends on a range of coefficients, some of which might be optimised using a GA.

At the Update stage, the **Bacterium** checks the level of nutrient, attractant and repellent in its local environment. It adjusts its direction according to the local gradients of these three quantities, their influence determined by [nutrientResponse], [attractantResponse] and [repellentResponse]. This new direction influences the overall velocity according to [gradient-Influence], which ranges between 0 (no influence) and 1 (full influence, original velocity is forgotten).

The velocity is also influenced by [swarmInfluence] (ranging between 0 and 1 again) of the average velocity of bacteria in the local neighbourhood, dictated by [swarmRadius].

The **Bacterium** will absorb a quantity of nutrient from the local environment, depending on its [consumptionRate] and the amount present - it will take all it can each timestep, up to [consumptionRate]. Nutrients are converted directly into energy, so the amount of nutrient consumed is simply added to the [energy] level.

The **Bacterium** may release attractant or repellent, depending on [attractantThreshold], [repellentThreshold] and the amount of nutrient present. If it does so, the amount released is dictated by the product of local nutrient with [attractantRate] or [repellentRate]. The Bacterium's energy is reduced according to [attractantCost] or [repellentCost]. Normal cell processes and the cost of moving around deplete the Bacterium's energy level, this is modelled by simply subtracting [energyRate] each timestep.

Each timestep each **Bacterium** loses sporeEnergyRate energy. If it is moving (i.e. not sporulated), it also loses its velocity $*$ energyRate. When sporulated, it will continue to absorb nutrients but will not move or release repellents or attractants.

If the local nutrient level goes up, the **Bacterium** will start moving again. If the Bacterium's [energy] level goes above the [reproductionThreshold], the **Bacterium** will lose [reproduction-Cost] energy, and then split into two. The new 'offspring' **Bacterium** will be the same colour and have the same constants and coefficients. The energy of the original **Bacterium** is split between the two.

If the Bacterium's energy level drops below [deathThreshold], the **Bacterium** will die, and is then removed from the simulation.

For more detailed information on how bacteria actually move, go to this address:

10.3.2 Member Function Documentation

10.3.2.1 void BEAST::Bacterium::FinishUpdate ()

This should be called at the end of the Update method to update the location based on the calculated velocity.

Also, the swarm process variables are reset.

10.3.2.2 Vector2D BEAST::Bacterium::GetAttractantGradient ()

Returns the local gradient of the attractant distribution.

Returns:
A normalised vector representing the direction of the gradient.

10.3.2.3 Vector2D BEAST::Bacterium::GetNutrientGradient ()

Returns the local gradient of the nutrient distribution.

Returns:
A normalised vector representing the direction of the gradient.

10.3.2.4 void BEAST::Bacterium::GetOffspring (std::list< Bacterium const * > & *babies*) const

Recursively compiles a list of all the offspring of this **Bacterium** and its offspring - a whole family tree.

Because the list to be filled is passed in by reference, this is the fastest version of the method to use.

Parameters:
babies The list to be filled with pointers to const Bacteriums, passed by reference.

10.3.2.5 std::list< Bacterium * > BEAST::Bacterium::GetOffspring () const

Recursively compiles a list of all the offspring of this **Bacterium** and its offspring - a whole family tree.

Because the resulting list is returned, this is a slightly slower version of the method.

Returns:
A list of pointers to Bacteriums.

10.3.2.6 void BEAST::Bacterium::GetOffspring (std::list< Bacterium * > & *babies*) const

Recursively compiles a list of all the offspring of this **Bacterium** and its offspring - a whole family tree.

Because the list to be filled is passed in by reference, this is the fastest version of the method to use.

Parameters:

babies The list to be filled with pointers to Bacteriums, passed by reference.

10.3.2.7 Vector2D BEAST::Bacterium::GetRepellentGradient ()

Returns the local gradient of the repellent distribution.

Returns:

A normalised vector representing the direction of the gradient.

10.3.2.8 void BEAST::Bacterium::ReleaseAttractant ()

Releases a quantity of attractant depending on the attractantRate, attractantThreshold and available energy.

The attractant is added to the attractant distribution.

Warning:

Does not check whether the nutrient is above the attractant threshold, this is done in UpdateDistributions.

10.3.2.9 void BEAST::Bacterium::ReleaseRepellent ()

Releases a quantity of repellent depending on the repellentRate, repellentThreshold and available energy.

The repellent is added to the repellent distribution.

Warning:

Does not check whether the nutrient is below the repellent threshold, this is done in UpdateDistributions.

10.3.2.10 void BEAST::Bacterium::Reproduce ()

This will cause a duplicate of the current **Bacterium** to be added to the **World**.

The Bacterium's energy is reduced by reproductionCost and the remaining energy is divided by two over the two resulting individuals. The offspring's location is immediately behind the parent. A pointer to the new individual is added to the parent's offspring list.

Warning:

This method does not check whether the **Bacterium** has enough energy to reproduce.
This method causes only **Bacterium** objects to be added to the **World**, if you require child classes of **Bacterium** to be produced in reproduction, overload this method.

10.3.2.11 void BEAST::Bacterium::SetNextCheck ()

Used with the Tumbling system of movement, this method calculates the time til the next sampling of the environment.

This depends on the comparative gradients of the nutrient, repellent and attractant distributions and the tumbleTime and tumbleScale variables.

10.3.2.12 string BEAST::Bacterium::ToString () const [virtual]

Outputs essential information about the **Bacterium**, N - current nutrient concentration, A - current attractant concentration, R - current repellent concentration, E - current energy, T - total food/energy consumed.

Returns:
A string containing the specified info.

See also:
WorldObject::OnSelect for an opportunity to pass more information.
WorldObject::GetLogStream for a means of outputting more info.

Reimplemented from **BEAST::WorldObject** (p. 214).

10.3.2.13 void BEAST::Bacterium::Update () [virtual]

Updates the location and energy levels of the **Bacterium**, and causes various changes in the nutrient, attractant and repellent distributions.

For a full explanation of what is going on here, see the **Bacterium** class documentation.

Reimplemented from **BEAST::Animat** (p. 73).

10.3.2.14 void BEAST::Bacterium::UpdateDistributions ()

Consumes nutrient and releases attractant and repellent according to the relevant variables.

Spores do not release attractants or relellants.

The documentation for this class was generated from the following files:

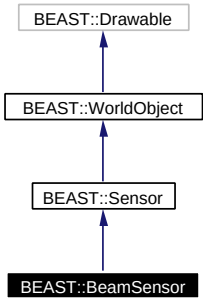
- **bacterium.h**
- bacterium.cc

10.4 BEAST::BeamSensor Class Reference

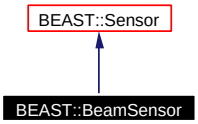
BeamSensors can really be three distinct kinds of sensor: Lasers, which just detect objects a certain distance away in a straight line from the sensor's origin.

`#include <sensorbase.h>`

Inheritance diagram for BEAST::BeamSensor:



Collaboration diagram for BEAST::BeamSensor:



Public Member Functions

- **BeamSensor** (double s=**BEAM_SENSOR_SCOPE**, double r=**BEAM_SENSOR_RANGE**, **Vector2D** l=**Vector2D**(0.0, 0.0), double o=0, **Animat** *owner=NULL)
- virtual void **Update** ()

  Checks if the sensor is wrapping, then sets wrap locations accordingly.

- virtual void **Interact** (**WorldObject** *other)

  A wrapper for _Interact (the real interaction method) for handling wrapping.

- void **_Interact** (**WorldObject** *other)

  Uses a number of collision detection functions to locate the nearest point of the nearest object in scope.

- virtual void **Display** ()

  A wrapper for _Display (the real display method) for handling wrapping.

- void **_Display** ()

  Positions the matrix according to the location of the ownerAnimat and draws the sensor's display list.

- virtual void **Draw** ()

Draws an alpha-blended line, segment or circle depending on the scope of the sensor.

- double **GetScope** () const
- double **GetRange** () const
- void **SetDrawScale** (float d)
- void **SetDrawFixed** (bool f)
- void **SetWrapping** (bool w)

Protected Member Functions

- bool **InScope** (const **Vector2D** &vec)

  Checks to see if a point is within the current testing angle of the sensor.

Protected Attributes

- double **scope**

  width of the beam in radians

- double **range**

  Sets maximum distance.

- float **drawScale**

  Scaling factor used in Display.

- bool **drawFixed**

  Whether to scale display according to output.

10.4.1 Detailed Description

BeamSensors can really be three distinct kinds of sensor: Lasers, which just detect objects a certain distance away in a straight line from the sensor's origin.

- $*$ - Scoped sensors, which detect objects a within a certain range and a specified angle.
- Unidirectional sensors which detect objects a certain distance away at any angle. The three types of sensor are achieved by specifying scopes of 0, [0 < TWOPI] and TWOPI respectively. Note that BeamSensors are the most computationally expensive sensors, so if you can substitute another kind of **Sensor**, do so.

  See also:
  Sensor

10.4.2 Member Function Documentation

10.4.2.1 void BEAST::BeamSensor::_Interact (WorldObject $*$ *other*)

Uses a number of collision detection functions to locate the nearest point of the nearest object in scope.

Automatic optimisation in the case of 360 degree scope.

10.4.2.2 void BEAST::BeamSensor::Draw () [virtual]

Draws an alpha-blended line, segment or circle depending on the scope of the sensor.

The number of points in the circle is determined by the scope and range of the sensor.

The documentation for this class was generated from the following files:

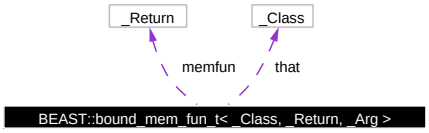
- **sensorbase.h**
- sensor.cc

10.5 BEAST::bound_mem_fun_t< _Class, _Return, _Arg > Struct Template Reference

A functor which creates a unary function from a unary member function, binding an instance of the class to which the function belongs.

`#include <utilities.h>`

Collaboration diagram for BEAST::bound_mem_fun_t< _Class, _Return, _Arg >:



Public Member Functions

- **bound_mem_fun_t** (_Class &t, _Return(_Class::*m)(_Arg))
- _Return **operator()** (_Arg a)

Public Attributes

- _Return(_Class::* **memfun**)(_Arg)
- _Class & **that**

10.5.1 Detailed Description

template<class _Class, typename _Return, typename _Arg> struct BEAST::bound_mem_fun_t< _Class, _Return, _Arg >

A functor which creates a unary function from a unary member function, binding an instance of the class to which the function belongs.

Equivalent to bind1st(mem_fun(class::fun)) which is not valid in standard C++.

The documentation for this struct was generated from the following file:

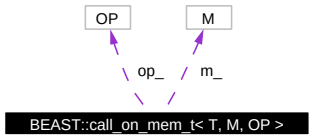
- **utilities.h**

10.6 BEAST::call_on_mem_t< T, M, OP > Class Template Reference

Allows us to bind functors so that they work on particular members of classes, useful for using for_each on maps.

`#include <utilities.h>`

Collaboration diagram for BEAST::call_on_mem_t< T, M, OP >:



Public Types

- typedef M **argument_type**
- typedef OP::result_type **result_type**

Public Member Functions

- **call_on_mem_t** (M T::*m, OP op)
- result_type **operator()** (T &val)

10.6.1 Detailed Description

template<class T, typename M, class OP> class BEAST::call_on_mem_t< T, M, OP >

Allows us to bind functors so that they work on particular members of classes, useful for using for_each on maps.

The documentation for this class was generated from the following file:

- **utilities.h**

10.7 BEAST::creator< T > Struct Template Reference

A functor for use with the for_each algorithm which can perform creation of objects when called on a container of pointers.

`#include <utilities.h>`

Public Member Functions

- T * **operator()** ()

10.7.1 Detailed Description

template<typename T> struct BEAST::creator< T >

A functor for use with the for_each algorithm which can perform creation of objects when called on a container of pointers.

The documentation for this struct was generated from the following file:

- **utilities.h**

10.8 BEAST::deleter< T > Struct Template Reference

A functor for use with the for_each algorithm which can perform deletion of objects when called on a container of pointers.

`#include <utilities.h>`

Public Member Functions

- void **operator()** (T *obj)

10.8.1 Detailed Description

template<typename T> struct BEAST::deleter< T >

A functor for use with the for_each algorithm which can perform deletion of objects when called on a container of pointers.

The documentation for this struct was generated from the following file:

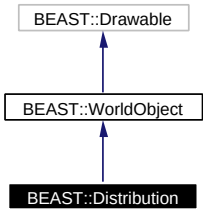
- **utilities.h**

10.9 BEAST::Distribution Class Reference

Implements a grid which stores spatial density information to a specified resolution, e.g.

`#include <distribution.h>`

Inheritance diagram for BEAST::Distribution:



Collaboration diagram for BEAST::Distribution:

Public Member Functions

- **Distribution** (int c, int r, int b=1)

  Sets up the **Distribution** *with specified width, height and kernel radius.*

- virtual ~**Distribution** ()

  Deletes the dynamic arrays and kernel used by the **Distribution**.

- virtual void **Init** ()

  Sets the **Distribution** *up so that it's detectable (with the correct edge vector) and has the same width and height as the world.*

- virtual void **Update** ()

  Filters the distribution every n frames, where n is specified by the diffusion speed.

- **Kernel** & **GetKernel** () const

  Provides access to the kernel so new convolutions can be plotted.

- **DistReal** & **ValueAt** (int x, int y) const

  Provides direct access to distribution data by column and row.

- **DistReal GetDensity** (int x, int y) const

  Returns the density at the specified column and row.

- **DistReal GetDensity** (const **Vector2D** &v) const

Returns the density at the specified point.

- **DistReal GetGradient** (const **Vector2D** &v, double o) const
- **Vector2D GetGradient** (int x, int y) const
- **Vector2D GetGradient** (const **Vector2D** &v) const
- void **SetDensity** (int x, int y, float d)
- void **SetDensity** (const **Vector2D** &v, float d)

  Sets the density to the specified value at the given location.

- void **AddDensity** (int x, int y, float d)
- void **AddDensity** (const **Vector2D** &v, float d)

  Adds the specified value to the distribution at the specified point.

- void **SetDiffusionSpeed** (int s)

  Sets the interval for calling diffusion, e.g.

- void **SetDecayRate** (**DistReal** r)

  Specifies the rate at which the distribution decays: 1.0 for no decay, 0.5 to reduce by half on every diffusion.

- void **SetMaxConc** (float f)

  Specifies the maximum expected concentration for purposes of display.

- virtual void **Render** ()

  Displays the distribution with transparency according to density, up to maxConc density (= opaque).

- void **Plot** (double val)

  Sets every point on the distribution to the specified value.

- template<class _Func> void **Plot** (_Func func)

  Plots the specified function so the value at each coordinate x, y becomes func(x, y).

- template<class _Op, class _Func> void **Filter** (_Op op, _Func func)

  Replaces the value at each coordinate with op(oldval, func(x, y).

- template<class _Op> void **Filter** (_Op op)

  Replaces each value of the distribution with op(oldval).

Protected Attributes

- double **width**

  The real-valued width (same as **World***).*

- double **height**

  The real-valued height (same as **World***).*

- double **colSize**

  The real-valued width of each column.

- double **rowSize**

  The real-valued height of each row.

- int **rows**

  The number of accessible rows.

- int **cols**

  The number of accessible columns.

- int **tRows**

  The total rows including convolution border.

- int **tCols**

  The total columns including convolution border.

- int **border**

  The size of the convolution border (used to make neighbourhood operations work).

- float **maxConc**

  The maximum concentration, for display purposes.

- int **diffusionSpeed**

  How often (in timesteps) the diffusion kernel is used.

- int **nextDiffusion**

  Timesteps left til the next diffusion.

- DistType **distribution**
- DistType **swapbuffer**

  Dynamic arrays containing distribution data.

- **Kernel** * **kernel**

  Used for diffusion.

Friends

- struct **Kernel**

10.9.1 Detailed Description

Implements a grid which stores spatial density information to a specified resolution, e.g.

nutrients in an agar dish. Diffusion is made possible using a fast convolution function. The distribution can be accessed directly or through a number of sensors.

10.9.2 Constructor & Destructor Documentation

10.9.2.1 BEAST::Distribution::Distribution (int *c*, int *r*, int *b* = 1)

Sets up the **Distribution** with specified width, height and kernel radius.

The position is set to 0,0 and the **Distribution** resizes to the fit the world (maintaining the same resolution specified by c and r). The kernel is initialised as a gaussian distribution and normalised.

Parameters:

c The number of columns in the **Distribution** grid.

r The number of rows in the **Distribution** grid.

b The radius of the diffusion kernel, so the width and height are 2 * b + 1.

10.9.3 Member Function Documentation

10.9.3.1 void BEAST::Distribution::AddDensity (const Vector2D & *v*, float *d*)

Adds the specified value to the distribution at the specified point.

Parameters:

v The location to alter, in **World** coordinates/scale.

d The amount to add.

10.9.3.2 template<class _Op> void BEAST::Distribution::Filter (_Op *op*) [inline]

Replaces each value of the distribution with op(oldval).

Parameters:

op A unary operator with input and output float.

10.9.3.3 template<class _Op, class _Func> void BEAST::Distribution::Filter (_Op *op*, _Func *func*) [inline]

Replaces the value at each coordinate with op(oldval, func(x, y).

This allows functions to be added and subtracted from the distribution, e.g.

```
dist.Filter(plus<double>(), GaussianNoise(0.0, 1.5))
```

Parameters:

func Any function with input (int, int) output float.

10.9.3.4 template<class _Func> void BEAST::Distribution::Plot (_Func *func*) [inline]

Plots the specified function so the value at each coordinate x, y becomes func(x, y).

Parameters:

func Any function with input (int, int) output float.

10.9.3.5 void BEAST::Distribution::Render () [virtual]

Displays the distribution with transparency according to density, up to maxConc density (= opaque).

See also:
SetMaxConc

10.9.3.6 void BEAST::Distribution::SetDecayRate (DistReal *r*) [inline]

Specifies the rate at which the distribution decays: 1.0 for no decay, 0.5 to reduce by half on every diffusion.

10.9.3.7 void BEAST::Distribution::SetDensity (const Vector2D & *v*, float *d*)

Sets the density to the specified value at the given location.

Parameters:
v Where in the distribution to update.
d The new density value.

10.9.3.8 void BEAST::Distribution::SetDiffusionSpeed (int *s*) [inline]

Sets the interval for calling diffusion, e.g.

every 2 timesteps. 0 disables diffusion altogether.

10.9.3.9 void BEAST::Distribution::SetMaxConc (float *f*) [inline]

Specifies the maximum expected concentration for purposes of display.

10.9.3.10 virtual void BEAST::Distribution::Update () [inline, virtual]

Filters the distribution every n frames, where n is specified by the diffusion speed.

See also:
SetDiffusionSpeed

Reimplemented from **BEAST::WorldObject** (p. 213).

The documentation for this class was generated from the following files:

- **distribution.h**
- **distribution.cc**

10.10 BEAST::Distribution::Kernel Struct Reference

Implements diffusion and other neighbourhood operations.

`#include <distribution.h>`

Public Member Functions

- **Kernel** (int w, int h)
- void **SetDistribution** (**Distribution** *)

  Sets the **Kernel** *up so that it can be used on the specified* **Distribution**.

- void **Set** (int x, int y, **DistReal** v)
- void **SetDivisor** (**DistReal** d)

  Sets the divisor of the kernel by dividing each value by the new divisor.

- void **Normalise** ()

  Ensures that the values in the kernel sum to 1 so that no density is lost.

- float **Get** (int x, int y)

  Returns the value at coordinate x, y of the **Kernel**.

- void **Filter** (**Distribution** *) const

  Performs one pass of the **Kernel** *over the specified* **Distribution**.

- template<class _Func> void **Plot** (_Func func)

  Plots the specified function (e.g.

Public Attributes

- int **width**
- int **height**
- int **widthjump**
- int **corner**
- **DistReal** * **kernel**
- **DistReal** const * **k**
- **DistReal** * **d**
- int **i**
- int **j**

10.10.1 Detailed Description

Implements diffusion and other neighbourhood operations.

10.10.2 Member Function Documentation

10.10.2.1 void BEAST::Distribution::Kernel::Filter (Distribution * *dist*) const

Performs one pass of the **Kernel** over the specified **Distribution**.

Kernel operations are performed by sliding the **Kernel** along the cells of the **Distribution** and replacing each value with the weighted sum of the surrounding pixels, corresponding to the weights of the **Kernel**. That probably doesn't make sense, so read about neighbourhood operations (e.g. blur) in a good image processing book.

Parameters:

dist The **Distribution** to apply the filter to.

10.10.2.2 template<class _Func> void BEAST::Distribution::Kernel::Plot (_Func *func*) [inline]

Plots the specified function (e.g.

Gaussian2D) in the kernel.

Parameters:

func Function with (int, int) input and float output.

10.10.2.3 void BEAST::Distribution::Kernel::SetDistribution (Distribution * *d*)

Sets the **Kernel** up so that it can be used on the specified **Distribution**.

Parameters:

d The **Distribution** this **Kernel** will be used on.

The documentation for this struct was generated from the following files:

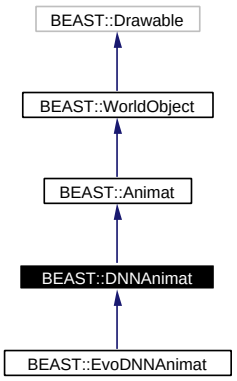
- **distribution.h**
- **distribution.cc**

10.11 BEAST::DNNAnimat Class Reference

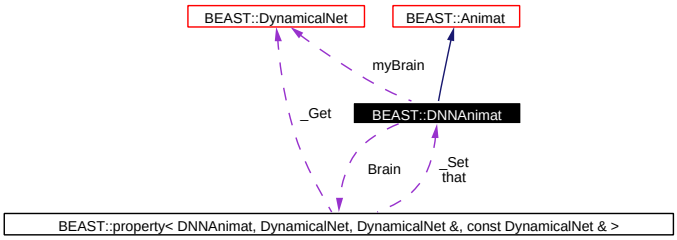
An **Animat** with a built-in dynamical network which is automatically configured depending on the Animat's sensor and control configuration.

`#include <neuralanimat.h>`

Inheritance diagram for BEAST::DNNAnimat:



Collaboration diagram for BEAST::DNNAnimat:



Public Member Functions

- virtual ~**DNNAnimat** ()

  Destructor for **DNNAnimat**, *if the* **DynamicalNet** *has been initialised, it is deleted here.*

- void **InitDNN** (int hidden=-1, int inputs=-1, int outputs=-1, bool multiInputs=true, bool multiOutputs=false)

  This method is responsible for initialising the DNNAnimat's neural network.

- virtual void **Control** ()

  The DNNAnimat's neural net is linked to its sensors and controls here.

- virtual void **Serialise** (std::ostream &) const

  Outputs the DNNAnimat's data to a stream.

- virtual void **Unserialise** (std::istream &)

  Inputs the DNNAnimat's data from a stream.

- void **SetBrain** (**DynamicalNet** &brain)
- const **DynamicalNet** & **GetBrain** () const
- bool **IsOwnBrain** () const

Public Attributes

- **property**< **DNNAnimat**, **DynamicalNet**, **DynamicalNet** &, const **DynamicalNet** & > **Brain**

Protected Member Functions

- **DynamicalNet** & **GetBrain** ()

10.11.1 Detailed Description

An **Animat** with a built-in dynamical network which is automatically configured depending on the Animat's sensor and control configuration.

See also:
EvoDNNAnimat for an evolvable version.

Todo
Review brain ownership/destructor

10.11.2 Member Function Documentation

10.11.2.1 void BEAST::DNNAnimat::Control () [virtual]

The DNNAnimat's neural net is linked to its sensors and controls here.

All sensor inputs are fed to the neural network and all control outputs are taken from the ANN's output values.

Warning:
It is assumed that there are at least as many input channels as sensors and at least as many output channels as controls. If your **Animat** is not set up in this way your needs are likely greater than can be provided for by **DNNAnimat**.

Reimplemented from **BEAST::Animat** (p. 71).

10.11.2.2 void BEAST::DNNAnimat::InitDNN (int *total* = -1, int *inputs* = -1, int *outputs* = -1, bool *multiInput* = true, bool *multiOutput* = false)

This method is responsible for initialising the DNNAnimat's neural network.

It should usually be called in the constructor of a derived class, after the sensors have been set up. Also randomises the neural network for use in evolutionary simulations.

Parameters:
hidden The number of nodes, defaults to be the same as the number of sensors on the **Animat**.

inputs The number of inputs, defaults to be the same as the number of sensors on the **Animat**.

outputs The number of outputs, defaults to be the same as the number of controls on the **Animat**.

multiInput Set to true if all inputs go to all nodes, false if not, defaults to true.

multiOutput Set to true if all outputs come from all nodes, false if not, defaults to false.

10.11.2.3 void BEAST::DNNAnimat::Serialise (std::ostream & *out*) const [virtual]

Outputs the DNNAnimat's data to a stream.

Parameters:

out A reference to an output stream.

Reimplemented from **BEAST::Animat** (p. 73).

10.11.2.4 void BEAST::DNNAnimat::Unserialise (std::istream & *in*) [virtual]

Inputs the DNNAnimat's data from a stream.

Parameters:

in A reference to an input stream.

Reimplemented from **BEAST::Animat** (p. 73).

The documentation for this class was generated from the following files:

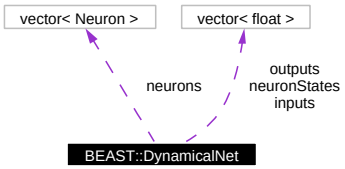
- **neuralanimat.h**
- neuralanimat.cc

10.12 BEAST::DynamicalNet Class Reference

This class implements a fully recurrent continuous (or dynamical) neural network.

`#include <dynamicalnet.h>`

Collaboration diagram for BEAST::DynamicalNet:



Public Member Functions

- **DynamicalNet** (int inputs, int outputs, int total, bool multiInputNodes=true, bool multiOutputNodes=false)

  Constructor, allows a **DynamicalNet** *to be configured with the following features: Variable number of input channels.*

- **~DynamicalNet** ()

  The destructor, does nothing at all.

- void **Init** (int inputs, int outputs, int total, bool multIn, bool multOut)

  For unserialisation purposes, the initialisation method is actually responsible for configuring the network, and is called by the constructor.

- void **Reset** ()

  Sets the output value of each neuron (i.e.

- void **SetInputChannel** (int neuron, int channel)

  Configures the network to channel inputs to different nodes.

- void **SetOutputChannel** (int neuron, int channel)

  Configures the network to channel output from different nodes.

- void **Randomise** ()

  Forces each **Neuron** *in the network to randomise itself, by calling its own Randomise member function.*

- void **SetInput** (int n, float f)

  Sets the input value for channel n to f.

- void **SetInput** (const std::vector< float > &v)

  Sets all the input values at once from a vector of floats.

- float **GetOutput** (int n) const

  Returns the output value for the specified output channel.

- const std::vector< float > & **GetOutputs** () const

  Returns all the outputs at once as a vector of floats.

- void **Fire** ()

  This is where it all happens, although all the **DynamicalNet** *class really has to do is: Clear the outputs.*

- void **SetConfiguration** (const std::vector< float > &)

  Sets the configuration of the network according to a provided vector of weights, biases and time constants.

- std::vector< float > **GetConfiguration** () const

  Returns all the weights and biases in the network, in a long list suitable for processing by a GA.

- std::string **ToString** () const

  Prints all the data in this network in a pretty format and returns it as a string.

- void **Serialise** (std::ostream &) const

  Outputs all setup and configuration data for this network to a stream.

- void **Unserialise** (std::istream &)

  Takes a string produced by **DynamicalNet::Serialise** *and turns it back into a DNN.*

Protected Member Functions

- std::vector< **Neuron** > & **GetNeurons** ()
- std::vector< float > & **GetInputs** ()
- std::vector< float > & **GetOutputs** ()
- std::vector< float > & **GetNeuronStates** ()
- bool **IsMultiInputNodes** ()
- bool **IsMultiOutputNodes** ()

Friends

- struct **Neuron**

10.12.1 Detailed Description

This class implements a fully recurrent continuous (or dynamical) neural network.

The network is configured with a number of nodes, some or all of which may also act as input nodes, and some or all of which may act as output nodes. Every node on firing takes a weighted sum of the activation states of every other node, including itself. This approach allows the network to store information and perform far more complex tasks than a feed-forward net might. The actual design of dynamical networks has many interpretations, but for reference, this one corresponds as closely as possible to the network described in Yamauchi, B. M., & Beer, R. D. (1994). Sequential behavior and learning in evolved dynamical neural networks. Adaptive Behavior 2(3), 219–246. `http://citeseer.nj.nec.com/yamauchi94sequential.html`

See also:
FeedForwardNet

10.12.2 Constructor & Destructor Documentation

10.12.2.1 BEAST::DynamicalNet::DynamicalNet (int *i*, int *o*, int *t*, bool *mi* = true, bool *mo* = false)

Constructor, allows a **DynamicalNet** to be configured with the following features: Variable number of input channels.

- * - Variable number of output channels.
- Variable total number of nodes.
- Option of all input channels going to all nodes, in which case each node receives a weighted sum of the inputs. This defaults to True.
- Option of all output channels coming from all nodes, in which case each node contributes a weighted value to each output. This defaults to false. Input nodes are enumerated starting at the first node, although they can be altered to point to different nodes. Output nodes are enumerated from the last node and may also be altered to point to different channels. If inputs + outputs > total, the network will be configured with the difference of nodes in the middle acting as both inputs and outputs.

Parameters:

i The number of input nodes.

o The number of output nodes.

t The total number of nodes.

mi True if multiple inputs per node (default true).

mo True if multiple outputs per node (default false).

See also:

DynamicalNet::Init
DynamicalNet::SetInputChannel
DynamicalNet::SetOutputChannel

10.12.3 Member Function Documentation

10.12.3.1 void BEAST::DynamicalNet::Fire ()

This is where it all happens, although all the **DynamicalNet** class really has to do is: Clear the outputs.

- * - Fire every **Neuron**.
- Retrieve and store every Neuron's output.

See also:

DynamicalNet::Neuron::Fire

10.12.3.2 vector< float > BEAST::DynamicalNet::GetConfiguration () const

Returns all the weights and biases in the network, in a long list suitable for processing by a GA.

Note that nearly all the values are initialised in the range [-1,1], except for the time constants which range between 1 and 70. The time constants are therefore stored as their natural log in the configuration output which makes for more sensible alteration by the GA.

Returns:
An ordered vector of floats describing the weights, biases and time constants.

See also:
DynamicalNet::Neuron::GetConfiguration
DynamicalNet::SetConfiguration

10.12.3.3 void BEAST::DynamicalNet::Init (int *i*, int *o*, int *t*, bool *mi*, bool *mo*)

For unserialisation purposes, the initialisation method is actually responsible for configuring the network, and is called by the constructor.

Init also calls Reset.

Parameters:
i The number of input nodes.

o The number of output nodes.

t The total number of nodes.

mi True if multiple inputs per node (default true).

mo True if multiple outputs per node (default false).

See also:
DynamicalNet::Reset

10.12.3.4 void BEAST::DynamicalNet::Randomise ()

Forces each **Neuron** in the network to randomise itself, by calling its own Randomise member function.

See also:
Neuron::Randomise

10.12.3.5 void BEAST::DynamicalNet::Reset ()

Sets the output value of each neuron (i.e.

its current output) to 0. This is always done on initialisation of the network.

See also:
DynamicalNet::Init

10.12.3.6 void BEAST::DynamicalNet::Serialise (std::ostream & *out*) const

Outputs all setup and configuration data for this network to a stream.

Parameters:
out An output stream.

See also:
DynamicalNet::Unserialise

Todo
Serialise input/output channel/node config

10.12.3.7 void BEAST::DynamicalNet::SetConfiguration (const std::vector< float > & *config*)

Sets the configuration of the network according to a provided vector of weights, biases and time constants.

Parameters:
config An ordered vector of floats containing weights, biases and time constants.

See also:
DynamicalNet::Neuron::SetConfiguration
DynamicalNet::GetConfiguration

10.12.3.8 void BEAST::DynamicalNet::SetInputChannel (int *neuron*, int *channel*)

Configures the network to channel inputs to different nodes.

Has no effect if the net is configured with multiple input nodes per channel.

Parameters:
neuron The number of the node to direct the input to.
channel The number of the channel to be redirected.

See also:
DynamicalNet::SetOutputChannel

10.12.3.9 void BEAST::DynamicalNet::SetOutputChannel (int *neuron*, int *channel*)

Configures the network to channel output from different nodes.

Has no effect if the net is configured with multiple output nodes per channel.

Parameters:
neuron The number of the node to be redirected.
channel The number of the channel to redirect it to.

See also:
DynamicalNet::SetInputChannel

10.12.3.10 string BEAST::DynamicalNet::ToString () const

Prints all the data in this network in a pretty format and returns it as a string.

Returns:
An STL string.

10.12.3.11 void BEAST::DynamicalNet::Unserialise (std::istream & *in*)

Takes a string produced by **DynamicalNet::Serialise** and turns it back into a DNN.

Parameters:
input The input string.

See also:
DynamicalNet::Serialise

Todo
Unserialise input/output channel/node config

The documentation for this class was generated from the following files:

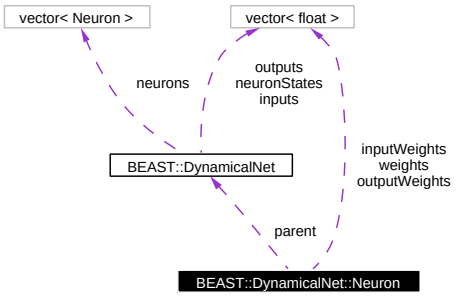
- **dynamicalnet.h**
- **dynamicalnet.cc**

10.13 BEAST::DynamicalNet::Neuron Struct Reference

Unlike the **FeedForwardNet**, the **Neuron** in **DynamicalNet** is more worthy of its name, since nearly all the processing of the DNN's firing algorithm occurs here.

`#include <dynamicalnet.h>`

Collaboration diagram for BEAST::DynamicalNet::Neuron:



Public Member Functions

- **Neuron** (int i, int o, int t, int inCh, int outCh, **DynamicalNet** *p)

  Constructor: sets the number of inputs, outputs and weights for this neuron, as well as input and output channels.

- void **Randomise** ()

  Fills all input, output and internal weights with random numbers in the range [-1,1].

- void **Fire** ()

  This is where everything /actually/ happens - this method calculates the amount by which the activation value changes.

- float **GetOutput** () const
- void **GetConfiguration** (std::vector< float > &config) const

  Copies all weights, bias and time constant into the provided vector.

- std::vector< float >::const_iterator **SetConfiguration** (std::vector< float >::const_iterator config)

  Sets the Neuron's configuration according to the input, which is an iterator of a vector of floats.

- std::string **ToString** () const

  Prints all the data in this **Neuron** *in a pretty format and returns it as a string.*

Static Public Member Functions

- float **RandomNum** ()

  Returns a random number in the range [-1,1].

- float **Sigmoid** (float y)

The standard ANN squashing function.

Public Attributes

- int **inputChannel**

  This neuron's input channel, or -1 for all.

- int **outputChannel**

  This neuron's output channel, or -1 for all.

- float **output**

  The neuron's current output.

- float **activation**

  The neuron's current activation.

- std::vector< float > **inputWeights**

  The input weights.

- std::vector< float > **outputWeights**

  The output weights.

- std::vector< float > **weights**

  Weights for every neuron, bias and time constant.

- **DynamicalNet** * **parent**

  The net this neuron belongs to.

- float **bias**

  The bias, taken from weights.end() - 2.

- float **timeConstant**

  Taken from weights.end() - 1.

10.13.1 Detailed Description

Unlike the **FeedForwardNet**, the **Neuron** in **DynamicalNet** is more worthy of its name, since nearly all the processing of the DNN's firing algorithm occurs here.

Each **Neuron** contains a weight for each other neuron (and one for itself), one or multiple input channels and one or multiple output channels, with a series of weights in each case if multiple channels are in use. The neuron also keeps track of its activation value and its output value, which is simply the biased, squashed activation value.

See also:
FeedForwardNet::Neuron

10.13.2 Constructor & Destructor Documentation

10.13.2.1 BEAST::DynamicalNet::Neuron::Neuron (int *i*, int *o*, int *t*, int *inCh*, int *outCh*, DynamicalNet * *p*) [inline]

Constructor: sets the number of inputs, outputs and weights for this neuron, as well as input and output channels.

Parameters:

i The number of input weights.

o The number of output weights.

t The total number of internal weights.

inCh The number of input channels.

outCh The number of output channels.

p A pointer to the parent **DynamicalNet**

10.13.3 Member Function Documentation

10.13.3.1 void BEAST::DynamicalNet::Neuron::Fire ()

This is where everything /actually/ happens - this method calculates the amount by which the activation value changes.

The code of the Firing method has been carefully commented, here is a summary:

- We start by subtracting the last round's activation.
- A weighted sum of the previous neuron outputs is taken.
- Inputs are applied, either as a weighted sum from all channels or as an individual input.
- The total is divided by the time constant...
- ... and added back to the previous activation.
- The new activation is then biased and squashed to produce the output.
- The output is applied to the relevant output channels.

See also:

Sigmoid

10.13.3.2 void BEAST::DynamicalNet::Neuron::GetConfiguration (std::vector< float > & *config*) const

Copies all weights, bias and time constant into the provided vector.

Parameters:

config A reference to the vector into which the configuration must be copied.

See also:

DynamicalNet::GetConfiguration
DynamicalNet::Neuron::SetConfiguration

10.13.3.3 void BEAST::DynamicalNet::Neuron::Randomise ()

Fills all input, output and internal weights with random numbers in the range [-1,1].

The time constant is also randomised with a value between 1 and 70.

10.13.3.4 vector< float >::const_iterator BEAST::DynamicalNet::Neuron::SetConfiguration (std::vector< float >::const_iterator *config*)

Sets the Neuron's configuration according to the input, which is an iterator of a vector of floats.

This has been done to enable easy configuration by a **DynamicalNet** of its neurons, without knowing how many values are required for each **Neuron**. The **Neuron** may therefore take what it needs and return an iterator pointing to the rest of the configuration data.

Parameters:

config An iterator pointing to the current position in the configuration data.

Returns:

An iterator pointing to the next place after this Neuron's data finishes.

10.13.3.5 string BEAST::DynamicalNet::Neuron::ToString () const

Prints all the data in this **Neuron** in a pretty format and returns it as a string.

Returns:

An STL string.

The documentation for this struct was generated from the following files:

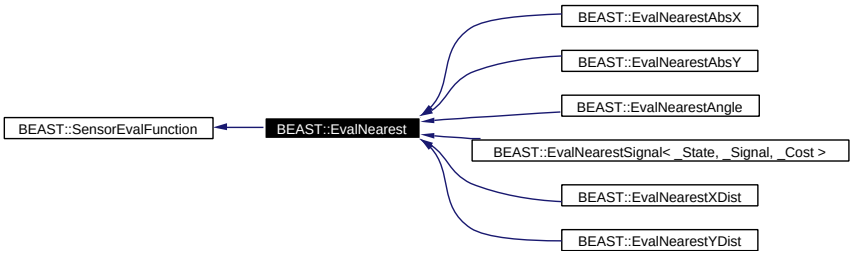
- **dynamicalnet.h**
- **dynamicalnet.cc**

10.14 BEAST::EvalNearest Class Reference

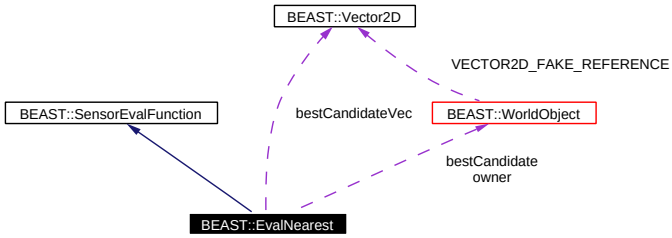
Keeps a tally of the nearest point passed in and returns it with GetOutput.

`#include <sensorfunctors.h>`

Inheritance diagram for BEAST::EvalNearest:



Collaboration diagram for BEAST::EvalNearest:



Public Member Functions

- **EvalNearest** (**WorldObject** *o, double range)
- virtual void **Reset** ()
- virtual void **operator()** (**WorldObject** *obj, const **Vector2D** &loc)
- virtual double **GetOutput** () const

Public Attributes

- **WorldObject** * **owner**
- double **range**
- double **nearestSoFar**
- **WorldObject** * **bestCandidate**
- **Vector2D** **bestCandidateVec**

10.14.1 Detailed Description

Keeps a tally of the nearest point passed in and returns it with GetOutput.

Also keeps a pointer to the nearest candidate and a copy of the nearest point on that candidate, this data can be accessed by an adapter such as EvalXDist and EvalAngle.

The documentation for this class was generated from the following file:

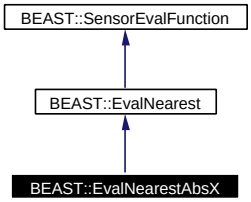
- sensorfunctors.h

10.15 BEAST::EvalNearestAbsX Class Reference

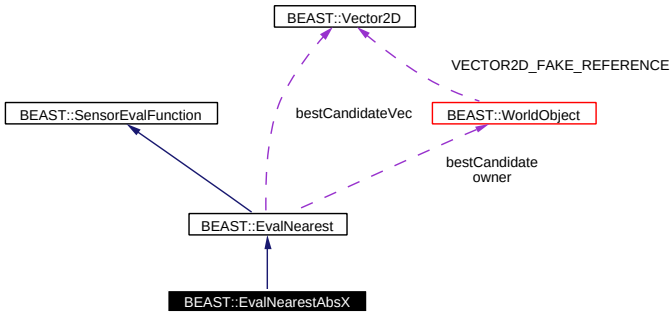
Returns the absolute x position of the nearest target.

`#include <sensorfunctors.h>`

Inheritance diagram for BEAST::EvalNearestAbsX:



Collaboration diagram for BEAST::EvalNearestAbsX:



Public Member Functions

- **EvalNearestAbsX** (**WorldObject** *o, double range)
- virtual double **GetOutput** () const

10.15.1 Detailed Description

Returns the absolute x position of the nearest target.

Most effective when coupled with **EvalNearestAbsY**.

The documentation for this class was generated from the following file:

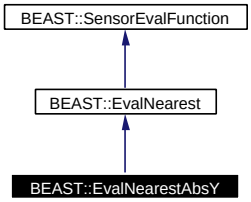
- sensorfunctors.h

10.16 BEAST::EvalNearestAbsY Class Reference

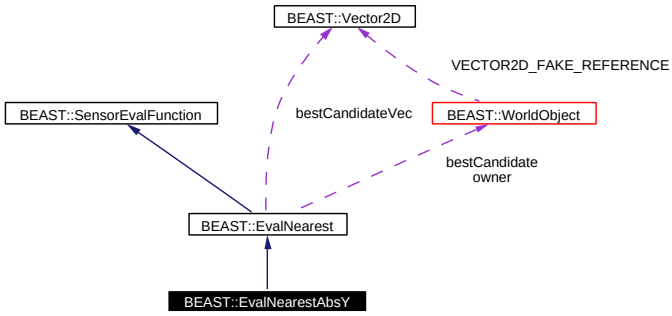
Returns the absolute y position of the nearest target.

`#include <sensorfunctors.h>`

Inheritance diagram for BEAST::EvalNearestAbsY:



Collaboration diagram for BEAST::EvalNearestAbsY:



Public Member Functions

- **EvalNearestAbsY** (**WorldObject** *o, double range)
- virtual double **GetOutput** () const

10.16.1 Detailed Description

Returns the absolute y position of the nearest target.

Most effective when coupled with **EvalNearestAbsX**.

The documentation for this class was generated from the following file:

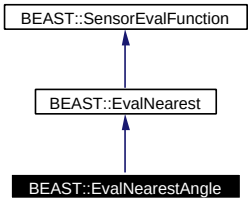
- sensorfunctors.h

10.17 BEAST::EvalNearestAngle Class Reference

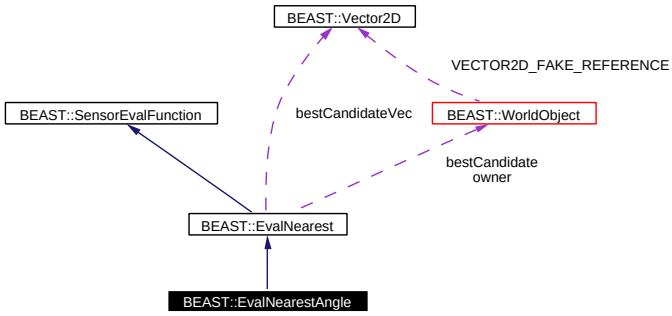
Returns the normalised angle to the nearest target.

`#include <sensorfunctors.h>`

Inheritance diagram for BEAST::EvalNearestAngle:



Collaboration diagram for BEAST::EvalNearestAngle:



Public Member Functions

- **EvalNearestAngle** (**WorldObject** *o, double range)
- virtual double **GetOutput** () const

10.17.1 Detailed Description

Returns the normalised angle to the nearest target.

The documentation for this class was generated from the following file:

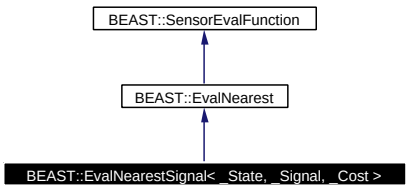
- sensorfunctors.h

10.18 BEAST::EvalNearestSignal< _State, _Signal, _Cost > Class Template Reference

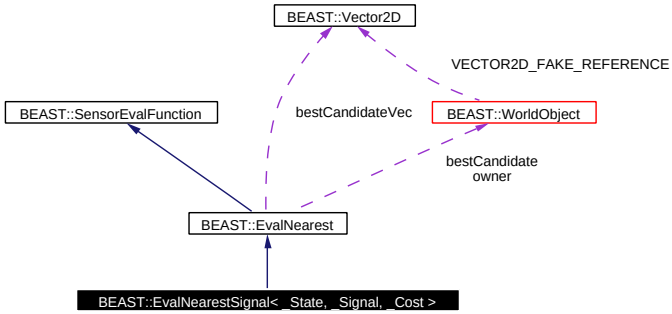
Sensor evaluation functor: returns the signal of the nearest individual.

`#include <signaller.h>`

Inheritance diagram for BEAST::EvalNearestSignal< _State, _Signal, _Cost >:



Collaboration diagram for BEAST::EvalNearestSignal< _State, _Signal, _Cost >:



Public Member Functions

- **EvalNearestSignal** (**WorldObject** *o, double range)
- virtual double **GetOutput** () const

  Returns the signal number (as a double) of the nearest signaller, or 0.0 if no signaller was found (or the signaller is signalling 0).

10.18.1 Detailed Description

template<typename _State, typename _Signal, typename _Cost = float> class BEAST::EvalNearestSignal< _State, _Signal, _Cost >

Sensor evaluation functor: returns the signal of the nearest individual.

The documentation for this class was generated from the following file:

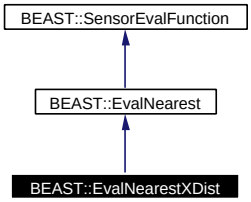
- **signaller.h**

10.19 BEAST::EvalNearestXDist Class Reference

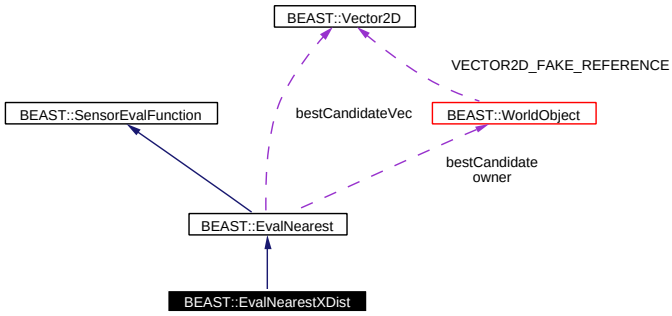
Returns the vertical distance to the nearest target.

`#include <sensorfunctors.h>`

Inheritance diagram for BEAST::EvalNearestXDist:



Collaboration diagram for BEAST::EvalNearestXDist:



Public Member Functions

- **EvalNearestXDist** (**WorldObject** *o, double range)
- virtual double **GetOutput** () const

10.19.1 Detailed Description

Returns the vertical distance to the nearest target.

Most effective when coupled with **EvalNearestYDist**.

The documentation for this class was generated from the following file:

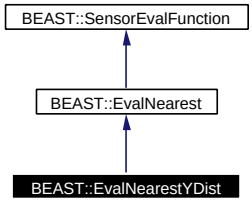
- sensorfunctors.h

10.20 BEAST::EvalNearestYDist Class Reference

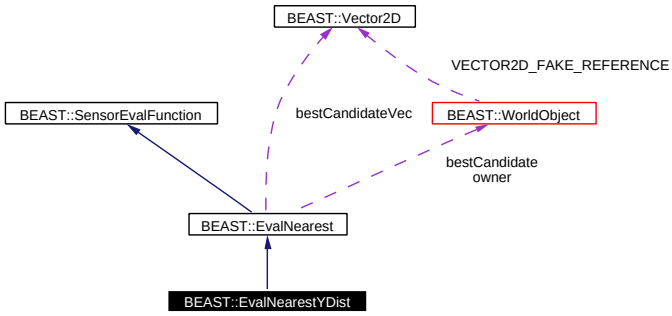
Returns the horizontal distance to the nearest target.

`#include <sensorfunctors.h>`

Inheritance diagram for BEAST::EvalNearestYDist:



Collaboration diagram for BEAST::EvalNearestYDist:



Public Member Functions

- **EvalNearestYDist** (**WorldObject** *o, double range)
- virtual double **GetOutput** () const

10.20.1 Detailed Description

Returns the horizontal distance to the nearest target.

Most effective when coupled with **EvalNearestXDist**.

The documentation for this class was generated from the following file:

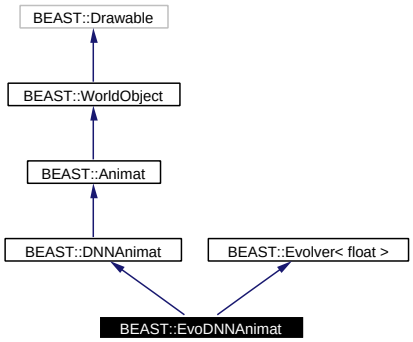
- sensorfunctors.h

10.21 BEAST::EvoDNNAnimat Class Reference

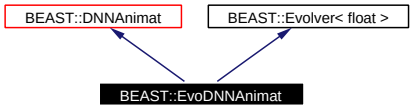
An evolvable version of **DNNAnimat** with GetGenotype/SetGenotype methods already set up.

`#include <neuralanimat.h>`

Inheritance diagram for BEAST::EvoDNNAnimat:



Collaboration diagram for BEAST::EvoDNNAnimat:



Public Member Functions

- virtual void **SetGenotype** (const std::vector< float > &g)
- virtual genotype_type **GetGenotype** () const

  Returns the genotype.

10.21.1 Detailed Description

An evolvable version of **DNNAnimat** with GetGenotype/SetGenotype methods already set up.

See also:
DNNAnimat

The documentation for this class was generated from the following file:

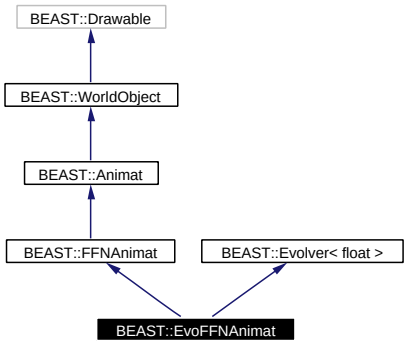
- **neuralanimat.h**

10.22 BEAST::EvoFFNAnimat Class Reference

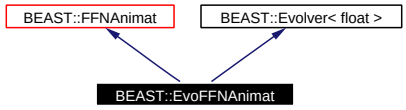
An evolvable version of **FFNAnimat** with GetGenotype/SetGenotype methods already set up.

`#include <neuralanimat.h>`

Inheritance diagram for BEAST::EvoFFNAnimat:



Collaboration diagram for BEAST::EvoFFNAnimat:



Public Member Functions

- virtual void **SetGenotype** (const std::vector< float > &g)
- virtual genotype_type **GetGenotype** () const

  Returns the genotype.

10.22.1 Detailed Description

An evolvable version of **FFNAnimat** with GetGenotype/SetGenotype methods already set up.

See also:
FFNAnimat

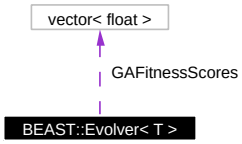
The documentation for this class was generated from the following file:

- **neuralanimat.h**

10.23 BEAST::Evolver< T > Class Template Reference

`#include <geneticalgorithm.h>`

Collaboration diagram for BEAST::Evolver< T >:



Public Types

- typedef T **gene_type**
- typedef std::vector< T > **genotype_type**

Public Member Functions

- virtual genotype_type **GetGenotype** () const=0

  Returns the genotype.

- virtual void **SetGenotype** (const genotype_type &)=0

  Sets the genotype.

- virtual float **GetFitness** () const=0

  Returns the fitness.

- virtual void **StoreFitness** ()

  Stores current fitness, overload this for one way of resetting individuals' internal fitness scores each assessment (another way might be e.g.

Public Attributes

- std::vector< float > **GAFitnessScores**

  A list of previous scores.

- float **GAProbability**

  Used by **GeneticAlgorithm**.

- float **GAFixedFitness**

  Used by **GeneticAlgorithm**.

- genotype_type **PSOBestSolution**
- float **PSOBestFitness**

10.23.1 Detailed Description

template<typename T> class BEAST::Evolver< T >

The **Evolver** class is an abstract base class from which you may derive the objects which will comprise your population. The best approach is to use multiple inheritance and create an evolvable subclass of whatever it is you want to use the GA on.

The type specified must match the type specified in the first template parameter of your GA, otherwise it won't work. You are not limited to basic types for your genes, any class may be used, but you will need to provide a suitable mutation operator for that class.

10.23.2 Member Function Documentation

10.23.2.1 template<typename T> virtual void BEAST::Evolver< T >::StoreFitness () [inline, virtual]

Stores current fitness, overload this for one way of resetting individuals' internal fitness scores each assessment (another way might be e.g.

to overload Init).

Note:

If you are using only one fitness score per individual you do not need to call StoreFitness, the GA will simply get the fitness from the fitness function

The documentation for this class was generated from the following file:

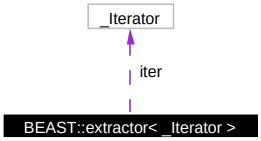
- **geneticalgorithm.h**

10.24 BEAST::extractor< _Iterator > Struct Template Reference

This is a function object which can be used for copying from an iterator when the number of input values is unknown.

`#include <serialfuncs.h>`

Collaboration diagram for BEAST::extractor< _Iterator >:



Public Types

- typedef _Iterator::value_type **_ValueType**

Public Member Functions

- **extractor** (_Iterator &Iter)
- _ValueType **operator()** ()

Public Attributes

- _Iterator **iter**
- _ValueType **tempVal**

10.24.1 Detailed Description

template<class _Iterator> struct BEAST::extractor< _Iterator >

This is a function object which can be used for copying from an iterator when the number of input values is unknown.

Best constructed using the extract helper function

See also:
extract

The documentation for this struct was generated from the following file:

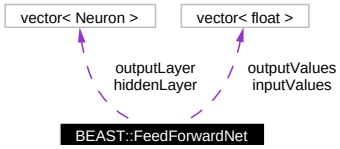
- **serialfuncs.h**

10.25 BEAST::FeedForwardNet Class Reference

This is an implementation of a simple two-layer feed-forward neural network.

`#include <feedforwardnet.h>`

Collaboration diagram for BEAST::FeedForwardNet:



Public Member Functions

- **FeedForwardNet** (int inputs, int outputs, int hidden=0, bool sig=true, bool bias=true)

  The constructor for **FeedForwardNet**, *which configures an empty network with the specified dimensions and features.*

- virtual ~**FeedForwardNet** ()

  The destructor for **FeedForwardNet**, *doesn't do very much at all.*

- void **Init** (int in, int out, int hid, bool sig, bool bias)

  Initialises the **FeedForwardNet** *with the specified dimensions and features.*

- void **Randomise** ()

  Initialises every weight and bias in the net with a random number in the range [-1,1].

- void **SetInput** (int n, float f)

  Sets the current value of the specified input.

- void **SetInput** (const std::vector< float > &v)
- float **GetOutput** (int n) const

  Returns the current value of the specified output.

- const std::vector< float > & **GetOutput** () const

  Returns the outputs of the neural net as a vector of floats.

- void **Fire** ()

  This is the main method of the Feed Forward Network, where inputs are processed to calculated the output values.

- void **SetConfiguration** (const std::vector< float > &)

  Takes the output of GetConfiguration and configures the weights and biases of the network accordingly.

- std::vector< float > **GetConfiguration** () const

  Returns all the weights and biases in the network as a vector of floats, ideal for representing the network in an evolutionary algorithm.

- std::string **ToString** () const

  Outputs the network's data in a pretty format.

- int **GetInputs** () const

  Returns the number of inputs on this network.

- int **GetOutputs** () const

  Returns the number of outputs on this network.

- int **GetHidden** () const

  Returns the number of hidden nodes on this network.

- bool **IsSigmoid** () const

  Returns true if a sigmoid activation function is in use.

- bool **IsBiasNode** () const

  Returns true if a bias term is added to each node.

- void **Serialise** (std::ostream &) const

  Outputs all the data for the net to the specified output stream.

- void **Unserialise** (std::istream &)

  Reconstructs the network from an input stream.

Protected Member Functions

- std::vector< float > & **GetInputValues** ()
- std::vector< float > & **GetOutputValues** ()
- std::vector< **Neuron** > & **GetHiddenLayer** ()
- std::vector< **Neuron** > & **GetOutputLayer** ()
- float **ActivationFunction** (float n)

  The squashing function for the network, either a sigmoid or threshold function.

- int **GetConfigurationLength** () const

  Calculates the expected length of the configuration data for the network.

Static Protected Member Functions

- float **RandomNum** ()

  Returns a number between -1 and 1.

10.25.1 Detailed Description

This is an implementation of a simple two-layer feed-forward neural network.

You may specify the number of inputs, the number of nodes in the output layer and the number of nodes in the hidden layer. Every node has an associated bias term, although this can be switched off. The default activation function is sigmoid, although it may be swtiched off and is then replaced by a simple threshold function. It is also possible to configure a **FeedForwardNet** with a hidden layer size of 0, in which case the net becomes a perceptron, bypassing the hidden layer entirely.

See also:
DynamicalNet

10.25.2 Constructor & Destructor Documentation

10.25.2.1 BEAST::FeedForwardNet::FeedForwardNet (int *in*, int *out*, int *hid* = 0, bool *sig* = true, bool *bias* = true)

The constructor for **FeedForwardNet**, which configures an empty network with the specified dimensions and features.

Parameters:
in The number of inputs.

out The number of outputs.

hig The size of the hidden layer. If hid is 0 then the FFN acts as a perceptron.

sig Whether or not the net will use a sigmoid activation function, defaults to true.

bias Whether of not each node has a bias value, defaults to true.

10.25.3 Member Function Documentation

10.25.3.1 float BEAST::FeedForwardNet::ActivationFunction (float *n*) [protected]

The squashing function for the network, either a sigmoid or threshold function.

Parameters:
n The input value.

Returns:
The output value.

10.25.3.2 void BEAST::FeedForwardNet::Fire ()

This is the main method of the Feed Forward Network, where inputs are processed to calculated the output values.

The Fire method assumes that inputs have previously been set using SetInput.

See also:
FeedForwardNet::SetInput

10.25.3.3 vector< float > BEAST::FeedForwardNet::GetConfiguration () const

Returns all the weights and biases in the network as a vector of floats, ideal for representing the network in an evolutionary algorithm.

No information about the dimensions of the network is returned by this method - to return complete configuration data use Serialise.

Returns:
A vector containing the weights and biases of the network.

See also:
FeedForwardNet::SetConfiguration
FeedForwardNet::Serialise

10.25.3.4 int BEAST::FeedForwardNet::GetConfigurationLength () const [protected]

Calculates the expected length of the configuration data for the network.

Returns:
The configuration length.

See also:
FeedForwardNet::GetConfiguration

10.25.3.5 const std::vector<float>& BEAST::FeedForwardNet::GetOutput () const [inline]

Returns the outputs of the neural net as a vector of floats.

10.25.3.6 float BEAST::FeedForwardNet::GetOutput (int *n*) const [inline]

Returns the current value of the specified output.

Parameters:
n The number of the output to return.

Warning:
This method does not check if the output specified exists, if you are unsure, check using GetOutputs first.

10.25.3.7 void BEAST::FeedForwardNet::Init (int *in*, int *out*, int *hid*, bool *sig*, bool *bias*)

Initialises the **FeedForwardNet** with the specified dimensions and features.

After this method is called, all weights in the net will be set to 0.

Parameters:

in The number of inputs.

out The number of outputs.

hig The size of the hidden layer. If hid is 0 then the FFN acts as a perceptron.

sig Whether or not the net will use a sigmoid activation function, defaults to true.

bias Whether of not each node has a bias value, defaults to true.

10.25.3.8 void BEAST::FeedForwardNet::Serialise (std::ostream & *out*) const

Outputs all the data for the net to the specified output stream.

Parameters:

out The output stream.

10.25.3.9 void BEAST::FeedForwardNet::SetConfiguration (const std::vector< float > & *config*)

Takes the output of GetConfiguration and configures the weights and biases of the network accordingly.

Note that since no data about the dimensions of the network is stored in the output of GetConfiguration, this method assumes that the network has already been set up with dimensions matching those of the input configuration. A vector containing weights and biases.

See also:

FeedForwardNet::GetConfiguration
FeedForwardNet::Unserialise

10.25.3.10 void BEAST::FeedForwardNet::SetInput (int *n*, float *f*) [inline]

Sets the current value of the specified input.

Parameters:

n The number of the input to set.

f The value to set it to.

Warning:

This method does not check if the input specified exists, if you are unsure, check using GetInputs first.

10.25.3.11 string BEAST::FeedForwardNet::ToString () const

Outputs the network's data in a pretty format.

Returns:

An STL string.

10.25.3.12 void BEAST::FeedForwardNet::Unserialise (std::istream & *in*)

Reconstructs the network from an input stream.

Parameters:
in The input stream.

The documentation for this class was generated from the following files:

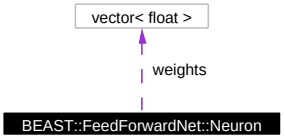
- **feedforwardnet.h**
- feedforwardnet.cc

10.26 BEAST::FeedForwardNet::Neuron Struct Reference

This member struct simply encapsulates the weighted sum function which has to be performed on the weights of each node when the net fires.

`#include <feedforwardnet.h>`

Collaboration diagram for BEAST::FeedForwardNet::Neuron:



Public Member Functions

- **Neuron** (int n)

  Constructor, simply specifies the number of weights for the **Neuron**.

- **Neuron** (std::vector< float > w)

  Constructor, initialises the neuron with a vector containing its weights.

- float **WeightedSum** (std::vector< float > &) const

  Returns the sum of each value of the input vector, multiplied by the Neuron's respective weight values.

Public Attributes

- std::vector< float > **weights**

  The weight values for this neuron (including bias).

10.26.1 Detailed Description

This member struct simply encapsulates the weighted sum function which has to be performed on the weights of each node when the net fires.

Note that **Neuron** is an inaccurate term for this class since it doesn't actually do everything a neuron in a "real" net might - biasing and squashing (via the activation function) occur outside **Neuron**, in the net's main firing function. This has been done for optimisation.

See also:
Fire
ActivationFunction
DynamicalNet::Neuron

10.26.2 Member Function Documentation

10.26.2.1 float BEAST::FeedForwardNet::Neuron::WeightedSum (std::vector< float > & *input*) const

Returns the sum of each value of the input vector, multiplied by the Neuron's respective weight values.

Parameters:

input A vector of floats.

The documentation for this struct was generated from the following files:

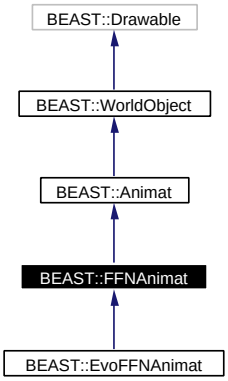
- **feedforwardnet.h**
- feedforwardnet.cc

10.27 BEAST::FFNAnimat Class Reference

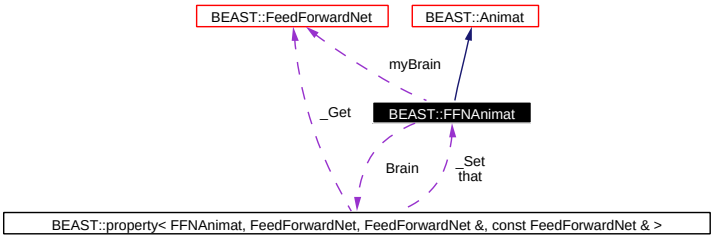
An **Animat** with a built-in feed-forward network which is automatically configured depending on the Animat's sensor and control configuration.

`#include <neuralanimat.h>`

Inheritance diagram for BEAST::FFNAnimat:



Collaboration diagram for BEAST::FFNAnimat:



Public Member Functions

- **FFNAnimat** ()

  Constructor.

- virtual ~**FFNAnimat** ()

  Destructor for **FFNAnimat**, *if the* **FeedForwardNet** *has been initialised, it is deleted here.*

- void **InitFFN** (int hidden=-1, int inputs=-1, int outputs=-1)

  This method is responsible for initialising the FFNAnimat's neural network.

- virtual void **Control** ()

  The FFNAnimat's neural net is linked to its sensors and controls here.

- virtual void **Serialise** (std::ostream &) const

  Outputs the FFNAnimat's data to a stream.

- virtual void **Unserialise** (std::istream &)

  Inputs the FFNAnimat's data from a stream.

- void **SetBrain** (**FeedForwardNet** &)
- const **FeedForwardNet** & **GetBrain** () const
- bool **IsOwnBrain** () const

Public Attributes

- **property**< **FFNAnimat**, **FeedForwardNet**, **FeedForwardNet** &, const **FeedForwardNet** & > **Brain**

Protected Member Functions

- **FeedForwardNet** & **GetBrain** ()

10.27.1 Detailed Description

An **Animat** with a built-in feed-forward network which is automatically configured depending on the Animat's sensor and control configuration.

See also:
EvoFFNAnimat for an evolvable version.

10.27.2 Member Function Documentation

10.27.2.1 void BEAST::FFNAnimat::Control () [virtual]

The FFNAnimat's neural net is linked to its sensors and controls here.

All sensor inputs are fed to the neural network and all control outputs are taken from the ANN's output values.

Warning:
It is assumed that there are at least as many input nodes as sensors and at least as many output nodes as controls. If your **Animat** is not set up in this way your needs are likely greater than can be provided for by **FFNAnimat**.

Reimplemented from **BEAST::Animat** (p. 71).

10.27.2.2 void BEAST::FFNAnimat::InitFFN (int *hidden* = -1, int *inputs* = -1, int *outputs* = -1)

This method is responsible for initialising the FFNAnimat's neural network.

It should usually be called in the constructor of a derived class, after the sensors have been set up. Also randomises the neural network for use in evolutionary simulations.

Parameters:
hidden The number of hidden nodes, defaults to be the same as the number of inputs.

inputs The number of inputs, defaults to be the same as the number of sensors on the **Animat**.

outputs The number of outputs, defaults to be the same as the number of controls on the **Animat**.

10.27.2.3 void BEAST::FFNAnimat::Serialise (std::ostream & *out*) const [virtual]

Outputs the FFNAnimat's data to a stream.

Parameters:

out A reference to an output stream.

Reimplemented from **BEAST::Animat** (p. 73).

10.27.2.4 void BEAST::FFNAnimat::Unserialise (std::istream & *in*) [virtual]

Inputs the FFNAnimat's data from a stream.

Parameters:

in A reference to an input stream.

Reimplemented from **BEAST::Animat** (p. 73).

The documentation for this class was generated from the following files:

- **neuralanimat.h**
- neuralanimat.cc

10.28 BEAST::Gaussian2D Struct Reference

Plots a two-dimensional Gaussian function in a distribution or distribution kernel.

`#include <bacteria.h>`

Public Member Functions

- **Gaussian2D** (int centerx, int centery, double sd=1.0, double scale=1.0)
- double **operator()** (int x, int y)

Public Attributes

- int **cx**
- int **cy**
- double **sdsq**
- double **s**

10.28.1 Detailed Description

Plots a two-dimensional Gaussian function in a distribution or distribution kernel.

The documentation for this struct was generated from the following file:

- **bacteria.h**

10.29 BEAST::GaussianNoise Struct Reference

Plots normally distributed noise in a distribution.

`#include <bacteria.h>`

Public Member Functions

- **GaussianNoise** (double mean, double stddev)
- double **operator()** (int, int)

Public Attributes

- double **m**
- double **sd**

10.29.1 Detailed Description

Plots normally distributed noise in a distribution.

10.29.2 Constructor & Destructor Documentation

10.29.2.1 BEAST::GaussianNoise::GaussianNoise (double *mean*, double *stddev*) [inline]

Parameters:

mean The mean value of the noise.

sd The standard deviation of the noise.

Warning:

Can return negative values.

The documentation for this struct was generated from the following file:

- **bacteria.h**

10.30 BEAST::GaussianRing2D Struct Reference

Plots a two dimensional Gaussian ring.

`#include <bacteria.h>`

Public Member Functions

- **GaussianRing2D** (int centerx, int centery, double mean=1.0, double stddev=1.0, double scale=1.0)

  Plots a two dimensional ring with a Gaussian function.

- double **operator()** (int X, int Y)

Public Attributes

- int **cx**
- int **cy**

  The centre of the Gaussian function.

- double **m**

  Store the mean, which is the radius.

- double **sd**
- double **sdsq**

  The standard deviation and square thereof.

- double **s**

  Store the scale.

- double **k**

10.30.1 Detailed Description

Plots a two dimensional Gaussian ring.

10.30.2 Constructor & Destructor Documentation

10.30.2.1 BEAST::GaussianRing2D::GaussianRing2D (int *centerx*, int *centery*, double *mean* = 1.0, double *stddev* = 1.0, double *scale* = 1.0) [inline]

Plots a two dimensional ring with a Gaussian function.

Parameters:

centerx The X coordinate of the center.

centery The Y coordinate of the center.

mean The mean for the Gaussian function.

standard deviation for the Gaussian function.

scale The value to output within the ring area.

The documentation for this struct was generated from the following file:

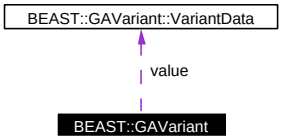
- **bacteria.h**

10.31 BEAST::GAVariant Struct Reference

This is a general purpose data type which takes five basic data types: int, float, double, char and bool.

`#include <geneticalgorithm.h>`

Collaboration diagram for BEAST::GAVariant:



Public Member Functions

- **GAVariant** (int v=0)

  Constructs **GAVariant** *from an int, doubles up as default constructor.*

- **GAVariant** (float v)

  Constructs **GAVariant** *from a float.*

- **GAVariant** (double v)

  Constructs **GAVariant** *from a double.*

- **GAVariant** (char v)

  Constructs **GAVariant** *from a char.*

- **GAVariant** (bool v)

  Constructs **GAVariant** *from a bool.*

- **operator int** () const

  int typecast operator

- **operator float** () const

  float typecast operator

- **operator double** () const

  double typecast operator

- **operator char** () const

  char typecast operator

- **operator bool** () const

  bool typecast operator

Public Attributes

- **BEAST::GAVariant::VariantData value**

  Union of five data types for **GAVariant**.

- **GAVariantType type**

  A flag signifying the type of data currently held.

10.31.1 Detailed Description

This is a general purpose data type which takes five basic data types: int, float, double, char and bool.

A union is used to ensure that **GAVariant** never takes up more space than the size of the largest datatype plus the size of one enumeration type.

The documentation for this struct was generated from the following file:

- **geneticalgorithm.h**

10.32 BEAST::GAVariant::VariantData Union Reference

Union of five data types for **GAVariant**.

`#include <geneticalgorithm.h>`

Public Member Functions

- **VariantData** (int v)
- **VariantData** (float v)
- **VariantData** (double v)
- **VariantData** (char v)
- **VariantData** (bool v)

Public Attributes

- int **i**
- float **f**
- double **d**
- char **c**
- bool **b**

10.32.1 Detailed Description

Union of five data types for **GAVariant**.

The documentation for this union was generated from the following file:

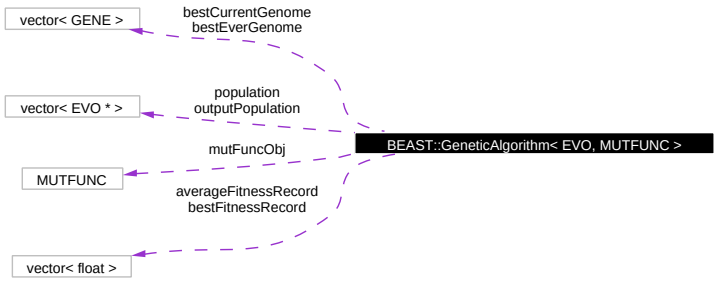
- **geneticalgorithm.h**

10.33 BEAST::GeneticAlgorithm< EVO, MUTFUNC > Class Template Reference

The **GeneticAlgorithm** class provides functionality to cover a range of GA methods, and may be extended to incorporate other approaches.

`#include <geneticalgorithm.h>`

Collaboration diagram for BEAST::GeneticAlgorithm< EVO, MUTFUNC >:



Public Types

- typedef EVO::gene_type **GENE**

  A typedef for the gene type (e.g. float), adapted from EVO.

- typedef EVO::genotype_type **GENOTYPE**

  A typedef for the genotype type (e.g. vector<float>), adapted from EVO.

- typedef std::vector< EVO * >::iterator **PopIter**

  An iterator type for the population.

- typedef std::vector< EVO * >::const_iterator **PopConstIter**

  A const iterator type for the population.

- typedef std::vector< **GENE** >::iterator **ChromIter**

  An iterator type for the chromosomes.

Public Member Functions

- **GeneticAlgorithm** (float crossover=0.7f, float mutation=0.01f, int popSize=0)

  Constructor: sets default parameters: elitism and subelitism 0, crossover points 1, tournament parameter 0.75, tournament size 2, rank pressure 1.5, exponent 1.0.

- void **SetPopulation** (const std::vector< EVO * > &p)
- std::vector< EVO * > & **GetPopulation** ()
- std::vector< EVO * > **GetPopulationCopy** () const

  Returns duplicates of the output population which won't be deleted by the GA if it does a **CleanUp()**.

- **PopIter begin** ()
- **PopIter end** ()
- void **SetCrossover** (float c)

  Sets the rate of crossover.

- void **SetMutation** (float m)

  Sets the rate of mutation.

- void **SetSelection** (**GASelectionType** s)

  Sets the selection operator.

- void **SetMutationFunction** (**GENE**(*ptr)(**GENE**))

  Sets the mutation function, which may be any C function which takes as its argument one variable of the gene type and returns a variable of the gene type.

- void **SetMutationFunction** (MUTFUNC obj)

  Sets the mutation function to a function object, the type of which should also be specified in the template constructor for the GA.

- void **SetElitism** (int e)

  Sets the elitism value, which decides how many of the fittest individuals go through to the next generation unchanged.

- void **SetSubelitism** (int s)

  Sets the subelitism value, which decides how many of the least fit individuals are barred from reproducing.

- void **SetCrossoverPoints** (int p)

  Sets the number of points of crossover.

- void **SetFitnessMethod** (**GAFitnessMethodType** f)

  Sets the method by which fitness scores are used.

- void **SetFitnessFix** (**GAFitnessFixType** f)

  Sets the method by which fitness scores < 0 are treated.

- void **SetOwnsData** (bool b)

  Set to true if the GA is responsible for deleting the old population objects.

- void **SetParameter** (**GAFltParamType** p, float f)

  Sets a real-valued parameter.

- void **SetParameter** (**GAIntParamType** p, int n)

  Sets an integer-valued parameter.

- void **SetPrintStyle** (int p)

  Sets the output style of the GA's ToString method.

- virtual void **Generate** ()

The generation function: this is where it all happens.

- void **ReGenerate** ()

  Calls the **Generate()** *method and copies the output population into the input.*

- int **GetGenerations** () const
- std::vector< float > **GetAvgFitnessHistory** () const
- std::vector< float > **GetBestFitnessHistory** () const
- std::vector< **GENE** > **GetBestCurrentGenome** () const
- std::vector< **GENE** > **GetBestEverGenome** () const
- float **GetBestCurrentFitness** () const
- float **GetBestEverFitness** ()
- std::string **ToString** () const

  Returns a string containing various details about the GA's current state, depending on what has been set with SetPrintStyle.

- std::string **GetCSV** (char separator= ',') const

  Returns a string containing a simple CSV table with average and best fitness for every generation so far.

- void **CleanUp** ()

  Deletes the objects comprising the input and output populations.

- void **Unserialise** (std::istream &)
- void **Serialise** (std::ostream &) const

Protected Member Functions

- void **CalcStats** ()

  Calculates some statistics used in some of the selection methods and also stored by the class for data collection.

- void **Setup** ()

  Prepares the GA for the next epoch.

- void **FixFitness** ()

  Adjusts the fitness according to.

- float **GetFitness** (EVO *i)

  Calculates the fitness score to be used by the GA from the stored fitness scores in the EVO object.

- void **SelectParentGenotype** (**GENOTYPE** &)

  Depending on the selection procedure chosen, a genotype is taken from the population and returned by reference.

- void **CrossoverGenotypes** (**GENOTYPE** &, **GENOTYPE** &)

  This method simply takes two chromosomes, mum and dad, and swaps them over at a random point along the length.

- void **MutateGenotype** (**GENOTYPE** &)

While crossover simulates the effect of sexual reproduction within a population, mutation artificially reproduces the effects of transcription errors in the replication of DNA.

- void **SelectProbability** (**GENOTYPE** &)

  Roulette and Rank Selection Having done the Setup function (above), each individual has a probability score, derived from their rank or fitness.

- void **SelectEven** (**GENOTYPE** &)
- void **SelectTournament** (**GENOTYPE** &)

  Tournament Selection The method implemented here is an amalgamation of two slightly different approaches.

10.33.1 Detailed Description

template<class EVO, class MUTFUNC = MutationOperator<typename EVO::gene_type>> class BEAST::GeneticAlgorithm< EVO, MUTFUNC >

The **GeneticAlgorithm** class provides functionality to cover a range of GA methods, and may be extended to incorporate other approaches.

The class is completely generic, in order to allow the widest possible application. The first template parameter is the type of the genes (e.g. int, float). The second is a class which must provide the methods exposed by the

Evolver ABC, ideally inherited from an **Evolver** of the same templated type. The third template parameter is the type of the mutation operator and will usually be fine as the default type of MutationOperator<T>, which is a **MutationOperator** function object with the same type as the genes. Phew!

10.33.2 Member Function Documentation

10.33.2.1 template<class EVO, class MUTFUNC = MutationOperator<typename EVO::gene_type>> int BEAST::GeneticAlgorithm< EVO, MUTFUNC >::GetGenerations () const [inline]

Returns:
The number of Generations so far

10.33.2.2 template<class EVO, class MUTFUNC = MutationOperator<typename EVO::gene_type>> void BEAST::GeneticAlgorithm< EVO, MUTFUNC >::SetCrossover (float *c*) [inline]

Sets the rate of crossover.

Parameters:
c A float in the range [0,1]

10.33.2.3 template<class EVO, class MUTFUNC = MutationOperator<typename EVO::gene_type>> void BEAST::GeneticAlgorithm< EVO, MUTFUNC >::SetCrossoverPoints (int *p*) [inline]

Sets the number of points of crossover.

Parameters:
p An integer >= 0 (0 results in no crossover)

10.33.2.4 template<class EVO, class MUTFUNC = MutationOperator<typename EVO::gene_type>> void BEAST::GeneticAlgorithm< EVO, MUTFUNC >::SetElitism (int *e*) [inline]

Sets the elitism value, which decides how many of the fittest individuals go through to the next generation unchanged.

Parameters:
e An integer in the range [0,population size]

10.33.2.5 template<class EVO, class MUTFUNC = MutationOperator<typename EVO::gene_type>> void BEAST::GeneticAlgorithm< EVO, MUTFUNC >::SetFitnessFix (GAFitnessFixType *f*) [inline]

Sets the method by which fitness scores < 0 are treated.

See also:
GAFitnessFixType

10.33.2.6 template<class EVO, class MUTFUNC = MutationOperator<typename EVO::gene_type>> void BEAST::GeneticAlgorithm< EVO, MUTFUNC >::SetFitnessMethod (GAFitnessMethodType *f*) [inline]

Sets the method by which fitness scores are used.

See also:
GAFitnessMethodType

10.33.2.7 template<class EVO, class MUTFUNC = MutationOperator<typename EVO::gene_type>> void BEAST::GeneticAlgorithm< EVO, MUTFUNC >::SetMutation (float *m*) [inline]

Sets the rate of mutation.

Parameters:
m A float in the range [0,1]

10.33.2.8 template<class EVO, class MUTFUNC = MutationOperator<typename EVO::gene_type>> void BEAST::GeneticAlgorithm< EVO, MUTFUNC >::SetParameter (GAIntParamType *p*, int *n*) [inline]

Sets an integer-valued parameter.

Parameters:

p The parameter to set

n The value to set it to.

See also:

GAIntParamType

10.33.2.9 template<class EVO, class MUTFUNC = MutationOperator<typename EVO::gene_type>> void BEAST::GeneticAlgorithm< EVO, MUTFUNC >::SetParameter (GAFltParamType *p*, float *f*) [inline]

Sets a real-valued parameter.

Parameters:

p The parameter to set

f The value to set it to.

See also:

GAFltParamType

10.33.2.10 template<class EVO, class MUTFUNC = MutationOperator<typename EVO::gene_type>> void BEAST::GeneticAlgorithm< EVO, MUTFUNC >::SetPrintStyle (int *p*) [inline]

Sets the output style of the GA's ToString method.

See also:

GAPrintStyleType

10.33.2.11 template<class EVO, class MUTFUNC = MutationOperator<typename EVO::gene_type>> void BEAST::GeneticAlgorithm< EVO, MUTFUNC >::SetSelection (GASelectionType *s*) [inline]

Sets the selection operator.

See also:

GASelectionType

10.33.2.12 template<class EVO, class MUTFUNC = MutationOperator<typename EVO::gene_type>> void BEAST::GeneticAlgorithm< EVO, MUTFUNC >::SetSubelitism (int *s*) [inline]

Sets the subelitism value, which decides how many of the least fit individuals are barred from reproducing.

Parameters:

s An integer in the range [0,population size]

The documentation for this class was generated from the following file:

- **geneticalgorithm.h**

10.34 BEAST::GeneticAlgorithm< EVO, MUTFUNC >::evo_sort< _EVO > Struct Template Reference

A little function object to enable us to sort the population by fitness.

`#include <geneticalgorithm.h>`

Public Member Functions

- bool **operator()** (_EVO *const &p1, _EVO *const &p2)

10.34.1 Detailed Description

template<class EVO, class MUTFUNC = MutationOperator<typename EVO::gene_type>>template<class _EVO> struct BEAST::GeneticAlgorithm< EVO, MUTFUNC >::evo_sort< _EVO >

A little function object to enable us to sort the population by fitness.

The documentation for this struct was generated from the following file:

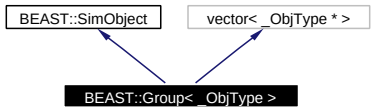
- **geneticalgorithm.h**

10.35 BEAST::Group< _ObjType > Class Template Reference

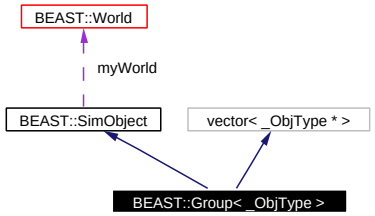
A simple class which creates and maintains a vector of objects of the specified type and adds them to the world each round.

`#include <simulation.h>`

Inheritance diagram for BEAST::Group< _ObjType >:



Collaboration diagram for BEAST::Group< _ObjType >:



Public Types

- typedef _ObjType **object_type**

Public Member Functions

- **Group** (int s=0)

  Sets the **Group** *up with a number of objects of the native type.*

- virtual ~**Group** ()

  Destructor - deletes all the objects in the group.

- virtual void **AddToWorld** ()

  Adds the contents of the **Group** *to the* **World**.

- void **ForEach** (void(_ObjType::*method)())

  Calls the specified member function on each object in the **Group**.

- template<typename _Arg> void **ForEach** (void(_ObjType::*method)(_Arg), _Arg arg)

  Calls the specified member function on each object in the group, passing an argument.

- template<typename _Result> void **ForEach** (_Result(_ObjType::*method)())

  Calls the specified member function on each object in the **Group**.

- template<typename _Result, typename _Arg> void **ForEach** (_Result(_ObjType::*method)(_Arg), _Arg arg)

  Calls the specified member function on each object in the group, passing an argument.

- virtual void **Serialise** (std::ostream &) const
- virtual void **Unserialise** (std::istream &)

10.35.1 Detailed Description

template<class _ObjType> class BEAST::Group< _ObjType >

A simple class which creates and maintains a vector of objects of the specified type and adds them to the world each round.

Note that **Group** is also responsible for deleting the objects it contains, if you attempt to delete any of the objects in a **Group** yourself, there will likely be segmentation faults.

Parameters:
_ObjType The type of objects to create.

See also:
Population
SimObject

10.35.2 Constructor & Destructor Documentation

10.35.2.1 template<class _ObjType> virtual BEAST::Group< _ObjType >::~Group () [inline, virtual]

Destructor - deletes all the objects in the group.

10.35.3 Member Function Documentation

10.35.3.1 template<class _ObjType> template<typename _Result, typename _Arg> void BEAST::Group< _ObjType >::ForEach (_Result(_ObjType::*method)(_Arg), _Arg arg) [inline]

Calls the specified member function on each object in the group, passing an argument.

The member function may be one returning a result, but the result will be discarded.

10.35.3.2 template<class _ObjType> template<typename _Result> void BEAST::Group< _ObjType >::ForEach (_Result(_ObjType::* method)()) [inline]

Calls the specified member function on each object in the **Group**.

The member function may be one returning a result, but the result will be discarded.

10.35.3.3 template<class _ObjType> template<typename _Arg> void BEAST::Group< _ObjType >::ForEach (void(_ObjType::* *method*)(_Arg), _Arg *arg*) [inline]

Calls the specified member function on each object in the group, passing an argument.

10.35.3.4 template<class _ObjType> void BEAST::Group< _ObjType >::ForEach (void(_ObjType::* *method*)()) [inline]

Calls the specified member function on each object in the **Group**.

The documentation for this class was generated from the following file:

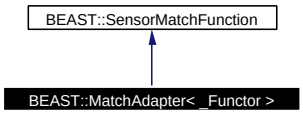
- **simulation.h**

10.36 BEAST::MatchAdapter< _Functor > Struct Template Reference

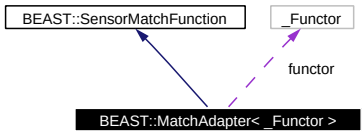
Allows any unary predicate to be adapted for use as a matching function.

`#include <sensorfunctors.h>`

Inheritance diagram for BEAST::MatchAdapter< _Functor >:



Collaboration diagram for BEAST::MatchAdapter< _Functor >:



Public Member Functions

- **MatchAdapter** (_Functor f)
- virtual bool **operator()** (**WorldObject** *input)

Public Attributes

- _Functor **functor**

10.36.1 Detailed Description

template<class _Functor> struct BEAST::MatchAdapter< _Functor >

Allows any unary predicate to be adapted for use as a matching function.

This includes all unary functors described by the STL and any binary functors after adaptation with bind1st or bind2nd. The ptr_fun adapter may be used to turn any function into a scaling functor.

The documentation for this struct was generated from the following file:

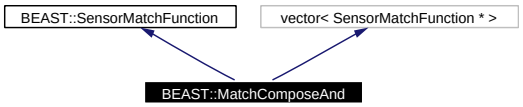
- sensorfunctors.h

10.37 BEAST::MatchComposeAnd Struct Reference

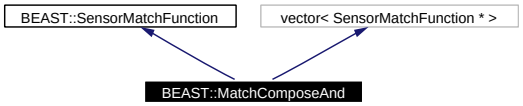
Chains any number of matching functions together such that only if all of them are true for the object being matched, **MatchComposeAnd** will return true.

`#include <sensorfunctors.h>`

Inheritance diagram for BEAST::MatchComposeAnd:



Collaboration diagram for BEAST::MatchComposeAnd:



Public Member Functions

- **MatchComposeAnd** (**SensorMatchFunction** *first, **SensorMatchFunction** *second)
- virtual bool **operator()** (**WorldObject** *obj)

10.37.1 Detailed Description

Chains any number of matching functions together such that only if all of them are true for the object being matched, **MatchComposeAnd** will return true.

The first two functors may be added in the constructor. If more are needed, they may be added using push_back. Because **MatchComposeAnd** is derived from std::vector, its contents may be manipulated just like any other vector.

The documentation for this struct was generated from the following file:

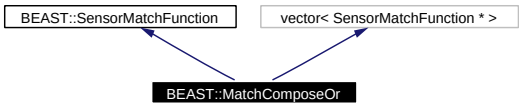
- sensorfunctors.h

10.38 BEAST::MatchComposeOr Struct Reference

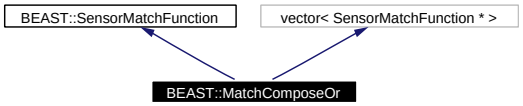
Chains any number of matching functions together such that should any of them be true for the object being matched, **MatchComposeOr** will return true.

`#include <sensorfunctors.h>`

Inheritance diagram for BEAST::MatchComposeOr:



Collaboration diagram for BEAST::MatchComposeOr:



Public Member Functions

- **MatchComposeOr** (**SensorMatchFunction** *first, **SensorMatchFunction** *second)
- virtual bool **operator()** (**WorldObject** *obj)

10.38.1 Detailed Description

Chains any number of matching functions together such that should any of them be true for the object being matched, **MatchComposeOr** will return true.

The first two functors may be added in the constructor. If more are needed, they may be added using push_back. Because **MatchComposeOr** is derived from std::vector, its contents may be manipulated just like any other vector.

The documentation for this struct was generated from the following file:

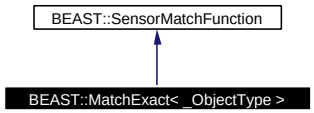
- sensorfunctors.h

10.39 BEAST::MatchExact< _ObjectType > Struct Template Reference

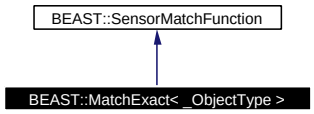
Identifies exact object types, so if defined with Cheese, will return true only for Cheese, and false for Cheddar and Gruyère.

`#include <sensorfunctors.h>`

Inheritance diagram for BEAST::MatchExact< _ObjectType >:



Collaboration diagram for BEAST::MatchExact< _ObjectType >:



Public Member Functions

- virtual bool **operator()** (**WorldObject** *obj)

10.39.1 Detailed Description

template<class _ObjectType> struct BEAST::MatchExact< _ObjectType >

Identifies exact object types, so if defined with Cheese, will return true only for Cheese, and false for Cheddar and Gruyère.

The documentation for this struct was generated from the following file:

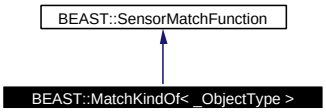
- sensorfunctors.h

10.40 BEAST::MatchKindOf< _ObjectType > Struct Template Reference

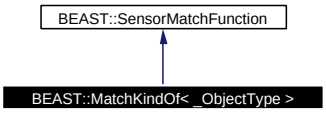
Identifies objects belonging to hierarchies, so if defined with Cheese, will return true for objects of type Cheese, or derived classes such as Cheddar and Gruyère.

`#include <sensorfunctors.h>`

Inheritance diagram for BEAST::MatchKindOf< _ObjectType >:



Collaboration diagram for BEAST::MatchKindOf< _ObjectType >:



Public Member Functions

- virtual bool **operator()** (**WorldObject** *obj)

10.40.1 Detailed Description

template<class _ObjectType> struct BEAST::MatchKindOf< _ObjectType >

Identifies objects belonging to hierarchies, so if defined with Cheese, will return true for objects of type Cheese, or derived classes such as Cheddar and Gruyère.

The documentation for this struct was generated from the following file:

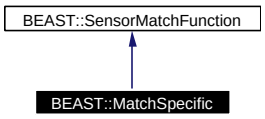
- sensorfunctors.h

10.41 BEAST::MatchSpecific Struct Reference

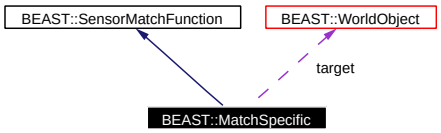
Identifies one particular object and returns true only for that object.

`#include <sensorfunctors.h>`

Inheritance diagram for BEAST::MatchSpecific:



Collaboration diagram for BEAST::MatchSpecific:



Public Member Functions

- **MatchSpecific** (**WorldObject** *obj)
- virtual bool **operator()** (**WorldObject** *obj)

Public Attributes

- **WorldObject** * **target**

10.41.1 Detailed Description

Identifies one particular object and returns true only for that object.

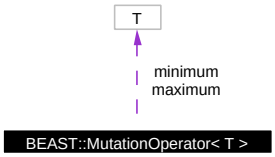
The documentation for this struct was generated from the following file:

- sensorfunctors.h

10.42 BEAST::MutationOperator< T > Struct Template Reference

`#include <geneticalgorithm.h>`

Collaboration diagram for BEAST::MutationOperator< T >:



Public Member Functions

- **MutationOperator** (T min=-1.0, T max=1.0)

  The maximum and minimum random values can be optionally specified.

- T **operator()** (T t)

  Performs a random mutation on the argument and returns the result.

Public Attributes

- T **minimum**

  Minimum mutation.

- T **maximum**

  Maximum mutation.

10.42.1 Detailed Description

template<typename T> struct BEAST::MutationOperator< T >

A functor which may be initialised with max and min values, and then returns a uniformly distributed random number between those values. **MutationOperator** may be adapted in a number of ways to suit different GA requirements:

The template can be initialised as any numeric type and will return good results for real values, usable results for integer types (remember to add one to the max limit to take account of the lack of proper rounding.)

The template can be specialised to provide an alternative default mutation operator for whichever type you are using for your genotype, e.g. you might want to make a gaussian distributed float mutation operator.

Alternatively you could do away with **MutationOperator** entirely and specify a different functor type when initialising your GA (the GA defaults to a mutation operator of a type matching the GA's gene type)

The documentation for this struct was generated from the following file:

- **geneticalgorithm.h**

10.43 BEAST::MutationOperator< bool > Struct Template Reference

Specialised **MutationOperator** for bool, simply NOT's its input.

`#include <geneticalgorithm.h>`

Public Member Functions

- **MutationOperator** ()
- **MutationOperator** (bool, bool)
- bool **operator()** (bool b)

10.43.1 Detailed Description

template<> struct BEAST::MutationOperator< bool >

Specialised **MutationOperator** for bool, simply NOT's its input.

This is mostly for a demonstration, although bool would be an easy way to implement binary genotypes, you would be better advised to specialise the int type of **MutationOperator** since the native integer type is faster on most systems.

The documentation for this struct was generated from the following file:

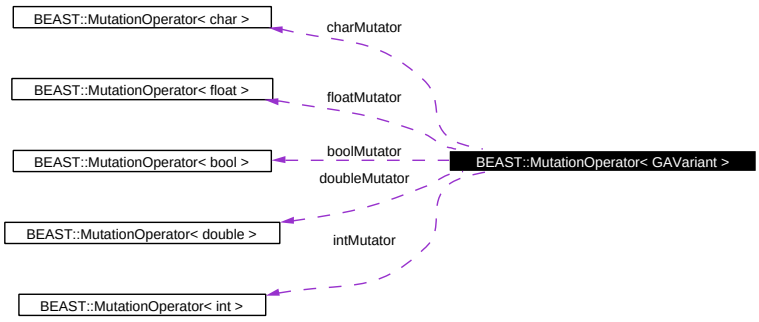
- **geneticalgorithm.h**

10.44 BEAST::MutationOperator< GAVariant > Struct Template Reference

This specialised mutation operator provides the facilities of the basic **MutationOperator** for **GAVariant**.

`#include <geneticalgorithm.h>`

Collaboration diagram for BEAST::MutationOperator< GAVariant >:



Public Member Functions

- **MutationOperator** (**GAVariant** min=-1.0, **GAVariant** max=1.0)

  Constructor, casts inputs to different types for sub-operators.

- **GAVariant operator()** (**GAVariant** in)

  Calls the mutation operator for the correct type.

Public Attributes

- **MutationOperator**< int > **intMutator**

  Mutation operator for ints.

- **MutationOperator**< float > **floatMutator**

  Mutation operator for floats.

- **MutationOperator**< double > **doubleMutator**

  Mutation operator for doubles.

- **MutationOperator**< char > **charMutator**

  Mutation operator for chars.

- **MutationOperator**< bool > **boolMutator**

  Mutation operator for bools.

10.44.1 Detailed Description

template<> struct BEAST::MutationOperator< GAVariant >

This specialised mutation operator provides the facilities of the basic **MutationOperator** for **GAVariant**.

Five separate **MutationOperator** objects are created, one for each data type (int, float, double, char, bool). These mutation operators are initialised with the same min and max mutations but can be accessed individually and reassigned via the public members which store them.

Todo
serialise/unserialise

The documentation for this struct was generated from the following file:

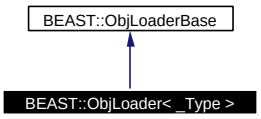
- **geneticalgorithm.h**

10.45 BEAST::ObjLoader< _Type > Struct Template Reference

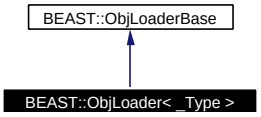
A functor for recreating templated object types using serialisation.

`#include <unserialiser.h>`

Inheritance diagram for BEAST::ObjLoader< _Type >:



Collaboration diagram for BEAST::ObjLoader< _Type >:



Public Member Functions

- virtual **WorldObject** * **operator()** (std::istream &in)

10.45.1 Detailed Description

template<typename _Type> struct BEAST::ObjLoader< _Type >

A functor for recreating templated object types using serialisation.

The documentation for this struct was generated from the following file:

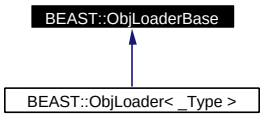
- **unserialiser.h**

10.46 BEAST::ObjLoaderBase Struct Reference

A simple abstract base class for **ObjLoader** functors.

`#include <unserialiser.h>`

Inheritance diagram for BEAST::ObjLoaderBase:



Public Member Functions

- virtual **WorldObject** * **operator()** (std::istream &in)=0

10.46.1 Detailed Description

A simple abstract base class for **ObjLoader** functors.

The documentation for this struct was generated from the following file:

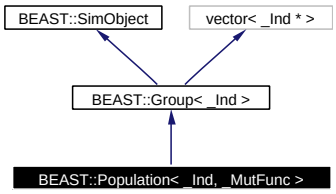
- **unserialiser.h**

10.47 BEAST::Population< _Ind, _MutFunc > Class Template Reference

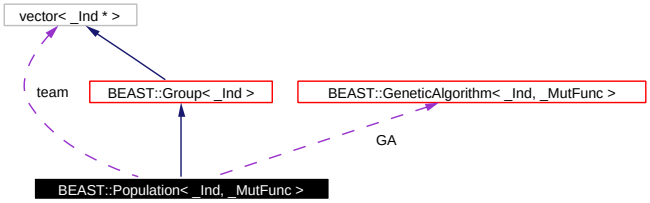
This class is derived from **Group** and adds a managed GA which is automatically run on the whole **Population** every epoch.

`#include <population.h>`

Inheritance diagram for BEAST::Population< _Ind, _MutFunc >:



Collaboration diagram for BEAST::Population< _Ind, _MutFunc >:



Public Member Functions

- **Population** (int s, **GeneticAlgorithm**< _Ind, _MutFunc > &ga)
- virtual void **BeginAssessment** ()

  This method is called at the beginning of the assessment and sets up teams if required.

- virtual void **EndAssessment** ()

  This method is called at the end of the assessment and causes each individual's fitness score to be stored.

- virtual void **BeginGeneration** ()

  This method is called at the beginning of the generation and.

- virtual void **EndGeneration** ()

  Performs GA management, by copying the **Population** *into the GA, running the GA on the population and then retrieving the new generation from the GA.*

- virtual void **BeginRun** ()

  This method is called at the beginning of the run and ensures that the contents of the population has been reset.

- virtual void **EndRun** ()

This method is called at the end of the run and currently doesn't do anything at all.

- virtual void **AddToWorld** ()

  Adds either the whole population, or the currently selected team to the world.

- void **SetTeamSize** (int n)

  Decides how many individuals will go into assessments.

- void **SetClones** (int n)

  Allows you to specify the number of clones to be made of each individual in each assessment.

- const std::vector< _Ind * > & **GetTeam** () const

  Returns the current team.

- virtual std::string **ToString** () const

  Outputs data about the current generation, currently only from the GA.

- virtual void **Serialise** (std::ostream &) const

  Copies the **Population** *to a stream.*

- virtual void **Unserialise** (std::istream &)

  Converts the data produced by Serialise back into a population.

Public Attributes

- **GeneticAlgorithm**< _Ind, _MutFunc > & **GA**

10.47.1 Detailed Description

template<class _Ind, class _MutFunc = MutationOperator<typename _Ind::gene_type>> class BEAST::Population< _Ind, _MutFunc >

This class is derived from **Group** and adds a managed GA which is automatically run on the whole **Population** every epoch.

Parameters:

_Ind The type of objects to create, must inherit from **Evolver**.

_MutFunc If your GA uses a custom mutation operator, it must also be specified here.

See also:

GeneticAlgorithm
Evolver
Group

10.47.2 Member Function Documentation

10.47.2.1 template<class _Ind, class _MutFunc = MutationOperator<typename _Ind::gene_type>> void BEAST::Population< _Ind, _MutFunc >::SetClones (int *n*) [inline]

Allows you to specify the number of clones to be made of each individual in each assessment.

If you want to assess clones of only one individual, set the team size to one.

The documentation for this class was generated from the following file:

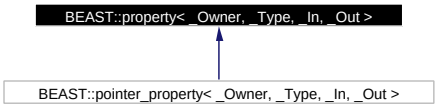
- population.h

10.48 BEAST::property< _Owner, _Type, _In, _Out > Class Template Reference

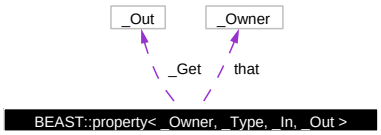
Class wrapper for a member variable which allows member data to be exposed with invisible get/set semantics.

`#include <utilities.h>`

Inheritance diagram for BEAST::property< _Owner, _Type, _In, _Out >:



Collaboration diagram for BEAST::property< _Owner, _Type, _In, _Out >:



Public Member Functions

- **property** (_Owner *t, _Out(_Owner::*get)(void) const, void(_Owner::*set)(_In))
- void **init** (_Owner *t, _Out(_Owner::*get)(void) const, void(_Owner::*set)(_In))
- **property** & **operator=** (_In v)
- **operator _Out** ()
- _Type * **operator** → () const
- template<typename _CastType> _CastType **as** () const

10.48.1 Detailed Description

template<class _Owner, typename _Type, typename _In = _Type, typename _Out = _In> class BEAST::property< _Owner, _Type, _In, _Out >

Class wrapper for a member variable which allows member data to be exposed with invisible get/set semantics.

Note:

A slight speed penalty is incurred by the time taken to dereference the accessor/mutator functions, if faster access is required the actual accessor functions should be used.

The documentation for this class was generated from the following file:

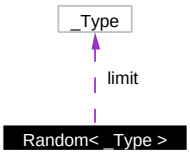
- **utilities.h**

10.49 Random< _Type > Struct Template Reference

Function object version of randval.

`#include <random.h>`

Collaboration diagram for Random< _Type >:



Public Member Functions

- **Random** (_Type l)
- _Type **operator()** (_Type n)
- _Type **operator()** ()

Public Attributes

- _Type **limit**

10.49.1 Detailed Description

template<typename _Type> struct Random< _Type >

Function object version of randval.

See also:
randval

The documentation for this struct was generated from the following file:

- **random.h**

10.50 BEAST::Ring2D Struct Reference

Plots a two dimensional ring.

`#include <bacteria.h>`

Public Member Functions

- **Ring2D** (int centerx, int centery, double OuterRadius=2.0, double InnerRadius=1.0, double scale=1.0)

  Plots a two dimensional ring.

- double **operator()** (int X, int Y)

Public Attributes

- int **cx**
- int **cy**

  The centre of the Gaussian function.

- double **rin2**
- double **rout2**

  Store the square radii.

- double **area**
- double **s**

  Store the area and scale for speed.

10.50.1 Detailed Description

Plots a two dimensional ring.

10.50.2 Constructor & Destructor Documentation

10.50.2.1 BEAST::Ring2D::Ring2D (int *centerx*, int *centery*, double *OuterRadius* = 2.0, double *InnerRadius* = 1.0, double *scale* = 1.0) [inline]

Plots a two dimensional ring.

Parameters:

centerx The X coordinate of the center.

centery The Y coordinate of the center.

OuterRadius The maximum radius of the ring.

InnerRadius The minimum radius of the ring (radius of the hole).

scale The value to output within the ring area.

The documentation for this struct was generated from the following file:

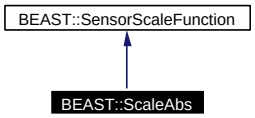
- **bacteria.h**

10.51 BEAST::ScaleAbs Struct Reference

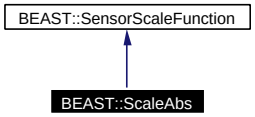
Returns the absolute value of the input, as for the std::abs function.

`#include <sensorfunctors.h>`

Inheritance diagram for BEAST::ScaleAbs:



Collaboration diagram for BEAST::ScaleAbs:



Public Member Functions

- virtual double **operator()** (double input)

10.51.1 Detailed Description

Returns the absolute value of the input, as for the std::abs function.

The documentation for this struct was generated from the following file:

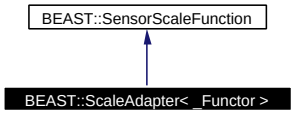
- sensorfunctors.h

10.52 BEAST::ScaleAdapter< _Functor > Struct Template Reference

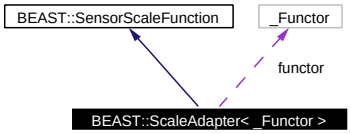
Allows any unary functor to be adapted for use as a scaling function.

`#include <sensorfunctors.h>`

Inheritance diagram for BEAST::ScaleAdapter< _Functor >:



Collaboration diagram for BEAST::ScaleAdapter< _Functor >:



Public Member Functions

- **ScaleAdapter** (_Functor f)
- virtual double **operator()** (double input)

Public Attributes

- _Functor **functor**

Related Functions

(Note that these are not member functions.)

- **ScaleAdapter**< _Functor > * **ScaleAdapt** (_Functor f)
  A helper function for creating **ScaleAdapter** *functors.*

10.52.1 Detailed Description

template<class _Functor> struct BEAST::ScaleAdapter< _Functor >

Allows any unary functor to be adapted for use as a scaling function.

This includes all unary functors described by the STL and any binary functors after adaptation with bind1st or bind2nd. The ptr_fun adapter may be used to turn any function into a scaling functor.

The documentation for this struct was generated from the following file:

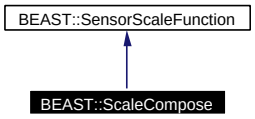
- sensorfunctors.h

10.53 BEAST::ScaleCompose Struct Reference

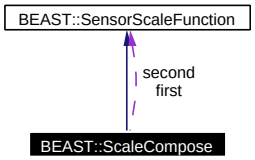
ScaleCompose allows the chaining of two scaling functions together, such the output of a **ScaleCompose** functor is the result of second(first(input)), where first and second are the arguments in ScaleCompose's constructor.

`#include <sensorfunctors.h>`

Inheritance diagram for BEAST::ScaleCompose:



Collaboration diagram for BEAST::ScaleCompose:



Public Member Functions

- **ScaleCompose** (**SensorScaleFunction** *f, **SensorScaleFunction** *s)
- virtual double **operator()** (double input)

10.53.1 Detailed Description

ScaleCompose allows the chaining of two scaling functions together, such the output of a **ScaleCompose** functor is the result of second(first(input)), where first and second are the arguments in ScaleCompose's constructor.

For example, to create a function which scales the input from [0:50] to [0:1] and then adds random noise between -0.5 and +0.5: s->SetScalingFunction(new **ScaleCompose**(new **ScaleLinear**(50.0), new **ScaleNoise**(-0.5, 0.5))); To compose more complex functions, instances of **ScaleCompose** may be nested. **ScaleCompose** is responsible for deleting its child functions.

The documentation for this struct was generated from the following file:

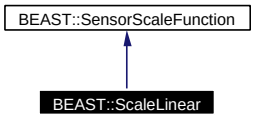
- sensorfunctors.h

10.54 BEAST::ScaleLinear Struct Reference

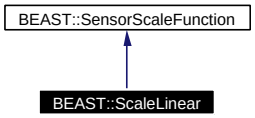
A simple linear scaling function which defaults to an input scale between 0 and a defined maximum, scaling to an output range between 0 and 1.

`#include <sensorfunctors.h>`

Inheritance diagram for BEAST::ScaleLinear:



Collaboration diagram for BEAST::ScaleLinear:



Public Member Functions

- **ScaleLinear** (double range)
- **ScaleLinear** (double inputMinimum, double inputMaximum, double outputMinimum, double outputMaximum)
- virtual double **operator()** (double input)

Public Attributes

- double **inMin**
- double **inMax**
- double **outMin**
- double **outMax**

10.54.1 Detailed Description

A simple linear scaling function which defaults to an input scale between 0 and a defined maximum, scaling to an output range between 0 and 1.

Any input and output range can be defined, including inverted ranges with min > max, which can invert the output.

The documentation for this struct was generated from the following file:

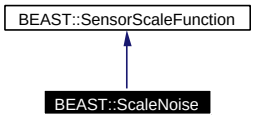
- sensorfunctors.h

10.55 BEAST::ScaleNoise Struct Reference

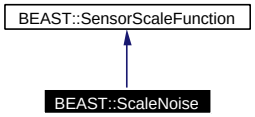
ScaleNoise adds uniform random noise to its input.

`#include <sensorfunctors.h>`

Inheritance diagram for BEAST::ScaleNoise:



Collaboration diagram for BEAST::ScaleNoise:



Public Member Functions

- **ScaleNoise** (double min=-0.1, double max=0.1)
- virtual double **operator()** (double input)

Public Attributes

- double **minimum**
- double **maximum**

10.55.1 Detailed Description

ScaleNoise adds uniform random noise to its input.

Minimum and maximum values may be specified, but default to [-0.1:0.1]

The documentation for this struct was generated from the following file:

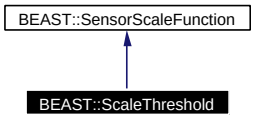
- sensorfunctors.h

10.56 BEAST::ScaleThreshold Struct Reference

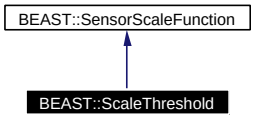
ScaleThreshold takes values: threshold, min and max and returns min if input < threshold, or max if input >= threshold.

`#include <sensorfunctors.h>`

Inheritance diagram for BEAST::ScaleThreshold:



Collaboration diagram for BEAST::ScaleThreshold:



Public Member Functions

- **ScaleThreshold** (double t, double a=0.0, double b=1.0)
- virtual double **operator()** (double input)

Public Attributes

- double **threshold**
- double **minimum**
- double **maximum**

10.56.1 Detailed Description

ScaleThreshold takes values: threshold, min and max and returns min if input < threshold, or max if input >= threshold.

Min and max default to 0 and 1.

The documentation for this struct was generated from the following file:

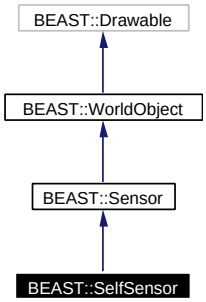
- sensorfunctors.h

10.57 BEAST::SelfSensor Class Reference

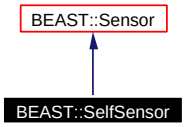
The **SelfSensor** is used to detect information about its owner.

`#include <sensorbase.h>`

Inheritance diagram for BEAST::SelfSensor:



Collaboration diagram for BEAST::SelfSensor:



Public Member Functions

- **SelfSensor** (**SelfSensorType** t, std::string ctrl="")
  Constructor, specifying type and optionally a control to watch.
- virtual void **Update** ()
  Resets the sensor ready for the next round of tests.
- virtual void **Interact** (**WorldObject** *)
  Calls the sensor's matching function on the **WorldObject**, *then if it's a match, calls the evaluation function.*
- virtual double **GetOutput** () const
  Returns the SelfSensor's output, which comes directly from the sensor's owner.

Protected Attributes

- **SelfSensorType myType**
  The type of **SelfSensor**.
- std::string **controlName**
  From **Animat** *controls , e.g. "left".*

10.57.1 Detailed Description

The **SelfSensor** is used to detect information about its owner.

An **Animat** can use a **SelfSensor** to get information on its location and the state of its controls.

10.57.2 Member Function Documentation

10.57.2.1 double BEAST::SelfSensor::GetOutput () const [virtual]

Returns the SelfSensor's output, which comes directly from the sensor's owner.

Currently supports sensing of x/y location, angle, or control output.

Todo

Sensing of fitness, perhaps adjusting **SelfSensor** into a template which can be made to sense any detail about the owner.

Reimplemented from **BEAST::Sensor** (p. 184).

The documentation for this class was generated from the following files:

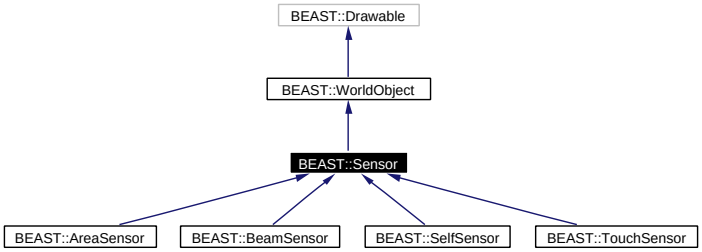
- **sensorbase.h**
- sensor.cc

10.58 BEAST::Sensor Class Reference

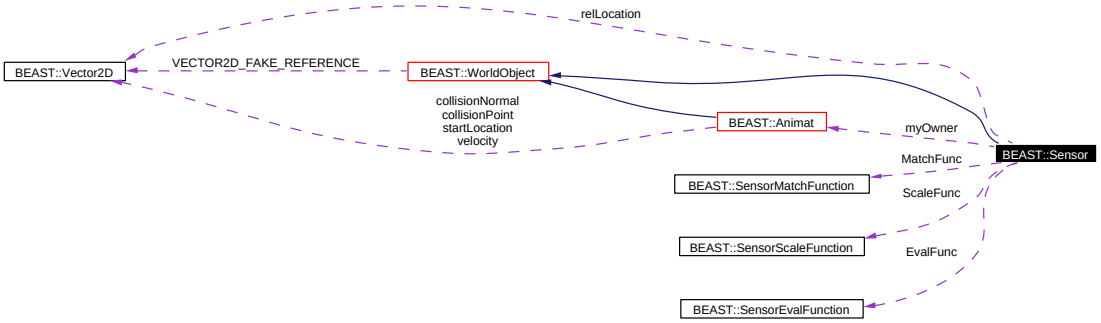
The **Sensor** class is the base class for all the different types of sensor: **TouchSensor**, **SelfSensor**, **AreaSensor** and **BeamSensor**.

`#include <sensorbase.h>`

Inheritance diagram for BEAST::Sensor:



Collaboration diagram for BEAST::Sensor:



Public Member Functions

- **Sensor** (**Vector2D** l=**Vector2D**(0.0, 0.0), double o=0.0)

  Constructor for the basic **Sensor** *class.*

- virtual ~**Sensor** ()

  Destructor: deletes the sensor's functors, if present.

- virtual void **Init** ()

  Initialises **WorldObject** *by calculating edges if required, and setting a random location if initRandom is set.*

- virtual void **Update** ()

  Resets the sensor ready for the next round of tests.

- virtual void **Interact** (**WorldObject** *)

  Calls the sensor's matching function on the **WorldObject**, *then if it's a match, calls the evaluation function.*

- virtual void **Display** ()
- virtual double **GetOutput** () const

  Returns the sensor's output for this round.

- void **SetOwner** (**Animat** *owner)
- **Animat** * **GetOwner** () const
- void **SetMatchingFunction** (**SensorMatchFunction** *func)

  Sets the matching function, deleting the old one if appropriate.

- void **SetEvaluationFunction** (**SensorEvalFunction** *func)

  Sets the evaluation function, deleting the old one if appropriate.

- void **SetScalingFunction** (**SensorScaleFunction** *func)

  Sets the scaling function, deleting the old one if appropriate.

Protected Attributes

- **Animat** * **myOwner**

  pointer to owner **Animat**, *NULL if none*

- **Vector2D relLocation**

  relative location to **Animat**

- double **relOrientation**

  relative orientation to **Animat**

- **SensorMatchFunction** * **MatchFunc**
- **SensorEvalFunction** * **EvalFunc**
- **SensorScaleFunction** * **ScaleFunc**

10.58.1 Detailed Description

The **Sensor** class is the base class for all the different types of sensor: **TouchSensor**, **SelfSensor**, **AreaSensor** and **BeamSensor**.

The basic **Sensor** class imposes no area restriction and so will allow its owner to detect any object in the world.

10.58.2 Constructor & Destructor Documentation

10.58.2.1 BEAST::Sensor::~Sensor () [virtual]

Destructor: deletes the sensor's functors, if present.

Todo

Shared functors?

10.58.3 Member Function Documentation

10.58.3.1 double BEAST::Sensor::GetOutput () const [virtual]

Returns the sensor's output for this round.

The scaling function is applied to the output of the evaluation function to get the result, thus the output might be divided by the range to yield a value [0:1], or randomly adjusted to simulate noise.

Reimplemented in **BEAST::SelfSensor** (p. 181).

The documentation for this class was generated from the following files:

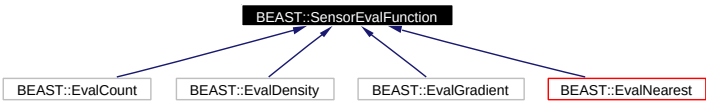
- **sensorbase.h**
- sensor.cc

10.59 BEAST::SensorEvalFunction Struct Reference

Abstract base class for evaluation functors.

`#include <sensorbase.h>`

Inheritance diagram for BEAST::SensorEvalFunction:



Public Member Functions

- virtual void **Reset** ()
- virtual void **operator()** (**WorldObject** *, const **Vector2D** &nearestPoint)=0
- virtual double **GetOutput** () const=0

10.59.1 Detailed Description

Abstract base class for evaluation functors.

Classes inherited from **SensorEvalFunction** must implement operator()(WorldObject*, const Vector2d&) and double GetOutput()const.

The documentation for this struct was generated from the following file:

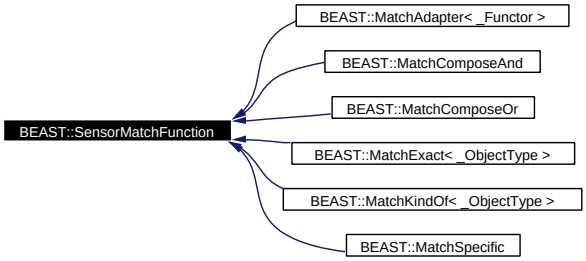
- **sensorbase.h**

10.60 BEAST::SensorMatchFunction Struct Reference

Abstract base class for matching functors.

`#include <sensorbase.h>`

Inheritance diagram for BEAST::SensorMatchFunction:



Public Member Functions

- virtual void **Reset** ()
- virtual bool **operator()** (**WorldObject** *)=0

10.60.1 Detailed Description

Abstract base class for matching functors.

Classes inherited from **SensorMatchFunction** must implement operator()(WorldObject*).

The documentation for this struct was generated from the following file:

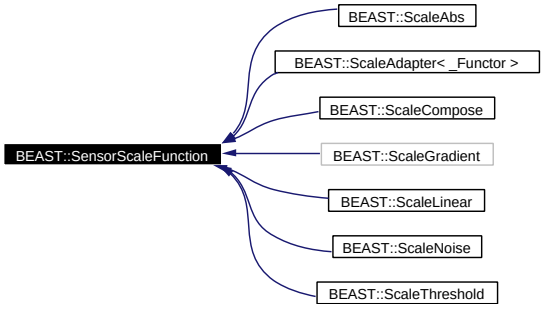
- **sensorbase.h**

10.61 BEAST::SensorScaleFunction Struct Reference

Abstract base class for scaling functors.

`#include <sensorbase.h>`

Inheritance diagram for BEAST::SensorScaleFunction:



Public Member Functions

- virtual void **Reset** ()
- virtual double **operator()** (double)=0

10.61.1 Detailed Description

Abstract base class for scaling functors.

Classes inherited from **SensorScaleFunction** must implement double operator()(double).

The documentation for this struct was generated from the following file:

- **sensorbase.h**

10.62 BEAST::SerialException Struct Reference

Since exceptions have an undesirable overhead, they have not been used elsewhere in the simulation environment for reasons of speed.

`#include <serialfuncs.h>`

Public Member Functions

- **SerialException** (**SerialErrorType** e, std::string n="", std::string msg="")
- std::string **ToString** () const

  Returns a textual description of the exception.

Public Attributes

- **SerialErrorType error**
- std::string **name**
- std::string **message**

10.62.1 Detailed Description

Since exceptions have an undesirable overhead, they have not been used elsewhere in the simulation environment for reasons of speed.

Loading and saving, however, is not speed-critical and the problems encountered suit exception handling well. If in the future a more complete exception framework is added, **SerialException** should be incorporated into that.

The documentation for this struct was generated from the following files:

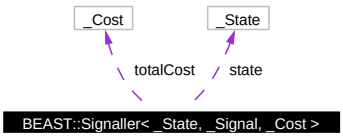
- **serialfuncs.h**
- serialfuncs.cc

10.63 BEAST::Signaller< _State, _Signal, _Cost > Class Template Reference

A general-purpose class for modelling signallers with discrete signal and state types.

`#include <signaller.h>`

Collaboration diagram for BEAST::Signaller< _State, _Signal, _Cost >:



Public Types

- typedef _State **state_type**

  A typedef for use in creating sensors.

- typedef _Signal **signal_type**

  A typedef for use in creating sensors.

- typedef _Cost **cost_type**

  A typedef for use in creating sensors.

Public Member Functions

- void **Reset** ()

  Resets the **Signaller** *by putting the total cost back to 0.*

- void **Randomise** (int numStates, int numSignals)

  Randomises the signaller so that each possible internal state has a random signal associated with it.

- void **PushCost** ()

  Adds the current signalling cost to the total signalling cost so far.

- _State **GetState** () const

  Returns the internal state of the signaller.

- _Signal **GetSignal** () const

  Returns the current signal.

- _Signal **GetSignal** (_State s) const

  Returns the signal for the specified state.

- _Cost **GetCost** () const

  Returns the current signalling cost.

- _Cost **GetTotalCost** () const

  Returns the total signalling cost so far.

- void **SetState** (_State s)

  Sets the internal state of the signaller.

- void **SetSignal** (_State st, _Signal si)

  Sets up the signals for each state.

Static Public Member Functions

- _Cost **GetCost** (_State st, _Signal si)

  Returns the signalling cost for the specified state/signal.

- void **SetCost** (_State st, _Signal si, _Cost co)

  Sets up costs associated with signalling.

10.63.1 Detailed Description

template<typename _State, typename _Signal, typename _Cost = float> class BEAST::Signaller< _State, _Signal, _Cost >

A general-purpose class for modelling signallers with discrete signal and state types.

Signallers can be made to keep track of signalling costs and are compatible with their own signal sensor functions (if multiply inherited with **WorldObject**). Each **Signaller** template maintains its own static costs map, so costs are shared between signallers with the same templated types.

The documentation for this class was generated from the following file:

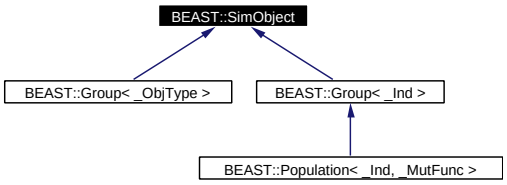
- **signaller.h**

10.64 BEAST::SimObject Class Reference

An abstract base class for the **Population** template, allowing populations with different templated types to be represented in **Simulation**.

`#include <simulation.h>`

Inheritance diagram for BEAST::SimObject:



Collaboration diagram for BEAST::SimObject:

Public Member Functions

- virtual **~SimObject** ()

  Virtual destructor.

- virtual void **BeginAssessment** ()

  This method may be overidden to do processing between assessments.

- virtual void **EndAssessment** ()
- virtual void **BeginGeneration** ()

  This method may be overidden to do processing between generations.

- virtual void **EndGeneration** ()
- virtual void **BeginRun** ()

  This method may be overidden to do processing between runs.

- virtual void **EndRun** ()
- virtual void **AddToWorld** ()=0

  Calls the world's Add method on the contents of this object.

- virtual void **Serialise** (std::ostream &) const

  Outputs object to a stream.

- virtual void **Unserialise** (std::istream &)

  Inputs object from a stream.

- virtual bool **Save** (const char *) const

  Uses the SimObject's Serialise method (selected through polymorphism) to stream the object to the specified file.

- virtual bool **Load** (const char *)

  Uses the SimObject's Unserialise method (selected through polymorphism) to reinstate the object from the specified file.

- **World** & **GetWorld** () const

  Returns the world in which the **SimObject** *resides.*

- void **SetWorld** (**World** *w)

  Sets the world used in AddToWorld.

- virtual std::string **ToString** () const

Related Functions

(Note that these are not member functions.)

- std::ostream & **operator<<** (std::ostream &out, const **SimObject** &obj)

  Output operator for all objects derived from **SimObject**.

- std::istream & **operator>>** (std::istream &in, **SimObject** &obj)

  Input operator for all objects derived from **SimObject**.

10.64.1 Detailed Description

An abstract base class for the **Population** template, allowing populations with different templated types to be represented in **Simulation**.

10.64.2 Member Function Documentation

10.64.2.1 bool BEAST::SimObject::Load (const char * *fileName*) [virtual]

Uses the SimObject's Unserialise method (selected through polymorphism) to reinstate the object from the specified file.

Returns:
True if successful.

10.64.2.2 bool BEAST::SimObject::Save (const char * *fileName*) const [virtual]

Uses the SimObject's Serialise method (selected through polymorphism) to stream the object to the specified file.

Returns:
True if successful.

The documentation for this class was generated from the following files:

- **simulation.h**
- **simulation.cc**

10.65 BEAST::Simulation Class Reference

The basic **Simulation** framework which must be derived to set up simulations.

`#include <simulation.h>`

Collaboration diagram for BEAST::Simulation:

Public Member Functions

- **Simulation** ()

  Constructor, creates an empty simulation.

- virtual void **Init** ()

  The current run is set to 0, the complete flag becomes false and each **SimObject** *is given a pointer to theWorld, then the run begins.*

- virtual bool **Update** ()

  Updates the **World** *once and checks to see if the time limit has passed.*

- void **Display** ()
- std::string **ToString** (**SimPrintStyleType** s=SIM_PRINT_STATUS) const

  Reports a few details about the current state of the simulation.

- void **Add** (std::string name, **SimObject** &pop)

  Adds a **Population**, **Group** *or other* **SimObject** *to the world.*

- void **ResetRun** ()

  This method is called when the GUI needs to break the current run and start again from the beginning of that run.

- void **ResetGeneration** ()

  This method is called when the GUI needs to break the current generation and start again from the beginning of that generation.

- void **ResetAssessment** ()

  This method is called when the GUI needs to break the current assessment and start again from the beginning of that assessment.

- void **SetRuns** (int r)

  Sets the number of runs for this simulation.

- void **SetGenerations** (int g)

  Sets the number of generations per run, if unset generations are unlimited.

- void **SetAssessments** (int a)

  Sets the number of assessments per generation, default is one.

- void **SetTimeSteps** (int t)

  Sets the number of time steps per assessment, default is 1000.

- void **SetLogStream** (std::ostream &o)

  Sets the output log stream.

- bool **HasSimObject** (std::string n)

  Checks if the specified simulation object is in the simulation.

- **SimObject** & **GetSimObject** (std::string n)

  Gets the specified simulation object.

- int **GetRun** () const

  Gets the current run.

- int **GetGeneration** () const

  Gets the current generation.

- int **GetAssessment** () const

  Gets the current assessment.

- int **GetTimeStep** () const

  Gets the current timestep.

- int **GetTotalRuns** () const

  Gets the total runs.

- int **GetTotalGenerations** () const

  Gets the total generations.

- int **GetTotalAssessments** () const

  Gets the total assessments.

- int **GetTotalTimeSteps** () const

  Gets the total time steps.

- std::ostream & **GetLogStream** ()

  Gets the log stream.

- **World** & **GetWorld** ()

  Gets the **World** *object for this simulation.*

- const std::map< std::string, **SimObject** * > & **GetContents** () const

Protected Member Functions

- void **Generate** ()
- void **TimeUp** ()
- virtual void **BeginAssessment** ()

  This method is called at the beginning of every assessment.

- virtual void **EndAssessment** ()

  This method is called at the end of every assessment.

- virtual void **BeginGeneration** ()

  This method is called at the beginning of every generation.

- virtual void **EndGeneration** ()

  This method is called at the end of every generation.

- virtual void **BeginRun** ()

  This method is called at the beginning of every run.

- virtual void **EndRun** ()

  This method is called at the end of every run and is responsible for either stopping the simulation if the maximum number of runs has been reached.

- std::map< std::string, **SimObject** * > & **GetContents** ()

10.65.1 Detailed Description

The basic **Simulation** framework which must be derived to set up simulations.

This class cannot itself be instantiated. To derive your own **Simulation**:

- Create a new class, publicly derived from **Simulation**.
- Specify your simulation contents (e.g. **Group** and **Population** classes) as member data.
- Create a public constructor which initialises the member data, then adds each object using **Simulation::Add**.
- Perform any setup of the **World**, the **Simulation** or the contents in the constructor.
- If you need to perform actions every epoch, override the virtual method, Init.

10.65.2 Member Function Documentation

10.65.2.1 void BEAST::Simulation::BeginAssessment () [protected, virtual]

This method is called at the beginning of every assessment.

If you want to override this method, ensure that **Simulation::BeginAssessment** is called at the end of your overridden version.

10.65.2.2 void BEAST::Simulation::BeginGeneration () [protected, virtual]

This method is called at the beginning of every generation.

If you want to override this method, ensure that **Simulation::BeginGeneration** is called at the end of your overridden version.

10.65.2.3 void BEAST::Simulation::BeginRun () [protected, virtual]

This method is called at the beginning of every run.

If you want to override this method, ensure that **Simulation::BeginRun** is called at the end of your overridden version.

10.65.2.4 void BEAST::Simulation::EndAssessment () [protected, virtual]

This method is called at the end of every assessment.

If you want to override this method, ensure that **Simulation::EndAssessment** is called at the end of your overridden version.

10.65.2.5 void BEAST::Simulation::EndGeneration () [protected, virtual]

This method is called at the end of every generation.

If you want to override this method, ensure that **Simulation::EndGeneration** is called at the end of your overridden version.

10.65.2.6 void BEAST::Simulation::EndRun () [protected, virtual]

This method is called at the end of every run and is responsible for either stopping the simulation if the maximum number of runs has been reached.

If you want to override this method, ensure that **Simulation::EndRun** is called at the end of your overridden version.

10.65.2.7 void BEAST::Simulation::SetRuns (int *r*) [inline]

Sets the number of runs for this simulation.

Has no effect if the number of generations has not been specified using SetGenerations.

10.65.2.8 std::string BEAST::Simulation::ToString (SimPrintStyleType *s* = SIM_PRINT_STATUS) const

Reports a few details about the current state of the simulation.

Returns:

The output string.

10.65.2.9 bool BEAST::Simulation::Update () [virtual]

Updates the **World** once and checks to see if the time limit has passed.

If the time limit has passed, EndAssessment is called.

The documentation for this class was generated from the following files:

- **simulation.h**
- **simulation.cc**

10.66 BEAST::switcher Struct Reference

The switcher is useful when configuring bools from string data.

`#include <serialfuncs.h>`

Public Member Functions

- **switcher** (std::string s, bool &b)

Public Attributes

- std::string **name**
- bool & **option**

10.66.1 Detailed Description

The switcher is useful when configuring bools from string data.

E.g. to set `bool isTrue` according to a string coming from a stream: `cin >> switcher("isTrue", isTrue);`

The documentation for this struct was generated from the following file:

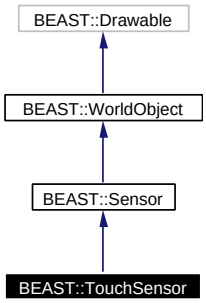
- **serialfuncs.h**

10.67 BEAST::TouchSensor Class Reference

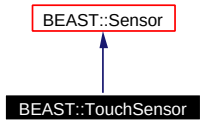
Detects objects which are touching the sensor's owner.

`#include <sensorbase.h>`

Inheritance diagram for BEAST::TouchSensor:



Collaboration diagram for BEAST::TouchSensor:



Public Member Functions

- virtual void **Init** ()

  Initialises sensor radius to match that of owner.

- virtual void **Interact** (**WorldObject** *other)

  Checks if the **WorldObject** *is the correct type using MatchFunc, then checks if the sensor's owner is touching the* **WorldObject** *and calls the EvalFunc.*

10.67.1 Detailed Description

Detects objects which are touching the sensor's owner.

See also:
Sensor

The documentation for this class was generated from the following files:

- **sensorbase.h**
- sensor.cc

10.68 BEAST::UniformNoise Struct Reference

Plots uniform noise in a distribution.

`#include <bacteria.h>`

Public Member Functions

- **UniformNoise** (double minimum, double maximum)
- double **operator()** (int, int)

Public Attributes

- double **range**
- double **mod**

10.68.1 Detailed Description

Plots uniform noise in a distribution.

10.68.2 Constructor & Destructor Documentation

10.68.2.1 BEAST::UniformNoise::UniformNoise (double *minimum*, double *maximum*) [inline]

Parameters:

minimum The lowest value of noise.

maximum The highest value of noise.

The documentation for this struct was generated from the following file:

- **bacteria.h**

10.69 BEAST::Unserialiser Class Reference

This class is available for unserialising objects from streams, without knowing which type of object is to be unserialised - the type is determined from the header of the stream.

`#include <unserialiser.h>`

Public Member Functions

- void **Add** (std::string name, **ObjLoaderBase** *func)

  Adds a new **ObjLoader** *functor to the map, deleting any existing one.*

- **WorldObject** * **operator()** (std::istream &in)

  Calling the **Unserialiser** *as a function object will return a pointer to the next serialised object on the specified istream, or NULL if none is found.*

Static Public Member Functions

- **Unserialiser** & **Instance** ()

  Returns a reference to the one and only **Unserialiser** *object.*

10.69.1 Detailed Description

This class is available for unserialising objects from streams, without knowing which type of object is to be unserialised - the type is determined from the header of the stream.

Since it's a singleton object, there is only ever one reference to it at a time. To access that reference, use **Unserialiser::Instance()**.

The documentation for this class was generated from the following files:

- **unserialiser.h**
- **unserialiser.cc**

10.70 BEAST::Vector2D Class Reference

A class for representing two-dimensional vectors and coordinates.

`#include <vector2d.h>`

Public Member Functions

- **Vector2D** (double X, double Y)
- **Vector2D** (double X, double Y, double l, double a)
- **Vector2D** (const **Vector2D** &v, double l, double a)
- **Vector2D operator+** (const **Vector2D** &) const

  Returns this vector plus other.

- **Vector2D operator-** (const **Vector2D** &) const

  Returns this vector minus other.

- **Vector2D operator** * (double) const

  Returns this vector multiplied by l.

- **Vector2D** & **operator+=** (const **Vector2D** &)

  Adds other to this vector.

- **Vector2D** & **operator-=** (const **Vector2D** &)

  Subtracts other from this vector.

- **Vector2D** & **operator** *= (double)

  Multiplies this vector's length by the specified value.

- **Vector2D operator-** () const

  Negates both values of the vector.

- bool **operator==** (const **Vector2D** &) const

  Returns true if this vector is equal to other.

- bool **operator!=** (const **Vector2D** &) const

  Returns true if this vector is not equal to other.

- void **SetX** (double X)
- void **SetY** (double Y)
- void **SetPolarCoordinates** (double, double)

  Sets the vector up using polar coordinates.

- void **SetCartesian** (double, double)

  Quick way of setting X and Y of vector at the same time.

- void **SetLength** (double)

  Sets the length of the vector.

- bool **incLength** (double)

- bool **decLength** (double)
- void **SetAngle** (double)

  Sets the angle of the vector.

- void **normalise** ()

  Converts the vector into a unit vector with the same angle.

- void **rotate** (double)

  Rotates the vector by the specified number of radians.

- **Vector2D rotation** (double) const

  Returns the vector rotated by the specified number of radians.

- double **GetX** () const
- double **GetY** () const
- double **GetLength** () const

  Returns the length of the vector, if possible use GetLengthSquared instead.

- double **GetLengthSquared** () const

  Returns the square of the vector's length, useful for quicker comparisons.

- double **GetAngle** () const

  Returns the angle of the vector in radians.

- double **GetGradient** () const

  Returns the gradient of the vector.

- **Vector2D GetReciprocal** () const

  Returns the opposite vector.

- **Vector2D GetNormalised** () const

  Returns a unit vector with the same angle as the current vector.

- **Vector2D GetPerpendicular** () const

  Returns the perpendicular to the vector/.

- double **dot** (const **Vector2D** &) const

  Returns the dot product of the vector with other.

- void **Serialise** (std::ostream &) const

  Writes a **Vector2D** *to an output stream.*

- void **Unserialise** (std::istream &)

  Reads a **Vector2D** *from an input stream.*

Public Attributes

- double **x**
- double **y**

Related Functions

(Note that these are not member functions.)

- std::ostream & **operator**<< (std::ostream &out, const **Vector2D** &v)
  Output operator for **Vector2D**.
- std::istream & **operator**>> (std::istream &in, **Vector2D** &v)
  Input operator for **Vector2D**.

10.70.1 Detailed Description

A class for representing two-dimensional vectors and coordinates.

All usual operators are overloaded and other features are provided.

The documentation for this class was generated from the following file:

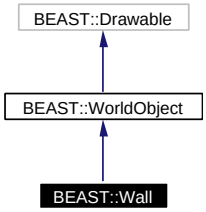
- vector2d.h

10.71 BEAST::Wall Class Reference

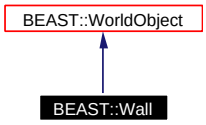
This is a handy class for putting the most common type of obstacle - walls - into the world.

`#include <worldobject.h>`

Inheritance diagram for BEAST::Wall:



Collaboration diagram for BEAST::Wall:



Public Member Functions

- **Wall** (**Vector2D** pos=**Vector2D**(), double w=50.0, double h=50.0, double o=PI/2)
  Constructs a rectangular grey **WorldObject** *with the specified properties.*

Static Public Member Functions

- std::vector< **Vector2D** > **GetSides** (double w, double h)
  Returns a vector of points on a wall using width and height.

10.71.1 Detailed Description

This is a handy class for putting the most common type of obstacle - walls - into the world.

10.71.2 Constructor & Destructor Documentation

10.71.2.1 BEAST::Wall::Wall (Vector2D *pos* = Vector2D(), double *w* = 50.0, double *h* = 50.0, double *o* = PI/2) `[inline]`

Constructs a rectangular grey **WorldObject** with the specified properties.

Parameters:

pos The x,y location of the centre of the wall.

w The width of the wall

h The height of the wall

o The orientation of the wall (defaults to vertical - 90 deg).

The documentation for this class was generated from the following files:

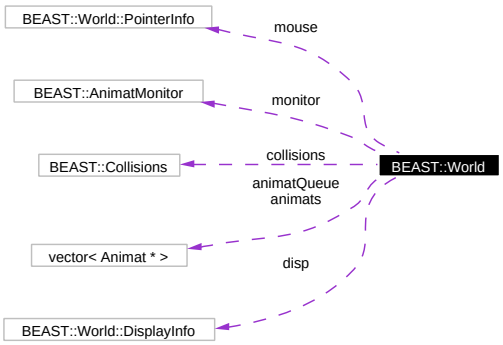
- worldobject.h
- **worldobject.cc**

10.72 BEAST::World Class Reference

This is where it all happens: **World** contains pointers to every object in the simulation environment and allows those objects to interact with each other, and then be displayed.

`#include <world.h>`

Collaboration diagram for BEAST::World:



Public Member Functions

- **World** ()

  Constructor, simply configures the world's monitor and collisions classes.

- void **Init** ()

  Calls Init on every **Animat** *and* **WorldObject** *in the* **World**.

- void **InitGL** () const

  Sets up GL with the correct background colour, projection mode and blend function.

- void **Add** (**Animat** *r)

  Adds an **Animat** *to the World's animat container, set's the Animat's world to this one and adds a pointer to the* **Animat** *to the monitor object.*

- void **Add** (**WorldObject** *r)

  Adds a **WorldObject** *to the World's worldobject container and set's the WorldObject's world to this one.*

- void **AddCollision** (**Vector2D** pv)

  Adds a collision to the collisions object.

- template<typename T> void **Add** (const vector< T * > &v)

  Adds a vector containing pointers to objects of the specified type to the world.

- template<typename T> void **Remove** (vector< T * > &v)

  Removes all objects of the specified type from the **World** *and inserts them into the vector.*

- template<typename T> void **Remove** ()

Removes all objects of the specified type from the **World**.

- void **Display** ()

  Calls the Display method of every object in the world, depending on this World's DisplayInfo struct.

- void **Update** ()

  Called every frame and responsible for calling Update on every **WorldObject** *and* **Animat** *in the* **World**.

- void **CleanUp** ()

  Clears all containers in this **World**.

- void **OnMouseLDown** (int x, int y)
- void **OnMouseRDown** (int x, int y)
- void **OnMouseLUp** (int x, int y)
- void **OnMouseRUp** (int x, int y)
- void **OnMouseMove** (int x, int y)
- void **OnSelectNext** ()
- void **OnSelectPrevious** ()
- **Vector2D Centre** ()

  Returns the centre coordinates.

- **Vector2D RandomLocation** () const

  Returns a random coordinate.

- void **SetWidth** (double w)
- void **SetHeight** (double h)
- void **SetWindow** (int w, int h)
- void **Toggle** (**WorldDisplayType** t)
- void **SetColour** (const float c[3])
- void **SetColour** (float r, float g, float b)
- double **GetWidth** () const

  Returns the current width of the world.

- double **GetHeight** () const

  Returns the current height of the world.

- int **GetWinWidth** () const

  Returns the current width of the world.

- int **GetWinHeight** () const

  Returns the current height of the world.

- int **GetDispConfig** () const

  Returns the current display configuration.

- bool **IsUpdating** () const

  Returns true if the **World** *is currently performing an update.*

- template<typename T> void **Get** (vector< T * > &v)

  Finds every object in the **World** *of the specified type and adds it to a vector.*

- **WorldObject** * **GetSelected** () const
- AnimatContainer::const_iterator **AnimatsBegin** () const

  Provides direct access to the start of the animat container.

- AnimatContainer::const_iterator **AnimatsEnd** () const

  Provides direct access to the end of the animat container.

- WorldObjectContainer::const_iterator **WorldObjectsBegin** () const

  Provides direct access to the start of the worldobject container.

- WorldObjectContainer::const_iterator **WorldObjectsEnd** () const

  Provides direct access to the end of the worldobject container.

10.72.1 Detailed Description

This is where it all happens: **World** contains pointers to every object in the simulation environment and allows those objects to interact with each other, and then be displayed.

10.72.2 Member Function Documentation

10.72.2.1 template<typename T> void BEAST::World::Add (const vector< T * > & *v*) [inline]

Adds a vector containing pointers to objects of the specified type to the world.

The vector may be a type derived from vector, e.g. **Population** or **Group**.

Parameters:

T The type of objects to be added, must be derived from **Animat** or **WorldObject**.

v A reference to an input vector of pointers to the specified type.

10.72.2.2 void BEAST::World::Add (WorldObject * *r*)

Adds a **WorldObject** to the World's worldobject container and set's the WorldObject's world to this one.

Parameters:

r A pointer to the **WorldObject** to be added.

10.72.2.3 void BEAST::World::Add (Animat * *r*)

Adds an **Animat** to the World's animat container, set's the Animat's world to this one and adds a pointer to the **Animat** to the monitor object.

Parameters:

r A pointer to the **Animat** to be added.

10.72.2.4 AnimatContainer::const_iterator BEAST::World::AnimatsBegin () const [inline]

Provides direct access to the start of the animat container.

Warning:
Should not be used when IsUpdating is true.

10.72.2.5 AnimatContainer::const_iterator BEAST::World::AnimatsEnd () const [inline]

Provides direct access to the end of the animat container.

Warning:
Should not be used when IsUpdating is true.

10.72.2.6 void BEAST::World::Display ()

Calls the Display method of every object in the world, depending on this World's DisplayInfo struct.

See also:
DisplayInfo
WorldDisplayType

10.72.2.7 template<typename T> void BEAST::World::Get (vector< T * > & *v*) [inline]

Finds every object in the **World** of the specified type and adds it to a vector.

Parameters:
T The type of objects to get.

v A reference to a vector of pointers to objects of the specified type.

10.72.2.8 void BEAST::World::Init ()

Calls Init on every **Animat** and **WorldObject** in the **World**.

Usually called at the start of a simulation, to allow objects to be set up correctly (e.g. defining display lists and performing configuration which can't be done til an object knows which **World** it's in.

See also:
WorldObject::Init
Animat::Init

10.72.2.9 template<typename T> void BEAST::World::Remove () [inline]

Removes all objects of the specified type from the **World**.

Warning:
Will not work during **World::Update** - only use between assessments.

10.72.2.10 template<typename T> void BEAST::World::Remove (vector< T * > & *v*) [inline]

Removes all objects of the specified type from the **World** and inserts them into the vector.

Warning:
Will not work during **World::Update** - only use between assessments.

Parameters:
T The type of objects to remove.
v A reference to a vector of pointers to the specified type.

10.72.2.11 void BEAST::World::Update ()

Called every frame and responsible for calling Update on every **WorldObject** and **Animat** in the **World**.

See also:
WorldObject::Update
Animat::Update

10.72.2.12 WorldObjectContainer::const_iterator BEAST::World::WorldObjectsBegin () const [inline]

Provides direct access to the start of the worldobject container.

Warning:
Should not be used when IsUpdating is true.

10.72.2.13 WorldObjectContainer::const_iterator BEAST::World::WorldObjectsEnd () const [inline]

Provides direct access to the end of the worldobject container.

Warning:
Should not be used when IsUpdating is true.

The documentation for this class was generated from the following files:

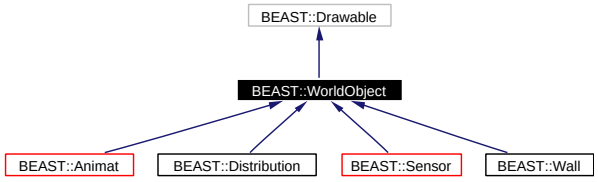
- **world.h**
- world.cc

10.73 BEAST::WorldObject Class Reference

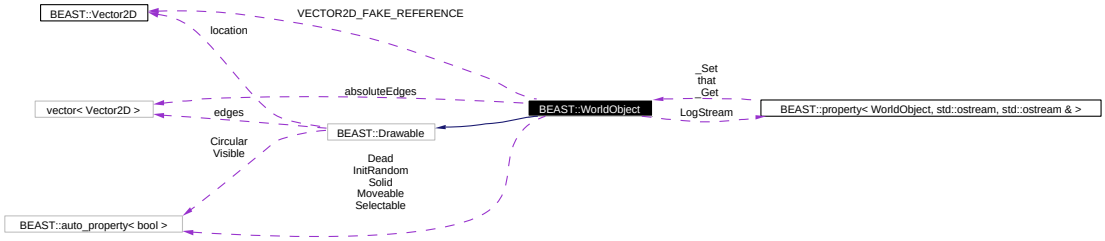
The base class for everything that makes a difference in the world, including Animats, Sensors and all types of scenery and interactive object.

`#include <worldobject.h>`

Inheritance diagram for BEAST::WorldObject:



Collaboration diagram for BEAST::WorldObject:



Public Member Functions

- **WorldObject** (const **Vector2D** &l=**Vector2D**(0, 0), double o=0.0, double d=**DRAWABLE_RADIUS**, bool so=false)
- **WorldObject** (const **Vector2D** &l, double o, std::vector< **Vector2D** > e, bool so=false)
- virtual void **Init** ()

  Initialises **WorldObject** *by calculating edges if required, and setting a random location if initRandom is set.*

- virtual void **Update** ()
- virtual void **Interact** (**WorldObject** *)

  Calls the one-way interact method of this and another **WorldObject**.

- virtual void **UniInteract** (**WorldObject** *)
- virtual void **OnCollision** (**WorldObject** *r)
- virtual bool **IsInside** (const **Vector2D** &vec) const

  Returns true if the specified point is inside this object.

- **Vector2D GetNearestPoint** (const **WorldObject** *i)
- **Vector2D GetNearestPoint** (const **Vector2D** &, **Vector2D** &r=VECTOR2D_FAKE_REFERENCE) const

Returns the nearest point on this object to the argument, and also returns the collision normal by reference.

- bool **Intersects** (const **Vector2D** &l1, const **Vector2D** &l2, **Vector2D** &r=VECTOR2D_FAKE_REFERENCE) const

  Returns true if the line defined by the two inputs intersects with this object at some point, and can also return the intersection by reference.

- virtual void **OnClick** ()
- virtual void **OnSelect** ()
- bool **IsSolid** () const
- bool **IsDead** () const
- bool **IsInitRandom** () const
- bool **IsMoveable** () const
- bool **IsSelectable** () const
- void **SetSolid** (bool s)
- void **SetDead** (bool d)
- void **SetInitRandom** (bool r)
- void **SetMoveable** (bool m)
- void **SetSelectable** (bool s)
- void **_SetLogStream** (std::ostream &o)
- std::ostream & **GetLogStream** () const
- virtual std::string **ToString** () const

  Returns basic information about this object as a string.

- virtual void **Serialise** (std::ostream &) const

  Outputs the object's data to a stream.

- virtual void **Unserialise** (std::istream &)

  Sets up the object from a stream.

Static Public Member Functions

- void **SetLogStream** (std::ostream &o)

Public Attributes

- auto_property< bool > **Solid**
- auto_property< bool > **Dead**
- auto_property< bool > **InitRandom**
- auto_property< bool > **Moveable**
- auto_property< bool > **Selectable**
- **property< WorldObject**, std::ostream, std::ostream & > **LogStream**

10.73.1 Detailed Description

The base class for everything that makes a difference in the world, including Animats, Sensors and all types of scenery and interactive object.

WorldObject provides many overridable methods which ensure that just about any kind of thing can be represented in the simulation environment. Of particular importance are Init, Update, Interact and OnCollision, the main methods used in making a useful simulation object. **WorldObject** also handles some collision detection and provides methods for detecting if other objects are touching this one.

10.73.2 Member Function Documentation

10.73.2.1 Vector2D BEAST::WorldObject::GetNearestPoint (const Vector2D & *vec*, Vector2D & *collisionNormal* = VECTOR2D_FAKE_REFERENCE) const

Returns the nearest point on this object to the argument, and also returns the collision normal by reference.

Parameters:

vec The point we are comparing.

collisionNormal The normal between the nearest side to vec and vec.

Returns:

The nearest point on this object to vec.

10.73.2.2 void BEAST::WorldObject::Interact (WorldObject * *other*) [virtual]

Calls the one-way interact method of this and another **WorldObject**.

See UniInteract for the use of one-way interactions.

Parameters:

other The **WorldObject** we're interacting with.

Reimplemented in **BEAST::Animat** (p. 72), **BEAST::Bacterium** (p. 75), **BEAST::Sensor** (p. 182), **BEAST::SelfSensor** (p. 180), **BEAST::AreaSensor** (p. 74), **BEAST::TouchSensor** (p. 200), and **BEAST::BeamSensor** (p. 85).

10.73.2.3 bool BEAST::WorldObject::Intersects (const Vector2D & *l1*, const Vector2D & *l2*, Vector2D & *intersection* = VECTOR2D_FAKE_REFERENCE) const

Returns true if the line defined by the two inputs intersects with this object at some point, and can also return the intersection by reference.

Parameters:

l1 The first point of the line to be tested.

l2 The second piont of the line to be tested.

intersection The point of intersection, returned by reference.

Returns:
True if the line intersects this object, false if not.

10.73.2.4 bool BEAST::WorldObject::IsInside (const Vector2D & *vecTest*) const [virtual]

Returns true if the specified point is inside this object.

Parameters:
vecTest The point being tested.

Returns:
True if vecTest is inside this object.

10.73.2.5 void BEAST::WorldObject::Serialise (std::ostream & *out*) const [virtual]

Outputs the object's data to a stream.

See also:
WorldObject::Unserialise
Drawable::Serialise

Reimplemented in **BEAST::Animat** (p. 73), **BEAST::FFNAnimat** (p. 135), and **BEAST::DNNAnimat** (p. 101).

10.73.2.6 void BEAST::WorldObject::Unserialise (std::istream & *in*) [virtual]

Sets up the object from a stream.

See also:
WorldObject::Serialise
Drawable::Unserialise

Reimplemented in **BEAST::Animat** (p. 73), **BEAST::FFNAnimat** (p. 135), and **BEAST::DNNAnimat** (p. 101).

The documentation for this class was generated from the following files:

- worldobject.h
- **worldobject.cc**

Chapter 11

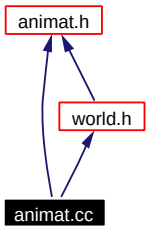
BEAST - Bioinspired Evolutionary Agent Simulation Toolkit File Documentation

11.1 animat.cc File Reference

Implementation of the Animat class.

```
#include "animat.h"
#include "world.h"
```

Include dependency graph for animat.cc:



Namespaces

- namespace **BEAST**
- namespace **std**

11.1.1 Detailed Description

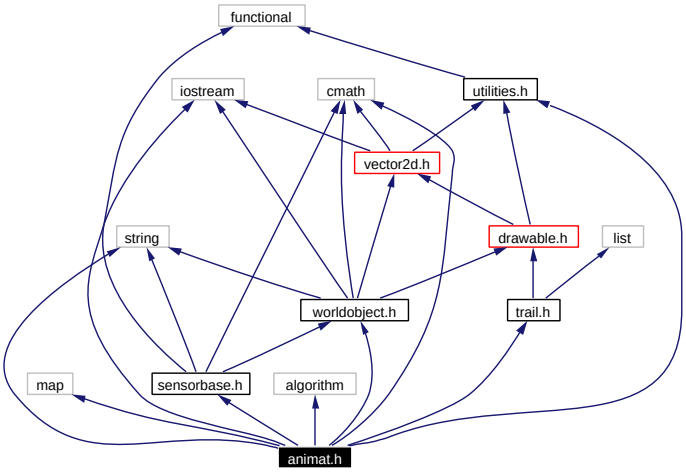
Implementation of the Animat class.

11.2 animat.h File Reference

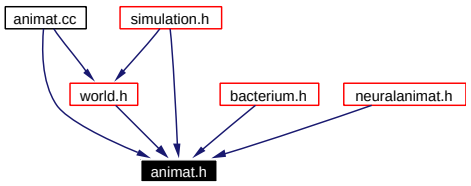
Interface of the Animat class and associated constants.

```
#include <iostream>
#include <map>
#include <string>
#include <cmath>
#include <algorithm>
#include "worldobject.h"
#include "sensorbase.h"
#include "trail.h"
#include "utilities.h"
```

Include dependency graph for animat.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace **BEAST**

11.2.1 Detailed Description

Interface of the Animat class and associated constants.

#include this file if you are deriving an Animat with a unique control system (overloaded Control method) which you are writing from scratch. If you are working with neural nets you may find it more useful to start with FFNAnimat and DNNAnimat which come with their own neural nets and automatic configuration methods.

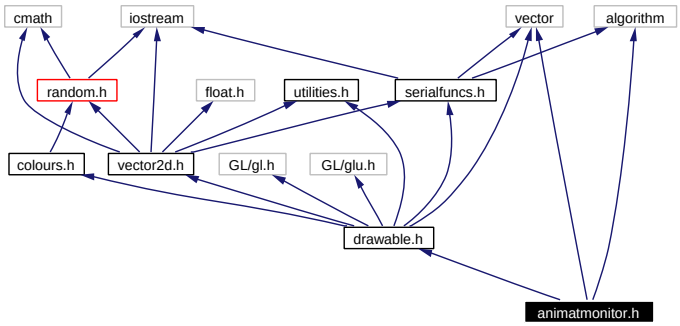
Author:
Tom Carden
David Gordon

See also:
neuralanimat.h
FFNAnimat
DNNAnimat

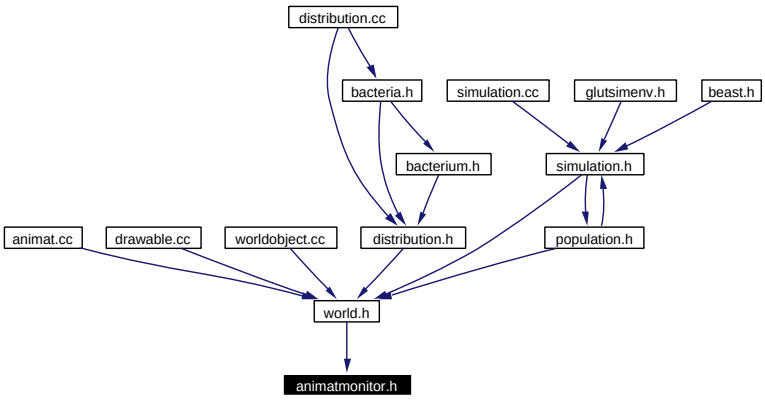
11.3 animatmonitor.h File Reference

```
#include <vector>
#include <algorithm>
#include "drawable.h"
```

Include dependency graph for animatmonitor.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace **BEAST**

11.3.1 Detailed Description

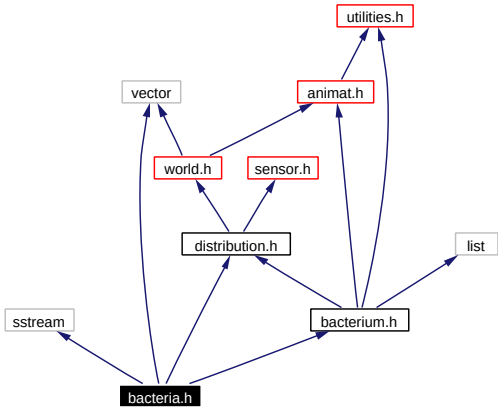
Author:
Tom Carden
David Gordon

11.4 bacteria.h File Reference

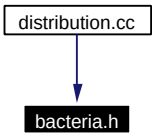
Global include for all bacteria-related classes.

```
#include <sstream>
#include <vector>
#include "distribution.h"
#include "bacterium.h"
```

Include dependency graph for bacteria.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace **BEAST**

11.4.1 Detailed Description

Global include for all bacteria-related classes.

Include this file to have access to all bacteria-related classes.

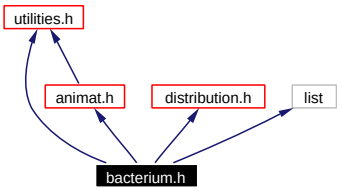
Author:
David Gordon

11.5 bacterium.h File Reference

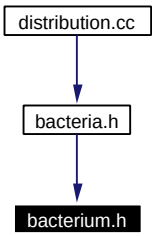
The interface for the Bacterium class.

```
#include "animat.h"
#include "distribution.h"
#include "utilities.h"
#include <list>
```

Include dependency graph for bacterium.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace **BEAST**

11.5.1 Detailed Description

The interface for the Bacterium class.

Author:
David Gordon

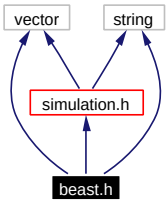
11.6 beast.h File Reference

```
#include <vector>
```

```
#include <string>
```

```
#include "simulation.h"
```

Include dependency graph for beast.h:



Namespaces

- namespace **BEAST**

Defines

- #define **BEGIN_SIMULATION_TABLE**

  Denotes the beginning of the simulation table.

- #define **ADD_SIMULATION**(_name, _type)

  Call this once with the textual name and class name of each simulation you wish to make available in the GUI.

- #define **END_SIMULATION_TABLE**

  Call this to denote the end of your simulation table, the list of simulations to be made available in the GUI.

11.6.1 Detailed Description

Author:
David Gordon

The main include file for the simulation environment/wx - include this file if you want to run simulations using the wxWindows-based interface.

11.6.2 Define Documentation

11.6.2.1 #define ADD_SIMULATION(_name, _type)

Value:

```
names.push_back(_name);                                 \
    funcs.push_back(new GetSimulation<_type>);
```

Call this once with the textual name and class name of each simulation you wish to make available in the GUI.

ADD_SIMULATION must only be called between BEGIN_SIMULATION_TABLE and END_SIMULATION_TABLE.

See also:
BEGIN_SIMULATION_TABLE
END_SIMULATION_TABLE

11.6.2.2 #define BEGIN_SIMULATION_TABLE

Value:

```
namespace BEAST {                                                \
void SetupSimulationTable                                        \
(std::vector<std::string>& names,                                \
 std::vector<GetSimulationBase*>& funcs)                         \
{
```

Denotes the beginning of the simulation table.

The simulation table is used to tell the GUI which classes (derived from Simulation) are to be made available in the File menu.

See also:
ADD_SIMULATION
END_SIMULATION_TABLE

11.6.2.3 #define END_SIMULATION_TABLE

Value:

```
}                                                                \
}
```

Call this to denote the end of your simulation table, the list of simulations to be made available in the GUI.

See also:
BEGIN_SIMULATION_TABLE
END_SIMULATION_TABLE

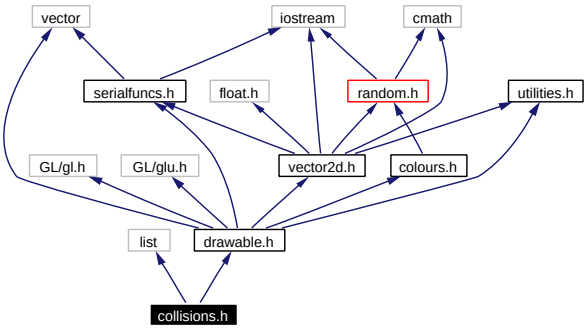
11.7 collisions.h File Reference

Draws collisions in the World.

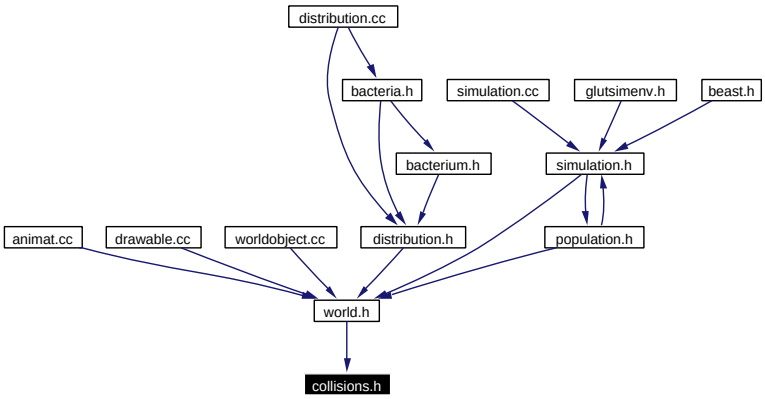
```
#include <list>
```

```
#include "drawable.h"
```

Include dependency graph for collisions.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace **BEAST**

11.7.1 Detailed Description

Draws collisions in the World.

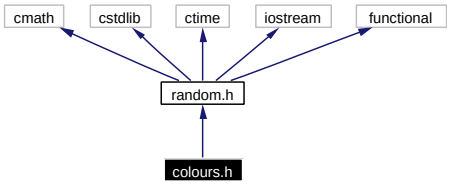
Author:
Tom Carden

11.8 colours.h File Reference

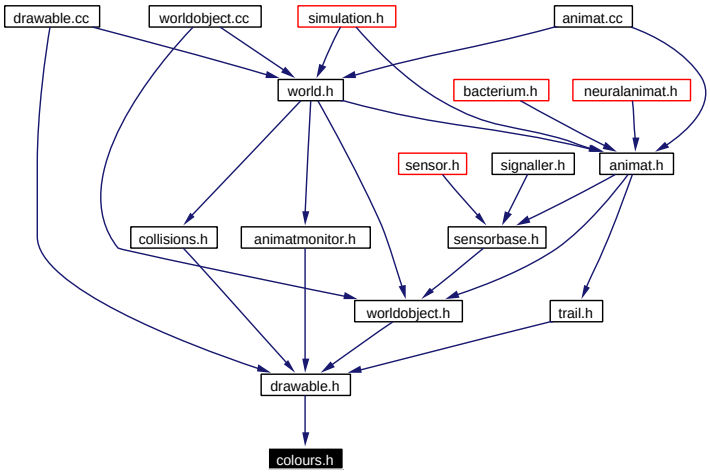
A handy include which provides a bunch of colours.

```
#include "random.h"
```

Include dependency graph for colours.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace **BEAST**

11.8.1 Detailed Description

A handy include which provides a bunch of colours.

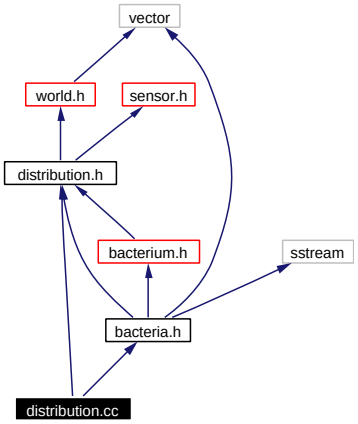
Author:
Tom Carden
David Gordon

11.9 distribution.cc File Reference

Implementation of the Distribution class.

```
#include "distribution.h"
#include "bacteria.h"
```

Include dependency graph for distribution.cc:



Namespaces

- namespace **BEAST**

11.9.1 Detailed Description

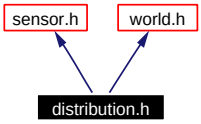
Implementation of the Distribution class.

11.10 distribution.h File Reference

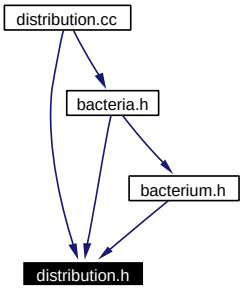
Implements a two-dimensional density distribution.

```
#include "sensor.h"
#include "world.h"
```

Include dependency graph for distribution.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace **BEAST**

11.10.1 Detailed Description

Implements a two-dimensional density distribution.

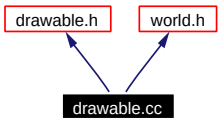
Author:
David Gordon

11.11 drawable.cc File Reference

Implementation of Drawable.

```
#include "drawable.h"
#include "world.h"
```

Include dependency graph for drawable.cc:



Namespaces

- namespace **BEAST**

11.11.1 Detailed Description

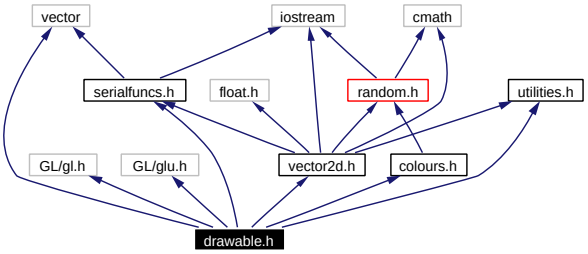
Implementation of Drawable.

11.12 drawable.h File Reference

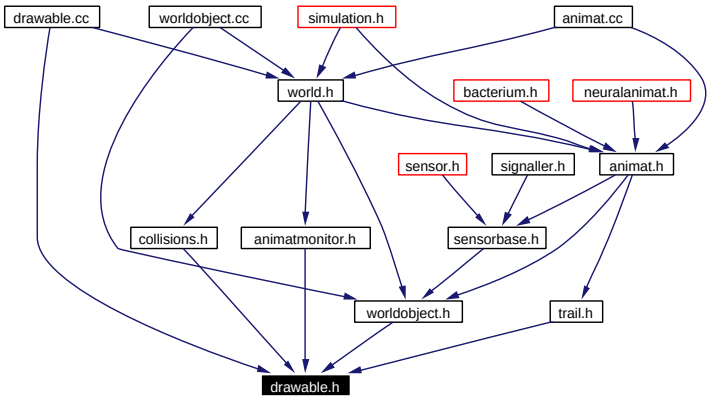
Include this file if you wish to create scenery or other non-interactive objects which appear in the world.

```
#include <vector>
#include <GL/gl.h>
#include <GL/glu.h>
#include "vector2d.h"
#include "colours.h"
#include "serialfuncs.h"
#include "utilities.h"
```

Include dependency graph for drawable.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace **BEAST**

Defines

- #define **This** (*this)

11.12.1 Detailed Description

Include this file if you wish to create scenery or other non-interactive objects which appear in the world.

Everything that appears in the world is a Drawable, and all WorldObjects and Animats are derived from it.

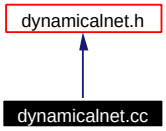
Author:
Tom Carden
David Gordon

11.13 dynamicalnet.cc File Reference

Implementation of DynamicalNet.

```
#include "dynamicalnet.h"
```

Include dependency graph for dynamicalnet.cc:



Namespaces

- namespace **BEAST**

11.13.1 Detailed Description

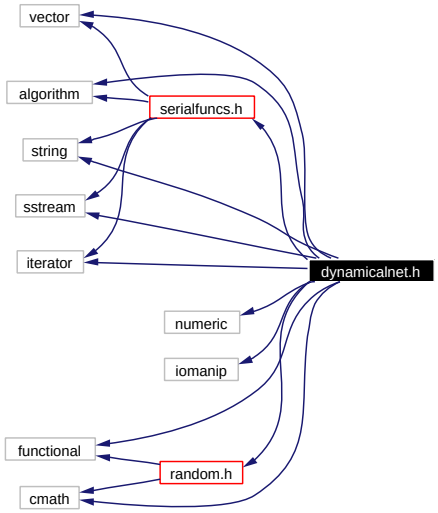
Implementation of DynamicalNet.

11.14 dynamicalnet.h File Reference

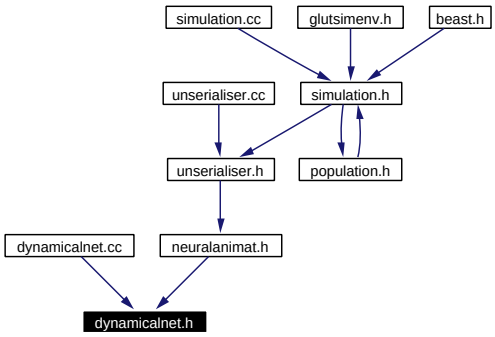
This file contains the interface for the DynamicalNet object, a fully-recurrent, continuous-time neural network.

```
#include <vector>
#include <algorithm>
#include <string>
#include <sstream>
#include <iterator>
#include <functional>
#include <numeric>
#include <iomanip>
#include <cmath>
#include "random.h"
#include "serialfuncs.h"
```

Include dependency graph for dynamicalnet.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace **BEAST**

11.14.1 Detailed Description

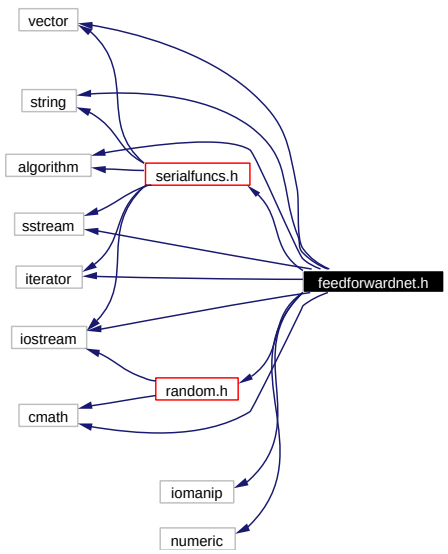
This file contains the interface for the DynamicalNet object, a fully-recurrent, continuous-time neural network.

Author:
David Gordon biosystems

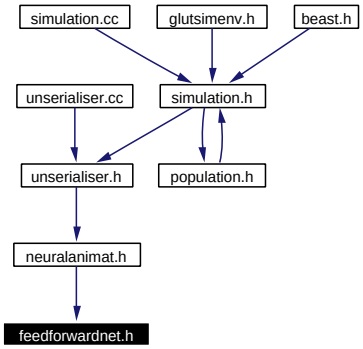
11.15 feedforwardnet.h File Reference

```
#include <vector>
#include <string>
#include <algorithm>
#include <cmath>
#include <iostream>
#include <sstream>
#include <iomanip>
#include <numeric>
#include <iterator>
#include "random.h"
#include "serialfuncs.h"
```

Include dependency graph for feedforwardnet.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace **BEAST**

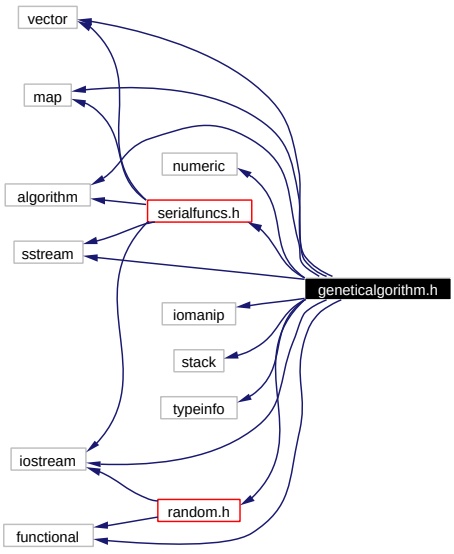
11.15.1 Detailed Description

Author:
David Gordon

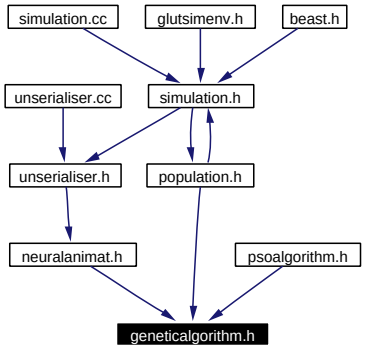
11.16 geneticalgorithm.h File Reference

```
#include <vector>
#include <map>
#include <algorithm>
#include <numeric>
#include <sstream>
#include <iostream>
#include <iomanip>
#include <stack>
#include <typeinfo>
#include <functional>
#include "random.h"
#include "serialfuncs.h"
```

Include dependency graph for geneticalgorithm.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace **BEAST**

11.16.1 Detailed Description

Author:
David Gordon

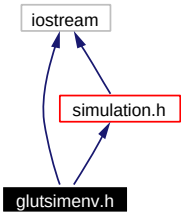
11.17 glutsimenv.h File Reference

The main include file for the simulation environment/GLUT - include this file if you want to run simulations using the simple GLUT-based interface.

```
#include <iostream>
```

```
#include "simulation.h"
```

Include dependency graph for glutsimenv.h:



Namespaces

- namespace **BEAST**

Defines

- #define **START_SIMULATION**(_SimClass)

11.17.1 Detailed Description

The main include file for the simulation environment/GLUT - include this file if you want to run simulations using the simple GLUT-based interface.

11.17.2 Define Documentation

11.17.2.1 #define START_SIMULATION(_SimClass)

Value:

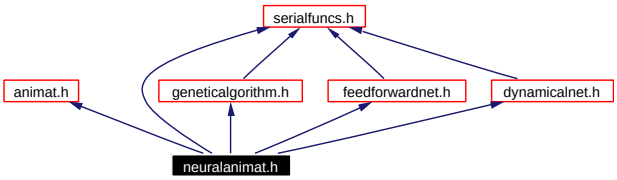
```
int main(int argc, char* argv[])\
{\
    _SimClass theSimulation;\
    glut_start_simulation(argc, argv, &theSimulation);\
    return 0;\
}
```

11.18 neuralanimat.h File Reference

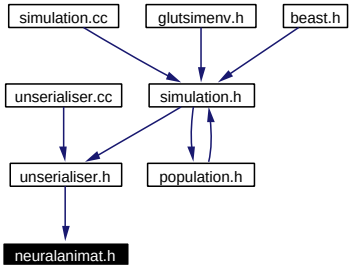
The basic Animat comes with no control system, so in the course of deriving a new type of Animat, a control system must be added.

```
#include "animat.h"
#include "geneticalgorithm.h"
#include "feedforwardnet.h"
#include "dynamicalnet.h"
#include "serialfuncs.h"
```

Include dependency graph for neuralanimat.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace **BEAST**

11.18.1 Detailed Description

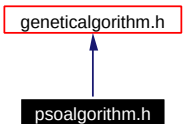
The basic Animat comes with no control system, so in the course of deriving a new type of Animat, a control system must be added.

Two useful control systems are the FeedForwardNet and DynamicalNet classes. FFNAnimat and DNNAnimat provide Animats with these control systems built-in and automatically configured from the Animat's sensors and controls. Two other classes, EvoFFNAnimat and EvoDNNAnimat are provided as evolvable versions in case the only data contained in the genotype is the ANN configuration. biosystems

11.19 psoalgorithm.h File Reference

```
#include "geneticalgorithm.h"
```

Include dependency graph for psoalgorithm.h:



Namespaces

- namespace **BEAST**

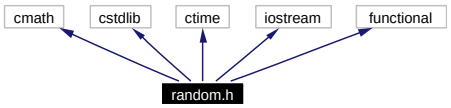
11.19.1 Detailed Description

Author:
David Gordon biosystems

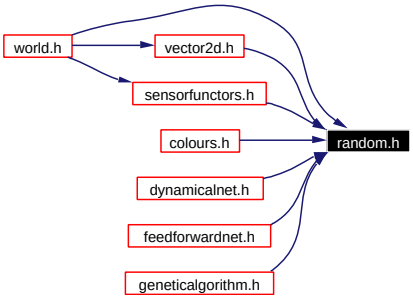
11.20 random.h File Reference

```
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <functional>
```

Include dependency graph for random.h:



This graph shows which files directly or indirectly include this file:



Compounds

- struct **Random**

  Function object version of randval.

Defines

- #define **M1** 259200
- #define **IA1** 7141
- #define **IC1** 54773
- #define **RM1** (1.0/M1)
- #define **M2** 134456
- #define **IA2** 8121
- #define **IC2** 28411
- #define **RM2** (1.0/M2)
- #define **M3** 243000
- #define **IA3** 4561
- #define **IC3** 51349

Functions

- float **ran1** (int *idum)
- int **rseed** (int *s, bool verbose)
- template<typename Real> Real **randval** (Real limit)

  Returns a (near) uniform distributed random number in the range 0..limit, as a Real.

- int **irand** (int limit)

  Returns a random integer in [0..limit-1].

- bool **brand** (double p)
- bool **brand** (float p)
- template<> int **randval** (int limit)

  Template specialisation to stop randval from being called with ints.

- template<> bool **randval** (bool)
- template<typename Real> Real **gaussrand** (void)

  Returns a normally distributed variable with zero mean and unit variance.

- template<typename Real> Real **normrand** (Real mean, Real sd)

  Returns a random deviate from a normal distribution with specified mean and standard distribution.

11.20.1 Detailed Description

Author:
Dave Cliff

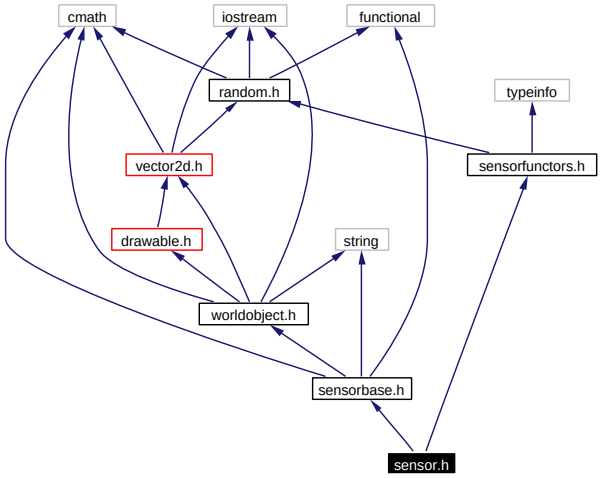
Date:
May 1991 utilities Some general random routines, and some useful #defines Modified by Dave Gordon 2002 Moved the #defines elsewhere and turned everything into C++ templates

11.21 sensor.h File Reference

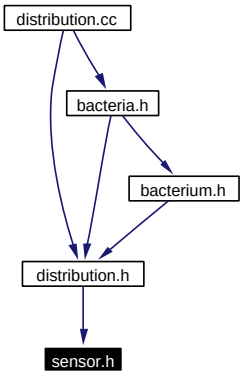
```
#include "sensorbase.h"
```

```
#include "sensorfunctors.h"
```

Include dependency graph for sensor.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace **BEAST**

11.21.1 Detailed Description

Author:

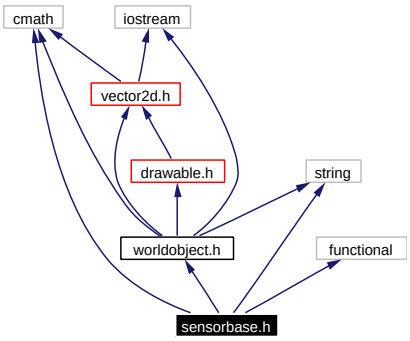
David Gordon sensors Include this file for all sensor functionality. **sensorbase.h** and **sensorfunctors.h** are included by this file. **sensor.h** contains some useful functions which set up the most common types of sensor.

11.22 sensorbase.h File Reference

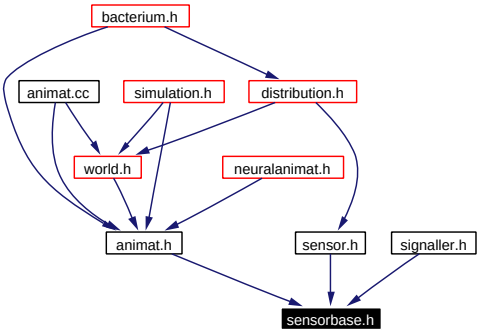
All basic sensor objects are defined in this file, but no sensor functors, or helper functions appear here.

```
#include <cmath>
#include <functional>
#include <string>
#include "worldobject.h"
```

Include dependency graph for sensorbase.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace **BEAST**

11.22.1 Detailed Description

All basic sensor objects are defined in this file, but no sensor functors, or helper functions appear here.

If you want to include sensors in your simulation, include the file **sensor.h** instead.

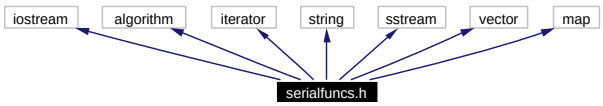
Author:
David Gordon sensors

11.23 serialfuncs.h File Reference

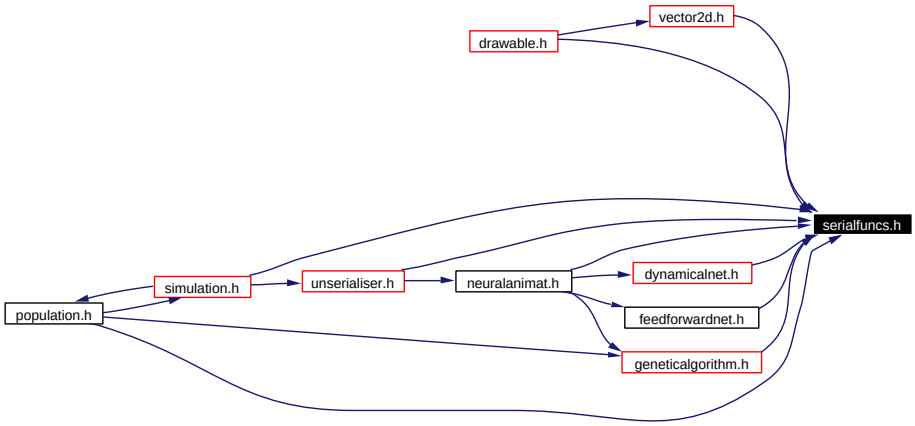
A bunch of methods, templates and classes for performing quick stream insertion and extraction, include this file if you need to serialise your objects' data.

```
#include <iostream>
#include <algorithm>
#include <iterator>
#include <string>
#include <sstream>
#include <vector>
#include <map>
```

Include dependency graph for serialfuncs.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace **BEAST**

Defines

- #define **IMPLEMENT_IOSTREAM_CAST**(_Ty, _Cast)

  Use this macro to create a custom output operator which simply casts _Ty to the type specified by _Cast.

- #define **IMPLEMENT_IOSTREAM_BINARY_CONVERSION**(_Ty)

This macro is intended as a quick solution to the problem of encoding structs and classes into output streams.

- #define **IMPLEMENT_SERIALISATION**(_Name, _Parent)

  Use this macro to add basic serialisation functionality to your derived classes by simply serialising under a new name, but using the parent serialisation methods.

- #define **IMPLEMENT_LOADER**(_Name, _Type) Unserialiser::Instance().Add(_Name, new ObjLoader<_Type>());

11.23.1 Detailed Description

A bunch of methods, templates and classes for performing quick stream insertion and extraction, include this file if you need to serialise your objects' data.

Author:
David Gordon serialisation

11.24 signaller.h File Reference

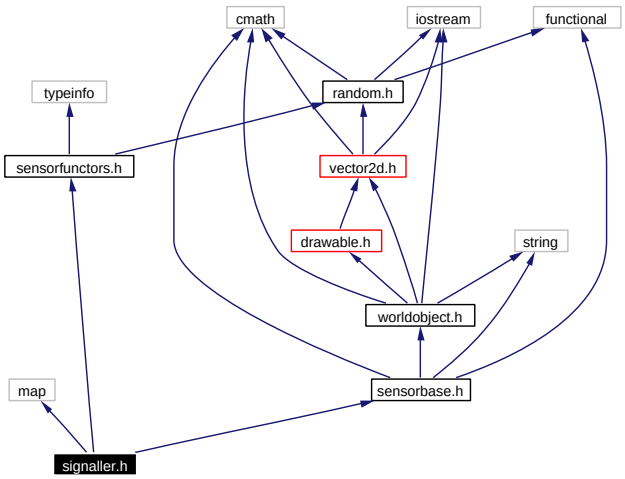
This file defines a class which can be multiply inherited with Animat to create a signalling Animat, and some associated sensor functors which.

```
#include <map>
```

```
#include "sensorbase.h"
```

```
#include "sensorfunctors.h"
```

Include dependency graph for signaller.h:



Namespaces

- namespace **BEAST**

11.24.1 Detailed Description

This file defines a class which can be multiply inherited with Animat to create a signalling Animat, and some associated sensor functors which.

Author:
David Gordon biosystems

11.25 simulation.cc File Reference

```
#include "simulation.h"
```

Include dependency graph for simulation.cc:

Namespaces

- namespace **BEAST**

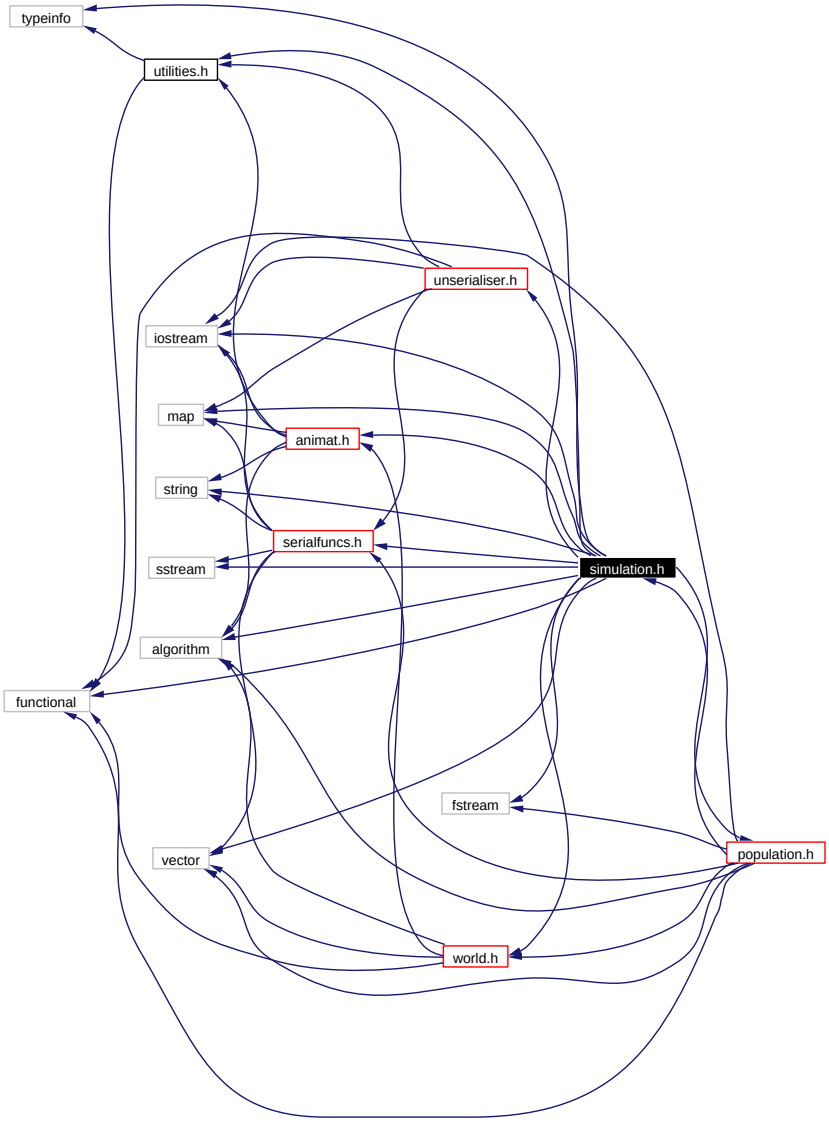
11.25.1 Detailed Description

Author:
David Gordon Implementation of the Simulation class.

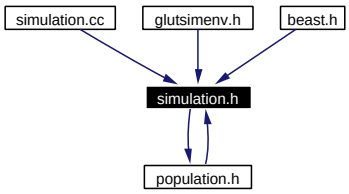
11.26 simulation.h File Reference

```
#include <string>
#include <vector>
#include <map>
#include <algorithm>
#include <functional>
#include <sstream>
#include <iostream>
#include <fstream>
#include <typeinfo>
#include "utilities.h"
#include "world.h"
#include "animat.h"
#include "serialfuncs.h"
#include "unserialiser.h"
#include "population.h"
```

Include dependency graph for simulation.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace **BEAST**

11.26.1 Detailed Description

Author:

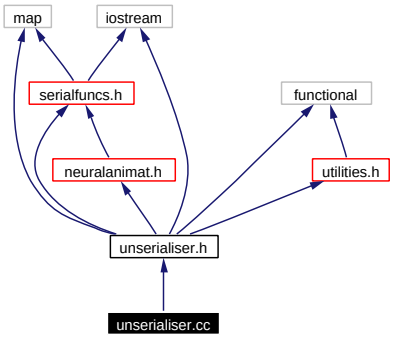
David Gordon The Simulation class is really a framework of classes providing the facilities for implementing a range of different types of simulation. SimObject is an interface class for the Group class, which simply adds objects to the Simulation, and the Population class which provides handles GA functionality, and the insertion of groups of Animats (or other evolvable WorldObjects into the World.

11.27 unserialiser.cc File Reference

Implementation of Unserialiser.

```
#include "unserialiser.h"
```

Include dependency graph for unserialiser.cc:



Namespaces

- namespace **BEAST**

11.27.1 Detailed Description

Implementation of Unserialiser.

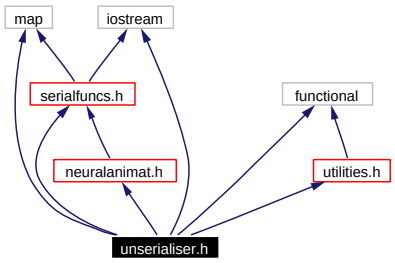
Author:
David Gordon

11.28 unserialiser.h File Reference

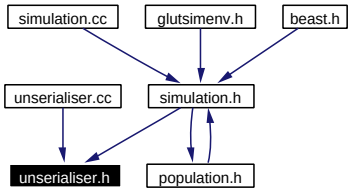
Interface of a class and related functors for unserialising unknown types.

```
#include <map>
#include <functional>
#include <iostream>
#include "utilities.h"
#include "serialfuncs.h"
#include "neuralanimat.h"
```

Include dependency graph for unserialiser.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace **BEAST**

11.28.1 Detailed Description

Interface of a class and related functors for unserialising unknown types.

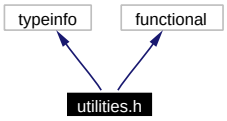
Author:
David Gordon serialisation

11.29 utilities.h File Reference

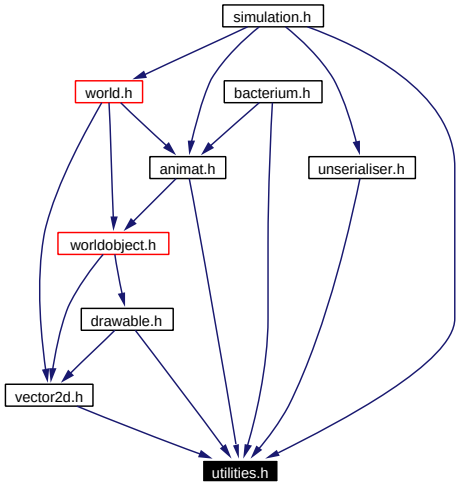
`#include <typeinfo>`

`#include <functional>`

Include dependency graph for utilities.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace **BEAST**

11.29.1 Detailed Description

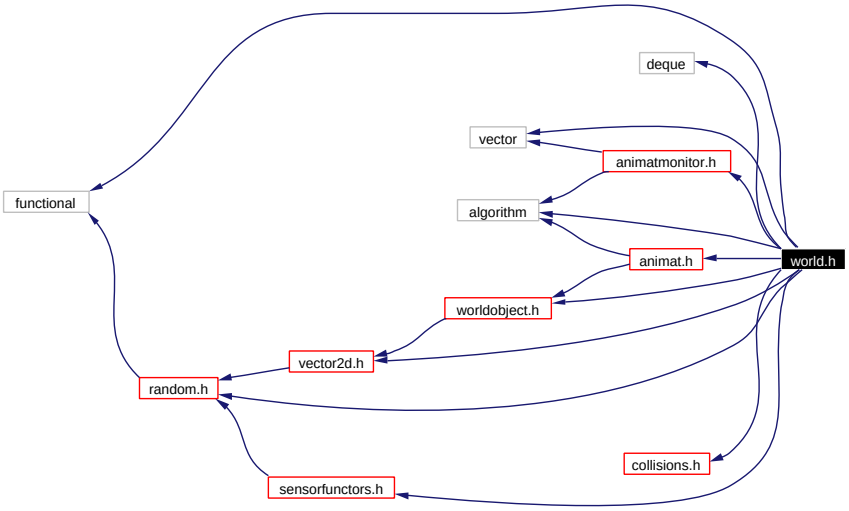
Author:

David Gordon utilities This file defines a number of useful functions and functors.

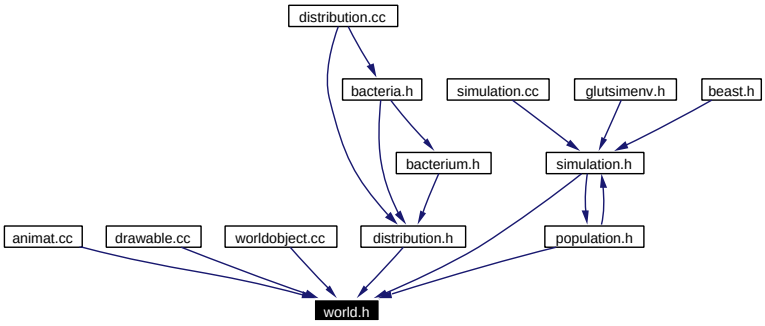
11.30 world.h File Reference

```
#include <vector>
#include <deque>
#include <algorithm>
#include <functional>
#include "vector2d.h"
#include "animat.h"
#include "animatmonitor.h"
#include "worldobject.h"
#include "collisions.h"
#include "random.h"
#include "sensorfunctors.h"
```

Include dependency graph for world.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace **BEAST**

11.30.1 Detailed Description

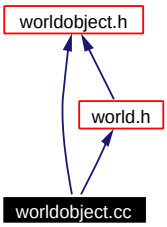
Author:
Tom Carden framework

11.31 worldobject.cc File Reference

The implementation of WorldObject.

```
#include "worldobject.h"
#include "world.h"
```

Include dependency graph for worldobject.cc:



Namespaces

- namespace **BEAST**

11.31.1 Detailed Description

The implementation of WorldObject.

Chapter 12

BEAST - Bioinspired Evolutionary Agent Simulation Toolkit Page Documentation

12.1 FeedForwardNet source code

```
#include "feedforwardnet.h"

using namespace std;

namespace BEAST {

FeedForwardNet::FeedForwardNet(int in, int out, int hid, bool sig, bool bias)
{
    Init(in, out, hid, sig, bias);
}

void FeedForwardNet::Init(int in, int out, int hid, bool sig, bool bias)
{
    inputs = in;
    outputs = out;
    hidden = hid;
    sigmoid = sig;
    biasNode = bias;
    inputValues = vector<float>(in);
    outputValues = vector<float>(out);
    hiddenLayer.clear();
    outputLayer.clear();

    // For each hidden layer neuron, we instantiate a Neuron object.
    // The Neuron is initialised with the number of weights it needs,
    // which in this case is one per input. If biasNode has been set
    // to true, an extra weight is added. The use of this is explained
    // in the fire method.
    for (int i=0; i<hidden; ++i) {
        hiddenLayer.push_back(Neuron(inputs + (biasNode ? 1 : 0) ));
    }

    // If the hidden layer size is set to 0, normally that would break the
    // network but here I'm taking it to mean the net is a perceptron (one
    // layer of inputs, one layer of outputs, no hidden layer) and so the
    // output layer neurons have <inputs>, rather than <hidden>, inputs.
    if (hid == 0) hid = inputs;
```

```
    // Add one neuron per output value to the output layer, each with
    // one weight per hidden neuron (since this is the output layer,
    // inputs are coming from the hidden layer) and again an extra
    // weight if biasNode is set to true.
    for (int i=0; i<outputs; ++i) {
        outputLayer.push_back(Neuron(hid + (biasNode ? 1 : 0) ));
    }
}

FeedForwardNet::~FeedForwardNet()
{
}

void FeedForwardNet::Randomise()
{
    vector<Neuron>::iterator i;

    for (i = hiddenLayer.begin(); i != hiddenLayer.end(); ++i) {
        generate(i->weights.begin(), i->weights.end(), RandomNum);
    }

    for (i = outputLayer.begin(); i != outputLayer.end(); ++i) {
        generate(i->weights.begin(), i->weights.end(), RandomNum);
    }
}

vector<float> FeedForwardNet::GetConfiguration()const
{
    // The configuration data contains no information about the
    // network's dimensions, bias nodes, or activation function.

    vector<float> config;
    vector<Neuron>::const_iterator i;

    // Using the STL copy template algorithm instead of a for loop. Wow!
    for (i = hiddenLayer.begin(); i != hiddenLayer.end(); ++i) {
        copy(i->weights.begin(), i->weights.end(), back_inserter(config));
    }

    for (i = outputLayer.begin(); i != outputLayer.end(); ++i) {
        copy(i->weights.begin(), i->weights.end(), back_inserter(config));
    }

    return config;
}

void FeedForwardNet::SetConfiguration(const vector<float>& config)
{
    // The configuration data contains no information about the
    // network's dimensions, bias nodes, or activation function.

    // First ensure the incoming vector is the right size
    unsigned int expectedSize = static_cast<unsigned int>(GetConfigurationLength());

    if (config.size() != expectedSize) return; // Error to go here

    vector<Neuron>::iterator i;
    vector<float>::const_iterator configIter(config.begin());

    // Now we move through config, assigning series of size inputs + biasnode
    // to the weights of each Neuron.

    for (i = hiddenLayer.begin(); i != hiddenLayer.end(); ++i) {
        i->weights = vector<float>(configIter, configIter+inputs+(biasNode?1:0));
        configIter += inputs + (biasNode ? 1 : 0);
    }
```

```
    // Same again for output layer
    for (i = outputLayer.begin(); i != outputLayer.end(); ++i) {
        i->weights = vector<float>(configIter, configIter+hidden+(biasNode?1:0));
        configIter += hidden + (biasNode ? 1 : 0);
    }

    // One more error check:
    if (configIter != config.end()) true; // Another error

    return;
}

float FeedForwardNet::Neuron::WeightedSum(vector<float>& input)const
{
    if (input.size() > weights.size()) {
        std::cerr << "Error: too many inputs!" << std::endl;
    }

    // All we're doing here is moving through the input values (which will
    // always be either the net's inputs or the outputs of the hidden layer)
    // and multiplying them by their respective weights
    return inner_product(input.begin(), input.end(), weights.begin(), 0.0f);

    // Note that the output from this function isn't ready to be passed onto
    // the next layer yet - this weighted sum needs to be (optionally) biased
    // and put through the activation function - either sigmoid or threshold.
}

void FeedForwardNet::Fire()
{
    vector<float> hiddenOutput;         // A holding space for the output
    outputValues.clear();               // of the hidden layer, to pass to
    float output;                       // the output layer neurons.

    // If the size of the hidden layer is set to 0, this particular ANN class
    // recognises that to mean there is no hidden layer, and so the input
    // values should be processed directly by the output layer. When this
    // happens - i.e. there is only one round of neural processing - you have
    // a perceptron. Normally a zero-neuron hidden layer would just break the
    // network, but here we're forcing something else to happen.
    if (hidden == 0) hiddenOutput = inputValues;

    vector<Neuron>::const_iterator currentNeuron(hiddenLayer.begin());

    for (; currentNeuron != hiddenLayer.end(); ++currentNeuron) {
        output = currentNeuron->WeightedSum(inputValues);

        if (biasNode) {
            // All this bit does is pick the last value on the current
            // neuron's vector of weights and subtract it from the
            // weighted sum:
            output += currentNeuron->weights.back();
            // The idea is that rather than insist on a neuron coming up with
            // outputs that are simply positive or negative, we might have a
            // neuron which produces a high valued output which we want to
            // tone down. So we subtract the bias value, which is just stored
            // at the end of the weights for convenience, from the neuron's
            // output. The end result is a much more flexible network.
        }

        // Now the value is processed by either a threshold function, which
        // will make our hard-computed output into nothing more than a 0 or 1,
        // or a sigmoid function which will make values <0 a bit lower, and
        // values >0 a bit higher. The net would work without this stage, but
        // not very well.
        output = ActivationFunction(output);
```

```
        // Now we store the outputs of each neuron ready to be passed onto the
        // next layer.
        hiddenOutput.push_back(output);
    }

    // Now we do exactly the same thing over again for the output layer. Note
    // that in this net we will never have more than one hidden layer, so I've
    // simply written out the firing code twice, once for each layer. Many-
    // layered ANN implementations would most likely do this bit as a loop,
    // with one iteration per layer.
    for (currentNeuron = outputLayer.begin();
         currentNeuron != outputLayer.end(); ++currentNeuron) {
        output = currentNeuron->WeightedSum(hiddenOutput);

        if (biasNode) {
            output += currentNeuron->weights.back();
        }

        output = ActivationFunction(output);

        // These are the values we'll be giving to the user when they request
        // the output with GetOutput();
        outputValues.push_back(output);

    }
}

float FeedForwardNet::ActivationFunction(float n)
{
    if (sigmoid) {
        // This is the sigmoid function, refer to the lecture notes or a good
        // book on ANNs for a better diagram than this.
        //
        //   1 |       .-'''    The sigmoid function is somewhere between an
        //     |      /         identity function (i.e. returns whatever you
        //     |     /          put in) and a threshold (returns 0 for <= 0,
        //   0 |__.-'_______    1 for > 0). When plotted it's a smooth curve,
        //     -10    0   10    just like the one to the left.
        //
        return static_cast<float>(1.0 / (1.0 + exp(-n / FFN_ACTIVATION_RESPONSE)));
    }
    else {
        // This is just a threshold function. You'll want to use sigmoid for
        // most ANN applications, this is really more use for testing, e.g.
        // with the XOR problem (although of course that can be done with
        // sigmoid too)
        return n > 0.0f ? 1.0f : 0.0f;
    }
}

void FeedForwardNet::Serialise(ostream& out)const
{
    out << "FeedForwardNet\n"
        << setprecision(36)
        << inputs << endl
        << outputs << endl
        << hidden << endl
        << (sigmoid ? "sigmoid" : "threshold") << endl
        << (biasNode ? "biasnode" : "nobiasnode") << endl;

    ostream_iterator<float> outIter(out, "\n");
    copy(inputValues.begin(), inputValues.end(), outIter);
    copy(outputValues.begin(), outputValues.end(), outIter);

    vector<float> config = GetConfiguration();
    copy(config.begin(), config.end(), outIter);
}
```

```
void FeedForwardNet::Unserialise(istream& in)
{
    string name;
    in >> name;
    if (name != "FeedForwardNet") {
        throw SerialException(SERIAL_ERROR_WRONG_TYPE, name,
                              "This object is type FeedForwardNet");
    }

    in >> inputs
       >> outputs
       >> hidden
       >> switcher("sigmoid", sigmoid)
       >> switcher("biasnode", biasNode);

    Init(inputs, outputs, hidden, sigmoid, biasNode); // Looks a little fishy

    copy_from_istream(inputValues.begin(), inputValues.end(), in);
    copy_from_istream(outputValues.begin(), outputValues.end(), in);

    vector<float> config(GetConfigurationLength());
    copy_from_istream(config.begin(), config.end(), in);

    SetConfiguration(config);
}

string FeedForwardNet::ToString()const
{
    // I'm not going to comment this because, well, it's just too boring.

    ostringstream out;
    vector<Neuron>::const_iterator i;
    vector<float>::const_iterator j;

    out << setprecision(2) << setiosflags(ios::fixed)
        << "Input values:" << endl;

    for (j = inputValues.begin(); j != inputValues.end(); ++j) {
        out << setw(FFN_COLSIZE) << *j;
    }

    out << endl << endl
        << "Hidden layer weights: " << endl;

    for (i = hiddenLayer.begin(); i != hiddenLayer.end(); ++i) {
        for (j = i->weights.begin(); j != i->weights.end() - 1; ++j) {
            out << setw(FFN_COLSIZE) << *j;
        }
        if (biasNode) out << " bias: ";
        out << setw(FFN_COLSIZE) << *j << endl;
    }

    out << endl
        << "Output layer weights: " << endl;

    for (i = outputLayer.begin(); i != outputLayer.end(); ++i) {
        for (j = i->weights.begin(); j != i->weights.end() - 1; ++j) {
            out << setw(FFN_COLSIZE) << *j;
        }
        if (biasNode) out << " bias: ";
        out << setw(FFN_COLSIZE) << *j << endl;
    }

    out << endl
        << "Output values: " << endl;
```

```
    for (j = outputValues.begin(); j != outputValues.end(); ++j) {
        out << setw(FFN_COLSIZE) << *j;
    }

    out << endl << endl;

    return out.str();
}

int FeedForwardNet::GetConfigurationLength()const
{
    return hidden * inputs + outputs * hidden
           + (biasNode ? 1 : 0) * (hidden + outputs);
}

ostream& operator<< (ostream& out, const FeedForwardNet& ffn)
{
    ffn.Serialise(out);
    return out;
}

istream& operator>> (istream& in, FeedForwardNet& ffn)
{
    ffn.Unserialise(in);
    return in;
}

} // namespace BEAST
```

12.2 DynamicalNet source code

```
#include "dynamicalnet.h"

using namespace std;

namespace BEAST {

DynamicalNet::DynamicalNet(int i, int o, int t, bool mi, bool mo)
{
    Init(i, o, t, mi, mo);
}

DynamicalNet::~DynamicalNet()
{
}

void DynamicalNet::Init(int i, int o, int t, bool mi, bool mo)
{
    inputs = vector<float>(i);
    outputs = vector<float>(o);
    neuronStates = vector<float>(t);
    multiInputNodes = mi;
    multiOutputNodes = mo;
    neurons.clear();

    int neuronInputs = multiInputNodes ? i : 0,
        neuronOutputs = multiOutputNodes ? o : 0,
        neuronInChl = -1,
        neuronOutChl = -1;

    for (int n = 0; n < t; ++n) {
        if (!multiInputNodes) {
            neuronInChl = n < i ? n : -1;
        }
        if (!multiOutputNodes) {
            neuronOutChl = n + (o - t);
            if (neuronOutChl < 0 || neuronOutChl >= o) {
                neuronOutChl = -1;
            }
        }
        neurons.push_back(Neuron(neuronInputs, neuronOutputs, t,
                                 neuronInChl, neuronOutChl, this));
    }
    Reset();
}

void DynamicalNet::Reset()
{
    fill(neuronStates.begin(), neuronStates.end(), 0.0f);
}

void DynamicalNet::SetInputChannel(int neuron, int channel)
{
    if (multiInputNodes || channel < 0
        || channel > static_cast<int>(inputs.size())) return;

    vector<Neuron>::iterator i = neurons.begin();
    for (; i != neurons.end(); ++i) {
        if (i->inputChannel == channel) {
            i->inputWeights.clear();
        }
    }

    neurons[neuron].inputChannel = channel;
    neurons[neuron].inputWeights = vector<float>(1);
}
```

```
void DynamicalNet::SetOutputChannel(int neuron, int channel)
{
    if (multiInputNodes || channel < 0
        || channel > static_cast<int>(outputs.size())) return;

    vector<Neuron>::iterator i = neurons.begin();
    for (; i != neurons.end(); ++i) {
        if (i->outputChannel == channel) {
            i->outputWeights.clear();
        }
    }

    neurons[neuron].outputChannel = channel;
    neurons[neuron].outputWeights = vector<float>(1);
}

void DynamicalNet::Randomise()
{
    for_each(neurons.begin(), neurons.end(), mem_fun_ref(&Neuron::Randomise));
}

void DynamicalNet::Neuron::Randomise()
{
    generate(inputWeights.begin(), inputWeights.end(), RandomNum);
    generate(outputWeights.begin(), outputWeights.end(), RandomNum);
    generate(weights.begin(), weights.end() - 1, RandomNum);
    bias = *(weights.end() - 2);
    timeConstant = randval(69.0f) + 1.0f;
    weights.back() = static_cast<float>(log(static_cast<double>(timeConstant)));
}

void DynamicalNet::Fire()
{
    // Clear the output values
    fill(outputs.begin(), outputs.end(), 0.0f);

    // Call Fire on every neuron
    for_each(neurons.begin(), neurons.end(), mem_fun_ref(&Neuron::Fire));

    // Store the output value of each neuron for next time.
    transform(neurons.begin(), neurons.end(), neuronStates.begin(), mem_fun_ref(&Neuron::GetOutput));
}

void DynamicalNet::Neuron::Fire()
{
    // Start off with the negative of last round's activation.
    float deltaActivation = -activation;

    // Add weighted sum of the other neurons' **output** values
    // (output value = sigmoid(activation - bias)
    deltaActivation += inner_product(parent->neuronStates.begin(),
                                     parent->neuronStates.end(),
                                     weights.begin(), 0.0f);

    // Apply any input values
    // ... if there is no particular input channel,
    if (inputChannel == -1) {
        // ... but we have input weights:
        if (!inputWeights.empty()) {
            // Add a weighted sum of the current inputs and this neuron's
            // input weights.
            deltaActivation += inner_product(parent->inputs.begin(),
                                             parent->inputs.end(),
                                             inputWeights.begin(), 0.0f);
        }
    }
```

```
    // ... if only one input channel goes to this node:
    else {
        // Add the unweighted input value.
        deltaActivation += parent->inputs[inputChannel];
    }

    // Divide by the time constant
    deltaActivation /= timeConstant;

    // And add to the previous activation
    activation += deltaActivation;

    // Bias and squash
    output = Sigmoid(activation - bias);

    // Send output values (if this or all neurons are output neurons)
    // ... if there is no particular output channel,
    if (outputChannel == -1) {
        // ... but we have output weights:
        if (!outputWeights.empty()) {
            // Add the neuron's output to each output channel, weighted by
            // the neuron's output weights.
            vector<float>::iterator i = parent->outputs.begin(),
                                    j = outputWeights.begin();
            for (; i != parent->outputs.end(); ++i, ++j) {
                *i += *j * output;
            }
        }
    }
    // ... or for just one output neuron:
    else {
        parent->outputs[outputChannel] += output;
    }
}

vector<float> DynamicalNet::GetConfiguration()const
{
    vector<float> config;
    vector<Neuron>::const_iterator i = neurons.begin();
    for (; i != neurons.end(); ++i) {
        i->GetConfiguration(config);
    }

    return config;
}

void DynamicalNet::Neuron::GetConfiguration(vector<float>& config)const
{
    copy(inputWeights.begin(), inputWeights.end(), back_inserter(config));
    copy(outputWeights.begin(), outputWeights.end(), back_inserter(config));
    copy(weights.begin(), weights.end(), back_inserter(config));
}

void DynamicalNet::SetConfiguration(const vector<float>& config)
{
    vector<Neuron>::iterator i = neurons.begin();
    vector<float>::const_iterator j = config.begin();

    do {
        j = i->SetConfiguration(j);
    } while (++i != neurons.end());
}

vector<float>::const_iterator
DynamicalNet::Neuron::SetConfiguration(vector<float>::const_iterator config)
{
    // First come the input weights...
```

```
    if (inputChannel == -1) {
        copy(config, config + inputWeights.size(), inputWeights.begin());
        config += inputWeights.size();
    }
    // ... or the input weight...
    else if (!inputWeights.empty()) {
        inputWeights.front() = *config;
        ++config;
    }

    // Then the output weights...
    if (outputChannel == -1) {
        copy(config, config + outputWeights.size(), outputWeights.begin());
        config += outputWeights.size();
    }
    // ... or just one output weight...
    else if (!outputWeights.empty()) {
        outputWeights.front() = *config;
        ++config;
    }

    // Then the remainder are the internal weights
    copy(config, config + weights.size(), weights.begin());

    bias = *(weights.end() - 2);
    timeConstant = static_cast<float>(exp(static_cast<double>(weights.back())));

    if (timeConstant < 1.0f) {
        timeConstant = 1.0f + 2 * (1.0f - timeConstant);
        weights.back() = static_cast<float>(log(static_cast<double>(timeConstant)));
    }

    return config + weights.size();
}

void DynamicalNet::Serialise(ostream& out)const
{
    out << "DynamicalNet\n"
        << setprecision(36)
        << static_cast<int>(inputs.size()) << endl
        << static_cast<int>(outputs.size()) << endl
        << static_cast<int>(neurons.size()) << endl
        << (multiInputNodes ? "multi_in" : "single_in") << endl
        << (multiOutputNodes ? "multi_out" : "single_out") << endl;

    vector<float> config = GetConfiguration();
    copy(config.begin(), config.end(), ostream_iterator<float>(out, "\n"));
}

void DynamicalNet::Unserialise(istream& in)
{
    string name;
    in >> name;
    if (name != "DynamicalNet") {
        throw SerialException(SERIAL_ERROR_WRONG_TYPE, name,
                              "This object is type DynamicalNet");
    }

    int i, o, t;
    bool mi, mo;

    in >> i >> o >> t >> switcher("multi_in", mi) >> switcher("multi_out", mo);
    Init(i, o, t, mi, mo);

    vector<float> config(GetConfigurationLength());
    copy_from_istream(config.begin(), config.end(), in);
    SetConfiguration(config);
```

```
}

string DynamicalNet::ToString()const
{
    ostringstream out;
    ostream_iterator<float> outIter(out, " ");

    out << "Input values:" << endl;
    copy(inputs.begin(), inputs.end(), outIter);
    out << endl << "Output values:" << endl;
    copy(outputs.begin(), outputs.end(), outIter);
    out << endl << "Activation states:" << endl;
    copy(neuronStates.begin(), neuronStates.end(), outIter);
    out << endl << "Neurons:" << endl;
    transform(neurons.begin(), neurons.end(),
              ostream_iterator<string>(out, "\n"),
              mem_fun_ref(&Neuron::ToString));
    out << endl;

    return out.str();
}

string DynamicalNet::Neuron::ToString()const
{
    ostringstream out;
    ostream_iterator<float> outIter(out, " ");

    if (!inputWeights.empty()) {
        out << "Input weight(s):" << endl;
        copy(inputWeights.begin(), inputWeights.end(), outIter);
        out << endl;
    }
    out << "Hidden layer weight(s):" << endl;
    copy(weights.begin(), weights.end() - 2, outIter);
    out << endl << "Bias: " << bias << " Time constant: " << timeConstant
        << endl;
    if (!outputWeights.empty()) {
        out << "Output weight(s):" << endl;
        copy(outputWeights.begin(), outputWeights.end(), outIter);
        out << endl;
    }

    return out.str();
}

int DynamicalNet::GetConfigurationLength()const
{
    int numNeurons = static_cast<int>(neurons.size()),
        numInputs  = static_cast<int>(inputs.size()),
        numOutputs = static_cast<int>(outputs.size());

    return numNeurons * numNeurons
        + (multiInputNodes ? numNeurons * numInputs : 0)
        + (multiOutputNodes ? numNeurons * numOutputs : 0);
}

ostream& operator<<(ostream& out, const DynamicalNet& dnn)
{
    dnn.Serialise(out);
    return out;
}

istream& operator>>(istream& in, DynamicalNet& dnn)
{
    dnn.Unserialise(in);
    return in;
}
```

```
} // namespace BEAST
```

12.3 Bacterium source code

```
#include "bacterium.h"

using namespace std;

namespace BEAST {

Bacterium::Bacterium():
reproductionCost(0.4),
energyRate(0.005),
sporeEnergyRate(0.01),
attractantCost(0.01), repellentCost(0.01),
deathThreshold(0.0),
tumbleTime(10.0), tumbleScale(10.0),
reproductionThreshold(0.4),
sporulationThreshold(0.25),
consumptionRate(0.1), attractantRate(0.5), repellentRate(4.0),
swarmRadius(20.0), swarmInfluence(0.5),
gradientInfluence(0.8),
nutrientResponse(0.8), attractantResponse(0.8), repellentResponse(0.8),
attractantThreshold(0.5), repellentThreshold(0.5),
nutrientDist(NULL), attractantDist(NULL),
repellentDist(NULL), trailDist(NULL),
energy(1.0), totalEnergy(0.1), lastNutrient(0.0),
currentNutrient(0.0), currentAttractant(0.0), currentRepellent(0.0),
nextCheck(0), swarmSize(0)
{
    SetInitRandom(true);
    SetSpeed(40.0);
}

Bacterium::~Bacterium()
{
    for_each(offspring.begin(), offspring.end(), deleter<Bacterium>());
}

void Bacterium::Update()
{
    CheckBoundary();
    ReadDistributions();
    UpdateDistributions();

    SetVelocity((1.0 - gradientInfluence) * GetVelocity() +
             gradientInfluence *
             ( nutrientResponse * GetNutrientGradient()
             + attractantResponse * GetAttractantGradient()
             - repellentResponse * GetRepellentGradient()));
    SetVelocity((1.0 - swarmInfluence) * GetVelocity()
             + swarmInfluence * GetSwarmVelocity());

    GetVelocity().SetLength(GetMaxSpeed());

    UpdateEnergy();

    // Apply velocity changes and clean up.
    FinishUpdate();
}

void Bacterium::CheckBoundary()
{
    if (GetLocation().x <= 0.0) {
        SetVelocityX(-GetVelocity().x);
        SetLocationX(0.0); }
    if (GetLocation().x >= GetWorld().GetWidth()) {
        SetVelocityX(-GetVelocity().x);
        SetLocationX(GetWorld().GetWidth() - 1.0); }
```

```
    if (GetLocation().y <= 0.0) {
        SetVelocityY(-GetVelocity().y);
        SetLocationY(0.0); }
    if (GetLocation().y >= GetWorld().GetHeight()) {
        SetVelocityY(-GetVelocity().y);
        SetLocationY(GetWorld().GetHeight() - 1.0); }
}

void Bacterium::ReadDistributions()
{
    if (nutrientDist != NULL) currentNutrient = nutrientDist->GetDensity(GetLocation());
    if (attractantDist != NULL) currentAttractant = attractantDist->GetDensity(GetLocation());
    if (repellentDist != NULL) currentRepellent = repellentDist->GetDensity(GetLocation());
}

void Bacterium::UpdateDistributions()
{
    if (nutrientDist != NULL) {

        double amount = consumptionRate <= currentNutrient ? consumptionRate : currentNutrient;
        energy += amount;
        totalEnergy += amount;
        nutrientDist->AddDensity(GetLocation(), static_cast<DistReal>(-amount));

        if  (!isSpore) {
            if (currentNutrient >= attractantThreshold) ReleaseAttractant();
            if (currentNutrient < repellentThreshold) ReleaseRepellent();
        }
    }

    if (trailDist != NULL) {
        if (trailDist->GetDensity(GetLocation()) <= 0.0)
            trailDist->SetDensity(GetLocation(), 0.5);
    }
}

void Bacterium::ReleaseAttractant()
{
    if (attractantDist == NULL) return;

    double amount = currentNutrient * attractantRate;
    double cost = amount * attractantCost;

    if (energy >= cost) energy -= cost;
    else {
        amount = energy / attractantCost;
        energy = 0.0;
    }

    attractantDist->AddDensity(GetLocation(), static_cast<DistReal>(amount));
}

void Bacterium::ReleaseRepellent()
{
    if (repellentDist == NULL) return;

    double amount = (repellentThreshold - (repellentThreshold - currentNutrient))
                  / repellentThreshold * repellentRate;
    double cost = amount * repellentCost;

    if (energy >= cost) energy -= cost;
    else {
        // If the amount is too expensive, release as much as possible.
        amount = energy / repellentCost;
        energy = 0.0;
    }
```

```
    repellentDist->AddDensity(GetLocation(), static_cast<DistReal>(amount));
}

Vector2D Bacterium::GetSwarmVelocity()
{
    return swarmSize > 0
        ? swarmTotalVel *= 1.0 / static_cast<double>(swarmSize)
        : GetVelocity();
}

Vector2D Bacterium::GetTumblingVelocity()
{
    --nextCheck;
    if (nextCheck <= 0) {
        SetNextCheck();
        if (nextCheck <= 0) tumblingVelocity.SetAngle(randval(TWOPI));
    }
    return tumblingVelocity;
}

Vector2D Bacterium::GetNutrientGradient()
{
    return nutrientDist != NULL
        ? nutrientDist->GetGradient(GetLocation()).GetNormalised()
        : Vector2D(0.0, 0.0);
}

Vector2D Bacterium::GetAttractantGradient()
{
    return attractantDist != NULL
        ? attractantDist->GetGradient(GetLocation()).GetNormalised()
        : Vector2D(0.0, 0.0);
}

Vector2D Bacterium::GetRepellentGradient()
{
    return repellentDist != NULL
        ? repellentDist->GetGradient(GetLocation()).GetNormalised()
        : Vector2D(0.0, 0.0);
}

void Bacterium::UpdateEnergy()
{
    if (!isSpore) energy -= energyRate * fabs(GetMaxSpeed());
    energy -= sporeEnergyRate;

    // Sporulate depending on energy.
    isSpore = energy <= sporulationThreshold;

    // Ensure energy never goes below
    if (energy <= 0.0) energy = 0.0;

    // If the energy level is below the deathThreshold, die next frame.
    if (energy <= deathThreshold) {
        if (trailDist != NULL) trailDist->SetDensity(GetLocation(), 1.0);
        SetDead(true);
    }

    // If there is enough energy to reproduce, do so.
    if (energy >= reproductionThreshold &&
        energy >= reproductionCost) Reproduce();
}

void Bacterium::FinishUpdate()
{
    // Update the location with the velocity.
    OffsetLocation(GetVelocity() * GetTimeStep());
```

```
    swarmSize = 0;
    swarmTotalVel.x = swarmTotalVel.y = 0.0;
}

void Bacterium::Interact(WorldObject* obj)
{
    if (dynamic_cast<Distribution*>(obj)) return;
    Animat::Interact(obj);
}

void Bacterium::UniInteract(WorldObject* obj)
{
    Bacterium* b;
    if (IsKindOf(obj, b) && (b->GetLocation() - GetLocation()).GetLengthSquared()
                          < (swarmRadius * swarmRadius)) {
        swarmTotalVel += b->GetVelocity();
        ++swarmSize;
    }

    Animat::UniInteract(obj);
}

void Bacterium::SetNextCheck()
{
    double  gradNutrient = currentNutrient - lastNutrient,
            gradAttractant = currentAttractant - lastAttractant,
            gradRepellent = currentRepellent - lastRepellent;

    nextCheck = static_cast<int>(tumbleTime
                                 + gradNutrient * tumbleScale * tumbleTime
                                 + gradAttractant * attractantResponse * tumbleTime
                                 - gradRepellent * repellentResponse * tumbleTime);

    lastNutrient = currentNutrient;
}

void Bacterium::Reproduce()
{
    energy -= reproductionCost;
    energy /= 2.0;
    Bacterium* baby = new Bacterium(*this);

    double o = randval(TWOPI);
    SetLocation(Vector2D(GetLocation(), GetRadius(), o));
    baby->SetLocation(Vector2D(GetLocation(), -GetRadius(), o));

    baby->Reset();
    GetWorld().Add(baby);
    offspring.push_back(baby);
}

void Bacterium::GetOffspring(std::list<Bacterium*>& babies)const
{
    std::list<Bacterium*>::const_iterator i = offspring.begin();

    for (; i != offspring.end(); ++i) {
        babies.push_back(*i);
        (*i)->GetOffspring(babies);
    }
}

std::list<Bacterium*> Bacterium::GetOffspring()const
{
    std::list<Bacterium*> result;
    GetOffspring(result);
    return result;
```

```
}

void Bacterium::GetOffspring(std::list<Bacterium const*>& babies)const
{
    std::list<Bacterium*>::const_iterator i = offspring.begin();

    for (; i != offspring.end(); ++i) {
        babies.push_back(*i);
        (*i)->GetOffspring(babies);
    }
}

string Bacterium::ToString()const
{
    ostringstream out;
    out << "N: " << currentNutrient << " A: " << currentAttractant
        << " R: " << currentRepellent << " E: " << energy
        << " T: " << totalEnergy;
    return out.str();
}

void Bacterium::OnClick()
{
    GetLogStream()
    << "Reproduction threshold: "   << reproductionThreshold << '\n'
    << "Consumption rate: "         << consumptionRate << '\n'
    << "Attractant rate: "          << attractantRate << '\n'
    << "Repellent rate: "           << repellentRate << '\n'
    << "Swarm radius: "             << swarmRadius << '\n'
    << "Swarm influence: "          << swarmInfluence << '\n'
    << "Gradient influence: "       << gradientInfluence << '\n'
    << "Nutrient response: "        << nutrientResponse << '\n'
    << "Attractant response: "      << attractantResponse << '\n'
    << "Repellent response: "       << repellentResponse << '\n'
    << "Attractant threshold: "     << attractantThreshold << '\n'
    << "Repellent threshold: "      << repellentThreshold << '\n'
    << "Radius: "                   << GetRadius() << '\n'
    << "Speed: "                    << GetSpeed() << "\n\n";
}

} // namespace BEAST
```

12.4 Braitenberg source code

```
// Vim users: for increased viewing pleasure :set ts=4

#include "beast.h"
#include "animat.h"
#include "sensor.h"

using namespace BEAST;

// Dots are just WorldObjects with the colour set to yellow and the
// radius set to 10. The constructor just allows you to specify a
// location. Dot relies on the default non-solid setting for
// WorldObject, which means that instead of bumping into Dots, Animats
// slide over them.
class Dot : public WorldObject
{
public:
    Dot(Vector2D l = Vector2D()):WorldObject(l, 0.0, 10.0){SetColour(ColourPalette[COLOUR_YELLOW]);}
    virtual ~Dot(){}

    // This simply adjusts the standard serialisation (save to file) code
    // so that the object remembers it's a 'Dot', rather than a
    // 'WorldObject' - since there are no properties in Dot which
    // WorldObject doesn't have, it's not necessary to add any new
    // serialisation code.
    IMPLEMENT_SERIALISATION("Dot", WorldObject)
};

// Braitenberg is a basic bot inherited from the Animat class, with two
// sensors representing the Braitenberg's left and right light sensors,
// a radius of 10 and a minimum speed of 0, ensuring that the Braitenberg
// does not go backwards when its control output is 0.
class Braitenberg : public Animat
{
public:
    Braitenberg()
    {
        This.Add("left", ProximitySensor<Dot>(PI/2, 75.0, -1));
        This.Add("right", ProximitySensor<Dot>(PI/2, 75.0, 1));

        This.SetInitRandom(true);

        This.SetMinSpeed(0.0);
        This.SetMaxSpeed(95.0);

        This.Radius = 10.0;
    }
};

// This is a Braitenberg with the control function overloaded so that the
// left sensor feeds the left motor, and the right sensor feeds the right
// motor.
class Braitenberg2a : public Braitenberg
{
public:
    virtual void Control()
    {
        This.Controls["left"] = This.Sensors["left"]->GetOutput();
        This.Controls["right"] = This.Sensors["right"]->GetOutput();
    }
};
#include <map>
// This is a Braitenberg with the control function overloaded so that the
// left sensor feeds the right motor, and the right sensor feeds the left
// motor.
class Braitenberg2b : public Braitenberg
```

```
{
public:
    virtual void Control()
    {
        This.Controls["left"] = This.Sensors["right"]->GetOutput();
        This.Controls["right"] = This.Sensors["left"]->GetOutput();
    }
};

// Sets up a simple simulation with a series of positioned dots and two
// Braitenberg vehicles.
class BraitenbergSimulation : public Simulation
{
    Group<WorldObject> ExampleWorld;

public:
    BraitenbergSimulation()
    {
        // We set up the ExampleWorld by filling it with instances of
        // the two types of Braitenbergs.
        ExampleWorld.push_back(new Braitenberg2a);
        ExampleWorld.push_back(new Braitenberg2b);

        // Now the Dots are added, initialised with their locations in
        // the world.
        ExampleWorld.push_back(new Dot(Vector2D(150.0, 100.0)));
        ExampleWorld.push_back(new Dot(Vector2D(200.0, 100.0)));
        ExampleWorld.push_back(new Dot(Vector2D(250.0, 100.0)));
        ExampleWorld.push_back(new Dot(Vector2D(300.0, 100.0)));
        ExampleWorld.push_back(new Dot(Vector2D(350.0, 100.0)));
        ExampleWorld.push_back(new Dot(Vector2D(400.0, 100.0)));
        ExampleWorld.push_back(new Dot(Vector2D(400.0, 150.0)));
        ExampleWorld.push_back(new Dot(Vector2D(400.0, 200.0)));
        ExampleWorld.push_back(new Dot(Vector2D(400.0, 250.0)));
        ExampleWorld.push_back(new Dot(Vector2D(400.0, 300.0)));
        ExampleWorld.push_back(new Dot(Vector2D(400.0, 350.0)));
        ExampleWorld.push_back(new Dot(Vector2D(350.0, 350.0)));
        ExampleWorld.push_back(new Dot(Vector2D(300.0, 350.0)));
        ExampleWorld.push_back(new Dot(Vector2D(250.0, 350.0)));
        ExampleWorld.push_back(new Dot(Vector2D(200.0, 350.0)));
        ExampleWorld.push_back(new Dot(Vector2D(200.0, 400.0)));
        ExampleWorld.push_back(new Dot(Vector2D(200.0, 450.0)));
        ExampleWorld.push_back(new Dot(Vector2D(200.0, 500.0)));
        ExampleWorld.push_back(new Dot(Vector2D(200.0, 550.0)));
        ExampleWorld.push_back(new Dot(Vector2D(250.0, 550.0)));
        ExampleWorld.push_back(new Dot(Vector2D(300.0, 550.0)));
        ExampleWorld.push_back(new Dot(Vector2D(350.0, 550.0)));
        ExampleWorld.push_back(new Dot(Vector2D(400.0, 550.0)));
        ExampleWorld.push_back(new Dot(Vector2D(450.0, 550.0)));
        ExampleWorld.push_back(new Dot(Vector2D(500.0, 550.0)));
        ExampleWorld.push_back(new Dot(Vector2D(550.0, 550.0)));
        ExampleWorld.push_back(new Dot(Vector2D(600.0, 550.0)));
        ExampleWorld.push_back(new Dot(Vector2D(600.0, 500.0)));
        ExampleWorld.push_back(new Dot(Vector2D(600.0, 450.0)));
        ExampleWorld.push_back(new Dot(Vector2D(600.0, 400.0)));
        ExampleWorld.push_back(new Dot(Vector2D(600.0, 350.0)));
        ExampleWorld.push_back(new Dot(Vector2D(550.0, 350.0)));
        ExampleWorld.push_back(new Dot(Vector2D(500.0, 350.0)));
        ExampleWorld.push_back(new Dot(Vector2D(500.0, 300.0)));
        ExampleWorld.push_back(new Dot(Vector2D(500.0, 250.0)));
        ExampleWorld.push_back(new Dot(Vector2D(500.0, 200.0)));
        ExampleWorld.push_back(new Dot(Vector2D(500.0, 150.0)));
        ExampleWorld.push_back(new Dot(Vector2D(500.0, 100.0)));
        ExampleWorld.push_back(new Dot(Vector2D(500.0,  50.0)));

        // All the objects in our ExampleWorld Group have now been
        // configured and can now be added to the World.
```

```
        Add("ExampleWorld", ExampleWorld);

        // Because we just want the braitenbergs to go on indefinitely,
        // the time limit is set to -1.
        SetTimeSteps(-1);
    }
};
```

12.5 Shrew source code

```
// Vim users: for increased viewing pleasure :set ts=4

#include "animat.h"
#include "sensor.h"

// For more information on this file, take a look at Tutorial 1 in the
// BEAST documentation.

using namespace BEAST;

class Shrew : public Animat // Shrew is derived from Animat
{
public:
    Shrew()
    {
        This.Add("left", ProximitySensor<Shrew>(PI/5, 200.0, -PI/20));
        This.Add("right", ProximitySensor<Shrew>(PI/5, 200.0, PI/20));

        This.SetInitRandom(true);   // Start in random locations

        This.Radius = 10.0;         // Shrews are a little bigger than usual
    }
    virtual ~Shrew(){}

    virtual void Control()
    {
        This.Controls["left"] = This.Sensors["right"]->GetOutput();
        This.Controls["right"] = This.Sensors["left"]->GetOutput();
    }
};

class ShrewSimulation : public Simulation
{
    Group<Shrew> grpShrew;

public:
    ShrewSimulation():
    grpShrew(30)
    {
        This.Add("Shrews", This.grpShrew);
    }
};
```

12.6 Mouse source code

```
// Vim users: for increased viewing pleasure :set ts=4

#include "neuralanimat.h"
#include "sensor.h"
#include "population.h"

using namespace std;
using namespace BEAST;

// A Cheese is a small, yellow WorldObject which changes position when
// the Eaten method is called on it (i.e. a Mouse collides with it)
class Cheese : public WorldObject
{
public:
    Cheese()
    {
        This.Radius = 2.5f;                         // Cheeses are quite small
        This.SetColour(ColourPalette[COLOUR_YELLOW]);// Cheeses are yellow
        This.InitRandom = true;                     // Cheases are scattered
    }
    virtual ~Cheese(){}

    // When a Cheese is Eaten, it reappears in a random location.
    void Eaten()
    {
        This.Location = This.MyWorld->RandomLocation();
    }
};

// This Mouse uses a simple calculation to forage for cheese.
class Mouse : public Animat
{
public:
    Mouse()
    {
        This.Add("angle", NearestAngleSensor<Cheese>());
        This.InitRandom = true;
    }

    // This incredibly simple control function is all a Mouse needs to
    // forage successfully - can our neural nets do as well?
    virtual void Control()
    {
        double o = This.Sensors["angle"]->GetOutput();
        This.Controls["right"] = 0.5 - (o > 0.0 ? o : 0.0);
        This.Controls["left"] = 0.5 + (o < 0.0 ? o : 0.0);
    }

    // This is called when a Mouse collides with any object in the World.
    // If the object is a Cheese, the Eaten method is called on Cheese.
    virtual void OnCollision(WorldObject* obj)
    {
        Cheese* cheese;

        if (IsKindOf(obj,cheese)) {
            cheese->Eaten();
        }

        Animat::OnCollision(obj);
    }
};

// This mouse uses a FeedForwardNet to locate the nearest cheese using a
// NearestAngleSensor. No GA or other learning algorithm is involved.
class NeuralMouse : Mouse
```

```
{
public:
    NeuralMouse(): myBrain(1, 2, 2, false)
    {
        vector<float> ffnConfig;
        ffnConfig.push_back(-1.0f); // 1st input 1st weight
        ffnConfig.push_back( 0.0f); // 1st input bias
        ffnConfig.push_back( 1.0f); // 2nd input 1st weight
        ffnConfig.push_back( 0.0f); // 2nd input bias

        ffnConfig.push_back( 0.0f); // 1st hidden 1st weight (for input 1)
        ffnConfig.push_back( 1.0f); // 1st hidden 2nd weight (for input 2)
        ffnConfig.push_back(-0.5f); // 1st hidden bias
        ffnConfig.push_back( 1.0f); // 2nd hidden 1st weight (for input 1)
        ffnConfig.push_back( 0.0f); // 2nd hidden 2nd weight (for input 2)
        ffnConfig.push_back(-0.5f); // 2nd hidden bias

        This.myBrain.SetConfiguration(ffnConfig);
    }

    // Here the single sensor output is set as the input for the FFN, the
    // FFN is fired once and the two controls are set to the output values.
    // Note that the original Mouse::Control method is overridden by this one.
    virtual void Control()
    {
        This.myBrain.SetInput(0, This.Sensors["angle"]->GetOutput());

        This.myBrain.Fire();

        This.Controls["left"] = myBrain.GetOutput(0);
        This.Controls["right"] = myBrain.GetOutput(1);
    }

private:
    FeedForwardNet myBrain;
};

// In an ideal world, EvoMouse would inherit from Mouse, thereby getting
// the same OnCollision function and initialisation code as Mouse, but
// it's more convenient to inherit from EvoFFNAnimat which gives us all
// the GA and FFN code. Multiple inheritance would be another option, but
// introduces a host of other unwanted complications.
class EvoMouse : public EvoFFNAnimat
{
public:
    EvoMouse(): cheesesFound(0)
    {
        This.Add("angle", NearestAngleSensor<Cheese>());
// An alternative to the NearestAngleSensor is the Proximity Sensor, which
// gives less precise directional information, but does let the mouse know
// how far away the cheese is.
//      This.Add("proximity", ProximitySensor<Cheese>(PI/8, 80.0, 0.0));
        This.InitRandom = true;
        This.InitFFN(4);
    }

    // This is identical to the OnCollision method for Mouse, except here we
    // are also recording the number of cheeses eaten.
    virtual void OnCollision(WorldObject* obj)
    {
        Cheese* cheese;

        if (IsKindOf(obj,cheese)) {
            cheesesFound++;
            cheese->Eaten();
        }
```

```
        EvoFFNAnimat::OnCollision(obj);
    }

    // The EvoMouse's fitness is the amount of cheese collected, divided by
    // the power usage, so a mouse is penalised for simply charging around
    // as fast as possible and randomly collecting cheese - it needs to find
    // its target carefully.
    virtual float GetFitness()const
    {
        return This.cheesesFound > 0 ? static_cast<float>(This.cheesesFound) / This.DistanceTravelled.as<float>() : 0;
    }

    // Overloading the ToString method allows us to output a small amount of
    // information which is visible in the status bar of the GUI when a
    // mouse is clicked on.
    virtual string ToString()const
    {
        ostringstream out;
        out << " Power used: " << This.PowerUsed;
        return out.str();
    }

private:
    int cheesesFound;   // The number of cheeses collected for this run.
};

class MouseSimulation : public Simulation
{
    GeneticAlgorithm<EvoMouse> theGA;
    Population<EvoMouse> theMice;
//  Group<Mouse> theMice;
    Group<Cheese> theCheeses;

public:
    MouseSimulation():
    theGA(0.7f, 0.05f), // Crossover probability of 0.7, mutation probability of 0.05
//  theMice(30),        // 30 mice are in the population.
    theMice(30, theGA), // 30 mice are in the population.
    theCheeses(30)      // 30 cheeses are around at one time.
    {
        // We're using a rank selection method. Consult the BEAST
        // documentation for GeneticAlgorithm, the ar23 course slides or
        // a good book on GAs for more details.
        This.theGA.SetSelection(GA_RANK);
        // The ranking selection pressure is set to 2.
        This.theGA.SetParameter(GA_RANK_SPRESSURE, 2.0);

        This.SetTimeSteps(100);

        This.Add("Mice", This.theMice);
        This.Add("Cheeses", This.theCheeses);
    }
};
```

12.7 Predator/Prey source code

```
// Vim users: for increased viewing pleasure :set ts=4

#include "neuralanimat.h"
#include "sensor.h"
#include "population.h"

using namespace std;
using namespace BEAST;

// Forward declaration for Prey
class Predator;

// A Prey is a small Animat which can be evolved with a GA, uses a
// feed forward neural network and has two Predator-detecting sensors.
// It records the number of times it has been eaten and the inverse of
// this number becomes its fitness score.
class Prey : public EvoFFNAnimat
{
public:
    // Prey is initialised with 1 for its timesEaten score, since otherwise
    // we might end up dividing by zero in the fitness function.
    Prey(): timesEaten(1)
    {
        // Two Predator-detecting sensors are added.
        Add("right", ProximitySensor<Predator>(PI/1.05, 100.0, -PI/2));
        Add("left", ProximitySensor<Predator>(PI/1.05, 100.0, PI/2));

        // The FeedForwardNet is initialised with four hidden nodes - the
        // number of inputs and outputs is decided by the number of sensors
        // (2) and the number of controls (2). The InitFFN function determines
        // these values and configures the network accordingly.
        InitFFN(4);
        SetInitRandom(true);
        This.MinSpeed = 0;
        This.MaxSpeed = 100;
    }

    // As with Cheeses, when a Predator collides with a Prey, the Eaten
    // method is called and the prey is repositioned somewhere random
    // in the World.
    void Eaten()
    {
        timesEaten++;
        This.Location = static_cast<World&>(This.MyWorld).RandomLocation();
    }

    float GetFitness()const
    {
        return 1.0f / static_cast<float>(timesEaten);
    }

    // This just lets us track the performance of individual Prey in the GUI.
    virtual string ToString()const
    {
        ostringstream out;
        out << "Times eaten: " << timesEaten << " Current fitness: " << GetFitness();
        return out.str();
    }

private:
    int timesEaten;
};

// The Predator class is an Animat which can be evolved in a GA and uses a
// feed forward neural network. It's fitness score depends on how many
```

```
// Prey it manages to collide with.
class Predator : public EvoFFNAnimat
{
public:
    Predator():preyEaten(0)
    {
        Add("left", ProximitySensor<Prey>(PI/5, 200.0, -PI/20));
        Add("right", ProximitySensor<Prey>(PI/5, 200.0, PI/20));

        InitFFN(4);
        SetInitRandom(true);

        This.MinSpeed = 0;
        This.MaxSpeed = 100;
        This.Radius = 10.0;
    }

    void OnCollision(WorldObject* obj)
    {
        Prey* ptr;

        if (IsKindOf(obj,ptr)) {
            preyEaten++;
            ptr->Eaten();
        }

        FFNAnimat::OnCollision(obj);
    }

    float GetFitness()const { return preyEaten; }

    virtual string ToString()const
    {
        ostringstream out;
        out << "Meals had: " << preyEaten << " Current fitness: " << GetFitness();
        return out.str();
    }

private:
    int preyEaten;
};


class ChaseSimulation : public Simulation
{
    GeneticAlgorithm<Predator> gaPred;
    GeneticAlgorithm<Prey> gaPrey;
    Population<Predator> popPred;
    Population<Prey> popPrey;

public:
    ChaseSimulation():
    gaPred(0.7f, 0.1f), gaPrey(0.7f, 0.1f),
    popPred(30,gaPred), popPrey(30,gaPrey)
    {
        gaPred.SetSelection(GA_RANK);
        gaPred.SetParameter(GA_RANK_SPRESSURE, 2.0);

        gaPrey.SetSelection(GA_RANK);
        gaPrey.SetParameter(GA_RANK_SPRESSURE, 2.0);

        // Although the population size for both Predator and Prey is
        // 30, it would be chaos if we were trying to assess 60 individuals
        // at once (although it still works after a fashion and takes less
        // time - remove these two lines to see that happening). Instead
        // 5 Predators are assessed against 10 Prey. There are 30 assessments,
        // and each time the Simulation class makes sure that the next 5
```

```
        // Prey and the next 10 Predators are picked.
        popPred.SetTeamSize(5);
        popPrey.SetTeamSize(10);
        SetAssessments(30);

        Add("Predators", popPred);
        Add("Prey", popPrey);
    }
};
```

12.8 Bacteria example source code

```
// Vim users: for increased viewing pleasure :set ts=4

#include "wxsimenv.h"
#include "simulation.h"
#include "bacteria.h"

using namespace std;
using namespace BEAST;

class TestBacterium : public Bacterium
{
public:
    TestBacterium() {
        // These variables should remain constant for the whole simulation.
        // They represent universal constraints within which the bacteria must
        // survive. You are free to alter them, but consider each time how it
        // affects the realism of the simulation. Note that for our purposes,
        // the most realistic values will not necessarily give the best
        // results, and also that the values below are neither realistic nor
        // effective.
        SetReproductionCost(0.4);   // Amount of energy lost in reproduction
        SetEnergyRate(0.005);       // Rate of energy use (* speed)
        SetSporeEnergyRate(0.01);   // Energy used whether moving or not
        SetAttractantCost(0.01);    // Cost to release one unit of attractant
        SetRepellentCost(0.01);     // Cost to release one unit of repellent
        SetDeathThreshold(0.0);     // Minimum energy before bacterium dies

        // These variables might change as a result of a genetic algorithm, or
        // your own experimentation. In most cases, an effect can be switched
        // off by setting the relevant variable to 0.0.
        SetReproductionThreshold(10.0); // The amount of energy needed before
                                        // the bacterium reproduces (divides)
        SetSporulationThreshold(0.25);  // Energy level at sporulation
        SetConsumptionRate(0.1);    // Maximum food consumed per timestep
        SetAttractantRate(0.5);     // Amount of attractant released/food unit
        SetRepellentRate(4.0);      // Rate of repellent/inverse of food
        SetSwarmRadius(0.0);        // Radius in which velocities are summed
        SetSwarmInfluence(0.0);     // Degree of influence of swarm velocity
        SetGradientInfluence(1.0);  // Degree of influence of all gradients
        SetNutrientResponse(1.0);   // Relative change in direction wrt
        SetAttractantResponse(0.0); // nutrient, attractant and repellent
        SetRepellentResponse(0.0);  // respectively
        SetAttractantThreshold(0.5);// Minimum food to trigger attractant
        SetRepellentThreshold(0.5); // Minimum food to trigger repellent

        // This decides the amount of energy bacteria start with. If you find
        // your bacteria dying as soon as the simulation starts, it might be
        // worth increasing this, but if their consumption rate does not allow
        // them to reach equilibrium with the amount of food available it will
        // usually be worth adjusting the above variables instead.
        SetEnergy(1.0);

        // The speed is always constant, and influences energy consumption.
        SetSpeed(40.0);
    }

    virtual ~TestBacterium() {}
};

// Sets up a few bacteria which demonstrate their gradient-following ability.
class BacteriaGradientSim : public Simulation
{
protected:
    // An object to help us put bacteria into the World.
    Group<TestBacterium>    grpBacteria;
```

```
    // An object to help us put distributions into the World.
    Group<WorldObject>      grpWorld;
    // Pointers to Distributions, so we can keep track of them for the
    // duration of the simulation.
    Distribution            *nutrient,
                            *trail;
public:
    // Constructor for our Simulation, in which Distributions are set up and
    // everything is put into the World.
    BacteriaGradientSim():
    grpBacteria(30)
    {
        GetWorld().Toggle(DISPLAY_COLLISIONS);
        GetWorld().Toggle(DISPLAY_TRAILS);
        GetWorld().Toggle(DISPLAY_MONITOR);
        GetWorld().Toggle(DISPLAY_SENSORS);

        // Set up a new 400 x 300 cell distribution.
        nutrient = new Distribution(400, 300, 1);
        nutrient->SetColour(1.0f, 1.0f, 1.0f); // Set the colour to white.
        nutrient->SetDiffusionSpeed(1); // Make it diffuse once per timestep.
        nutrient->Plot(0.5);            // Set each point to 0.5.
        // Add a Gaussian blob to give the bacteria a gradient to climb.
        nutrient->Filter(plus<double>(), Gaussian2D(200, 150, 30.0, 5000.0));
        nutrient->SetMaxConc(2.0);      // Max display density is 3.
        grpWorld.push_back(nutrient);   // Put it into the grpWorld object.

        // Set up a new 800 x 600 distribution.
        trail = new Distribution(800, 600, 1);
        trail->SetColour(1.0f, 1.0f, 1.0f); // Set the colour to orange.
        trail->Plot(0.0);               // Zero the distribution.
        grpWorld.push_back(trail);      // Put it into the grpWorld object.

        Add("Bacteria", grpBacteria);   // Add the bacteria to the World.
        Add("Distributions", grpWorld); // Add the distributions to the World.

        // Ignore the funny syntax; this just gives each Bacterium a pointer
        // to the nutrient and trail distributions.
        grpBacteria.ForEach<Distribution*>(&Bacterium::SetNutrientDist, nutrient);
        grpBacteria.ForEach<Distribution*>(&Bacterium::SetTrailDist, trail);

        // This makes the first 'assessment' go on for ever.
        SetTimeSteps(-1);
    }

};

// Now the bacteria will be adjusted to demonstrate swarming.
class BacteriaSwarmSim : public BacteriaGradientSim
{
public:
    BacteriaSwarmSim()
    {
        grpBacteria.ForEach<double>(&Bacterium::SetSwarmRadius, 20.0);
        grpBacteria.ForEach<double>(&Bacterium::SetSwarmInfluence, 0.5);
    }
};

class BacteriaRepellentSim : public BacteriaSwarmSim
{
protected:
    // A pointer to our repellent object.
    Distribution *repellent;

public:
    BacteriaRepellentSim()
    {
```

```
        // Make and add a new, nasty pink distribution for repellent.
        repellent = new Distribution(400, 300, 1);
        repellent->SetColour(1.0f, 0.5f, 0.5f);
        repellent->SetDiffusionSpeed(2);
        repellent->Plot(0.0);
        repellent->SetMaxConc(1.5f);
        nutrient->SetDiffusionSpeed(1); // Make it diffuse like nutrients.
        grpWorld.push_back(repellent);

        // Tell the bacteria about it.
        grpBacteria.ForEach<Distribution*>(&Bacterium::SetRepellentDist, repellent);

        // Make them respond to repellents.
        grpBacteria.ForEach<double>(&Bacterium::SetRepellentResponse, 0.5);

        // Just for a laugh, plot some noise, keep the bacteria guessing...
        nutrient->Filter(plus<double>(), GaussianNoise(0.0, 0.5));
    }
};

// Now we have reproduction, repellent and swarming.
class BacteriaReproSim : public BacteriaRepellentSim
{
public:
    BacteriaReproSim()
    {
        // In principle the bacteria could have reproduced in the previous
        // simulations, but the reproduction threshold was too high.
        grpBacteria.ForEach<double>(&Bacterium::SetReproductionThreshold, 0.4);
        grpBacteria.ForEach<double>(&Bacterium::SetConsumptionRate, 0.2);

    }
};

// Now we have reproduction, repellent and swarming.
class BacteriaAttractantSim : public BacteriaReproSim
{
protected:
    // A pointer to our attractant object.
    Distribution *attractant;

public:
    BacteriaAttractantSim()
    {
        // Make and add a new, slimey green distribution for attractant.
        attractant = new Distribution(400, 300, 1);
        attractant->SetColour(0.5f, 1.0f, 0.5f);
        attractant->SetDiffusionSpeed(2);
        attractant->Plot(0.0);
        attractant->SetMaxConc(1.5f);
        attractant->Update();
        grpWorld.push_back(attractant);

        // Tell the bacteria about it.
        grpBacteria.ForEach<Distribution*>(&Bacterium::SetAttractantDist, attractant);

        // Make them respond to attractants.
        grpBacteria.ForEach<double>(&Bacterium::SetAttractantResponse, 0.5);
    }
};

BEGIN_SIMULATION_TABLE
    ADD_SIMULATION("BacteriaGradient", BacteriaGradientSim)
    ADD_SIMULATION("BacteriaSwarm", BacteriaSwarmSim)
    ADD_SIMULATION("BacteriaRepellent", BacteriaRepellentSim)
    ADD_SIMULATION("BacteriaRepro", BacteriaReproSim)
    ADD_SIMULATION("BacteriaAttractant", BacteriaAttractantSim)
```

```
END_SIMULATION_TABLE
```

12.9 A short user's guide

This guide explains briefly how to use the graphical interface to **BEAST**, and the commandline batch mode. These are currently the only interface options, although the library does provide scope for users to implement their own interfaces, e.g. an ActiveX interface allowing **BEAST** demos to run in web pages (I can dream!).

12.9.1 The Graphical User Interface

12.9.1.1 Starting BEAST

To run the program, you need two things: the beast executable (called 'beast') and a plugin. Plugins are usually known as 'shared objects' to Linux users, or 'dynamically linked libraries' in Windows parlance. **BEAST** doesn't do anything at all without having at least one plugin loaded. Plugins have the extension '.so' on Linux or '.dll' on Windows and are produced automatically when you compile some **BEAST** code with the provided makefile. To get **BEAST** running a plugin called plugin.so, you would type:

```
beast ./plugin.so
```

assuming plugin.so is in your current directory.

12.9.1.2 Running simulations in the GUI

To start a simulation, go to the *File* menu and click on the simulation of your choice from a list of up to ten. The simulation should now start in the main window. Any log output (e.g. fitness scores) will appear in the log window below. Different elements may be displayed by toggling the options in the *View* menu, e.g. you can switch off the monitors, trails, collisions or even the animats.

The simulation can be paused and reset in the *Simulation* menu. *High speed* mode can be selected from this menu or by pressing [space] in the main window. When high speed mode is activated, the simulation runs as fast as possible but does not update the display (except during screen redraws).

12.9.1.3 Interacting with the world

You can grab hold of anything which isn't fixed down, this includes all animats and any other objects which have their moveable property set to true. When you click on an object, it will usually display some information in the status bar and can also output information to the log window (programmers note: these are achieved with **ToString** and **OnClick** respectively.)

To change the orientation of the current grabbed object, while holding down with the left button, click and hold the right. You can now spin the object around by moving the mouse around the screen. Animats are not able to move while under the influence of the mouse pointer.

Annoyingly, clicking on objects often leads to them being moved slightly which can adversely affect the simulation (it's the uncertainty principle at work again...). To avoid this, you can cycle through the animats by pressing tab and shift-tab to go backwards. (Programmers: in this case **OnSelect** is called.) This option is also available in the *World* menu.

12.9.1.4 Saving and loading

Depending on the simulation, groups of animats or objects may be saved and loaded through the *File* menu. When a population - that is, a group of animats which are being evolved using a GA - is saved, the parameters and history of the GA are saved with them, and the data stored usually corresponds to the genotypes of the population rather than the configuration of the Animats in the world. On reloading saved populations, the number of generations for that population will not be reflected in the status bar, but will appear in log window output. The reason for this is that if the number of generations registered by the simulation were linked to the number of generations for any given population in the simulation, things would cease to make sense when comparing different generations of two coevolved populations.

12.9.2 Batch mode

For long simulations the GUI is an unnecessary burden. Batch mode allows you to run simulations from the commandline without using the GUI. Batch mode only works for simulations with finite runs, generations, assessments and timesteps (programmers: the default is one run, infinite generations, 1 assessment and 1000 timesteps - change these from the constructors of your Simulation objects.)

The syntax for running in batch mode is:

```
beastbatch [plugin] [simname] [<grpname> <filename> ...]
```

Where `plugin` is the name of your compiled simulation, `simname` is the name of the simulation you wish to run, as normally found in the *File* menu. You may optionally specify groups to be saved at the end of the simulation by including the name of the group (`grpname`) and a `filename` to save to. To save more than one group, include additional pairs at the end of the command.

The only situation in which you won't want to specify group and file names on the commandline is if you have implemented your own code for saving results, or are only interested in the log output which will be redirected to standard out.

An example commandline to run batch mode:

```
./demos -batch Chase Predators predators.pop Prey prey.pop
```

Note that this won't work, since the supplied code has infinite generations, but hopefully you get the picture.

12.10 Tutorial 0: A note on the code

The coding style of the **BEAST** examples may appear unusual to a seasoned C++ programmer, having more in common with object oriented scripting languages and higher-level languages such as C#. The two main departures from usual C++ style are the use of properties, and the presence of a fake `This` reference.

12.10.1 Properties

Properties are used to wrap member variables of classes in the same way as accessors and mutators. Unlike accessors and mutators properties do not appear as function calls, but rather look like the original variables. So, instead of writing:

```
object.SetRadius(3);
```

We can write:

```
object.Radius = 3;
```

What are the benefits of this? Ultimately properties will allow more readable code (although their **BEAST** implementation is only in its infancy), replacing:

```
object.SetRadius(object.GetRadius() + 5);
```

with

```
object.Radius += 5;
```

In the meantime, they mainly aid code readability. One other minor feature which is really useful when defining fitness functions which require typecasting in order to avoid compiler errors is the as template method. Instead of writing:

```
fitness = static_cast<float>(object.GetPowerUsed());
```

we can write

```
fitness = object.PowerUsed.as<float>();
```

which is slightly shorter and a whole lot more readable.

If you're a hardcore C++ programmer and you're concerned about speed and other overheads when using properties then yes, your fears are correct: referencing a property rather than a member function directly does lead to one additional dereference although it is hoped that a good compiler will optimise this away. Internally, properties are not used within the **BEAST** library, they are simply made available for convenience when programming through the API.

In **BEAST**, properties are normally named as the capitalised form of the variable they wrap.

12.10.2 tutorial0_2

The `This` reference is defined exactly like this:

```
#define This (*this)
```

And is used to clarify code by making the scope of member variables explicit in functions which use them. Admittedly we could just as easily write `this->` to achieve the same result, but `This` just looks neater. So there.

12.11 Tutorial 1: Building your first Animat

In this tutorial we are going to:

- Introduce the C++ interface for animats
- Derive a new, moving animat from the Animat base class
- Play with some very simple emergent behaviour

First of all, here's a (very) cut down version of the animati's C++ interface, contained in header file **animat.h**:

```
class Animat : public WorldObject
{
public:
    Animat();
    ~Animat();
    virtual void    Init();
    void            Add(std::string name, Sensor* s);

    virtual void    Control(){}

    virtual void    OnCollision(WorldObject* r){}

protected:
    SensorContainer sensors;
    ToolContainer tools;
    ControlContainer controls;

    Vector2D        velocity;
    int             maximumSpeed;
    double          maxRotateSpeed;
};
```

As you can see, Animat is derived from WorldObject, which itself is derived from Drawable, so to really understand what the Animat class is and isn146t for, we need a quick overview of Drawable and WorldObject.

Drawable is a base class for anything in the World which you see on the screen, so animats, obstacles and objects in the world, and also display features such as collision dots, animat trails and sensor output are all derived from Drawable. The only features of Drawable important here are:

```
    virtual void    Init();

    Vector2D            location;
    double              orientation;
```

The location of objects is stored as a vector from the bottom left corner of the screen, so location is really just x,y coordinates. The orientation, as with all orientations in the program, is in radians going anti-clockwise and counting from east. Init is called at the start of a simulation and gives the object the chance to set itself up.

WorldObject is derived from Drawable and is the base class for everything involved in simulations - animats, obstacles and other objects (but not collision dots, trails or other display features which are purely Drawable). WorldObject provides features such as collision detection, and lifecycle methods such as Update which is called at the start of each frame and is where the animat does

most of its 145thinking146 and moving, usually dependent on sensor outputs, but we'll be looking at how animats are controlled in more detail later.

Now we can look at the contents of the Animat class. Animats are mobile, so they have a vector describing their velocity. They are equipped with sensors, which come in a variety of shapes and sizes and tell them about their environment, and tools, which let them manipulate their environment. The ControlContainer is really just a series of real values, the first two of which control the left and right wheels. This two-wheel approach works a little like tracks on a tank and gives the animat a full range of movement from as few inputs as possible. The sensors, tools and controls attributes are actually maps, a class provided by C++'s Standard Template Library which works something like a vector, but lets us enumerate the contents using any type. In this case, string is used, so to set the speed of each wheel to zero, simply put:

```
Controls["left"] = 0.0;
Controls["right"] = 0.0;
```

As it is, the vanilla Animat class above won146t do anything. You could make some plain animats and put them into a simulation but they146ll just sit there. To make animats which do something, we need to derive a new class from the one provided.

12.11.1 Making Dancing Animats

We're going to make Animats which follow each other around, doing the conga. If you'd find a biological metaphor more convincing, we are going to create Musk Shrews. Juvenile Musk Shrews follow their mother around by forming a 'caravan', which is just a trail of shrews biting the tail of the shrew in front. Since shrews are born blind, this behaviour is important to their survival, and if the mother is lost, a desperate shrew at the front of a caravan will clamp on to whatever is in front of it, even if it's the back of the caravan. Although not as impressive as flocks and shoals, this is still a simple form of emergent behaviour: by following a simple rule, "follow the nearest shrew in front of you", shrews produce the more complex behaviour of forming caravans.

Our shrew will be a simple two-sensor animat, wired up like a Braitenburg vehicle with two sensors each driving the opposite wheel. The file we're about to write will be a .cc file, called something like 'shrew.cc'.

First of all, we need to include the interface header:

```
#include "wxsimenv.h"
```

And also the header for the Animat class, and the header for sensors:

```
#include "animat.h"
#include "sensor.h"
```

Next we specify the namespace, GASimEnv. Everything in the simulation environment resides within this namespace.

```
using namespace GASimEnv;
```

Now to create our new animat, which will be called `Shrew`. The whole definition for the class is below, including all methods.

```
class Shrew : public Animat // Shrew is derived from Animat
{
```

```
public:
    Shrew()
    {
        This.Add("left", ProximitySensor<Shrew>(PI/5, 200.0, -PI/20));
        This.Add("right", ProximitySensor<Shrew>(PI/5, 200.0, PI/20));

        This.InitRandom = true;     // Start in random locations

        This.Radius = 10.0;         // Shrews are a little bigger than usual
    }
    virtual ~Shrew(){}

    virtual void Control()
    {
        This.Controls["left"] = This.Sensors["right"]->GetOutput();
        This.Controls["right"] = This.Sensors["left"]->GetOutput();
    }
};
```

Note that the overridden Control method has the keyword `virtual` this is important because it ensures that even though the simulation environment doesn't know whether it contains plain animats, special derived animats or a mixture, the correct control function will be selected at runtime. Without the virtual keyword, the new Control method might be ignored and the original empty one used instead.

Now that we have defined the shrew, all that's left to do is set up the simulation.

```
class ShrewSimulation : public Simulation
{
    Group<Shrew> grpShrew;

public:
    ShrewSimulation():
    grpShrew(30)
    {
        This.Add("Shrews", grpShrew);
    }
};
```

The class `ShrewSimulation` has simply been derived from `Simulation`. A data member has been added called `grpShrew`, which is a `Group` of `Shrews`. A `Group` is really just a `vector` with some additional methods for adding its contents to the `World`. `ShrewSimulation's` constructor only does two things: `grpShrew` is configured to create 30 shrews, and then the group is added to the `World` and given the name, "Shrews".

Finally, we need to tell the simulation environment about the new simulation class. This is done using macros:

```
BEGIN_SIMULATION_TABLE
    ADD_SIMULATION("Shrews", ShrewSimulation)
END_SIMULATION_TABLE
```

As you can see, simulations go into a table, and it's possible to compile the simulation environment with up to ten simulations which are started from the File menu.

To compile the simulation, just type (assuming the above code is saved in the file shrew.cc):

make shrew

And after a few moments you will be able to see your simulation in action by typing:

./shrew

You should then be looking at a screen full of quite confused, dancing shrew-bots.

12.11.2 More things to try

You can see from the code above how simple it is to set up sensors and to turn sensor outputs into wheel inputs. By adding more sensors you can create more complex behaviours including steering and rudimentary flocking. For inspiration, take a look at this page: `http://www.red3d.com/cwr/boids/`

12.12 Tutorial 2: Adding Objects and Interactive Animats

In this tutorial we are going to:

- Add objects to the world
- Make the animats interact with those objects
- Hand-code a simple neural controller
- Create more specialised objects with inheritance
- Take a look at how sensors work

Some simulations need only animats to be present in the world, but sometimes it will be necessary to introduce new types of object which represent other features such as walls, food or holes in the ground. If your simulation requires the animats to have an interactive relationship with those objects, such as eating food or falling in holes, you will need to add a small amount of code which produces this response.

12.12.1 Making a Cheese Collecting Animat

For this tutorial we're going to make a new type of animat which looks for cheese. 'CheeseCollectingAnimat' is rather long, so we'll call it Mouse. Mouse's world has only two types of thing in: other mice, and cheese. Cheeses will be represented by yellow dots.

We'll give Mouse a simpler sensor than Shrew had, called a NearestAngleSensor. All this tells us is the angle to the nearest object of a specified type: negative angles for objects on the left, positive angles for objects on the right and zero for objects dead ahead. Initially Mouse will have a simple hand-coded controller which orients him towards the nearest Cheese and then moves him forward, but later on we'll replace this controller with an equivalent neural network.

As before we need to #include the files wxsimenv.h, **animat.h** and **sensor.h**.

```
#include "wxsimenv.h"
#include "animat.h"
#include "sensor.h"
```

Now, we'll define the Cheese class. Since cheese is just a stationary object, it can be derived from WorldObject.

```
class Cheese : public WorldObject
{
public:
    Cheese()
    {
        This.Radius = 2.5f;                    // Cheeses are quite small
        This.SetColour(ColourPalette[COLOUR_YELLOW]);// Cheeses are yellow
        This.InitRandom = true;                // Cheases are scattered
    }
    virtual ~Cheese(){}

    // When a Cheese is Eaten, it reappears in a random location.
    void Eaten()
    {
        This.Location = myWorld->RandomLocation();
    }
};
```

The constructor for Cheese sets the radius to 2.5, the colour to yellow, and the initRandom flag to true so that cheeses will be randomly scattered throughout the world. As before, Cheese bears the mandatory virtual destructor.

One new method has been added, called Eaten. When a Mouse bumps into a Cheese, he will call Eaten on the cheese, which then causes the cheese to be relocated elsewhere in the world. myWorld is a pointer to whichever world the cheese is in. RandomLocation is a method of World which returns a pair of random coordinates somewhere in the boundary of the world.

Now to define Mouse:

```
class Mouse : public Animat
{
public:
    Mouse()
    {
        This.Add("angle", NearestAngleSensor<Cheese>());
        This.InitRandom = true;
    }

    virtual void Control()
    {
        double o = This.Sensors["angle"]->GetOutput();
        This.Controls["right"] = 0.5 - (o > 0.0 ? o : 0.0);
        This.Controls["left"] = 0.5 + (o < 0.0 ? o : 0.0);
    }

    virtual void OnCollision(WorldObject* obj)
    {
        Cheese* cheese;

        if (IsKindOf(obj,cheese)) {
            cheese->Eaten();
        }

        Animat::OnCollision(obj);
    }
};
```

As before, Mouse is derived from Animat. The constructor adds just one sensor this time, the very simple NearestAngleSensor which has a range beyond the limit of the World (i.e. unlimited for our purposes), and as usual tells Mouse to start in a random place in the world.

The Control method does a couple of simple calculations which give Mouse the behaviour we want: the right wheel goes at half speed plus the leftwards magnitude of the angle to the nearest cheese, the left wheel goes at half speed plus the rightwards magnitude of the angle, again something like a Braitenburg vehicle.

This time the OnCollision method is also overridden. OnCollision is called on an object whenever it is involved in a collision. The argument is a pointer to the object Mouse is colliding with. Because every object in the world is derived from WorldObject, a handy feature of C++ known as polymorphism allows us to use a pointer to WorldObject to point to any object of a class derived from WorldObject. The object being passed into OnCollision might be a Cheese, or it might be another Mouse - both types of objects can be passed in, and then a test can be done to determine how Mouse acts.

We don't want our mice to be cannibals so we need to test the object for cheesiness. This is done using a system known as Runtime Type Identification (RTTI), but in the simulation environment, RTTI is encapsulated into two functions: IsA and IsKindOf. Both functions take two pointers as arguments: a pointer to the object being tested and an empty pointer of the type we are testing for. Both functions return a bool - true if a match is made, false if not. IsA will only return true if the type of both pointers is exactly the same, whereas IsKindOf will return true if the object

being tested is of exactly the same type, or if it's of a type derived from the type being tested. We'll look at this in more detail later.

If IsA or IsKindOf return true, the test pointer which was passed in (in this case a pointer to a Cheese) will be made to point to the object which has successfully passed the test. Now that we have a Cheese pointer to the object, we can call Cheese methods on it, so Eaten is called.

Finally, once the overridden OnCollision method is over, Animat's native OnCollision method still has work to do, such as stopping the Animat going through solid obstacles (other mice in this case) so Animat::OnCollision is called, passing the obj pointer through.

The Simulation is set up in much the same way as before, but with two Group objects:

```
class MouseSimulation : public Simulation
{
    Group<Mouse> theMice;
    Group<Cheese> theCheeses;

public:
    MouseSimulation():
    theMice(30)
    theCheeses(50)
    {
        This.Add("Mice", theMice);
        This.Add("Cheeses", theCheeses);
    }
};
```

And the usual simulation table:

```
BEGIN_SIMULATION_TABLE
    ADD_SIMULATION("Mice", MouseSimulation)
END_SIMULATION_TABLE
```

Compile and run as before and you should be looking at a bunch of greedy mice.

12.12.2 Introducing Neural Controllers

Two types of Artificial Neural Network (ANN) are provided with the simulation environment: FeedForwardNet, which is a feed-forward network with one hidden layer; and DynamicalNet which is a fully-recurrant dynamical neural network. For more details on these two classes and the networks they implement, see their documentation.

As a test of the FeedForwardNet class and a demonstration of one way of using it, we will code the simple controller used by the mouse into an equivalent neural network.

This is what the configuration of the neural network will look like: Note that each node's output will be put through a simple threshold function, rather than the sigmoid function normally used in this type of neural network. The two weights at the input nodes ensure that if the angle is positive, one side of the network is activated, if the angle is negative the other side is activated. The bias values of -0.5 at the two output nodes ensure that each output is at least 0.5, as in the original. To adjust the mouse to use this neural network, we need to make the following code changes:

The header file for FeedForwardNet needs to be included:

```
#include "feedforwardnet.h"
```

A private member must be added for the FeedForwardNet object:

```
private:
    FeedForwardNet myBrain;
```

The above configuration must be set up in myBrain when the Mouse is constructed. To do this, change the Mouse constructor to:

```
Mouse():
myBrain(1, 2, 2, false)
{
    vector<float> ffnConfig;
    ffnConfig.push_back( 1.0f); // 1st input 1st weight
    ffnConfig.push_back( 0.0f); // 1st input bias
    ffnConfig.push_back(-1.0f); // 2nd input 1st weight
    ffnConfig.push_back( 0.0f); // 2nd input bias
    ffnConfig.push_back( 1.0f); // 1st hidden 1st weight (for input 1)
    ffnConfig.push_back( 0.0f); // 1st hidden 2nd weight (for input 2)
    ffnConfig.push_back(-0.5f); // 1st hidden bias
    ffnConfig.push_back( 0.0f); // 2nd hidden 1st weight (for input 1)
    ffnConfig.push_back( 1.0f); // 2nd hidden 2nd weight (for input 2)
    ffnConfig.push_back(-0.5f); // 2nd hidden bias

    This.myBrain.SetConfiguration(ffnConfig);

    This.Add("angle", NearestAngleSensor<Cheese>());
    This.InitRandom = true;
}
```

The line myBrain(1, 2, 2, true) configures the built-in FeedForwardNet with one input, two hidden layers, two outputs and the false specifies that a simple threshold function should be used rather than the sigmoid function.

The vector created in the constructor contains the values from the diagram. The FeedForwardNet146s SetConfiguration method inserts those values into the correct places in the network. This method of configuring networks is not very human-friendly, but this network wasn146t really designed to be hand-configured. However the idea of configuring the network using a fixed-length list of values is ideal for a Genetic Algorithm (surprise!) which we146ll be looking at in the next tutorial.

Finally, the Mouse needs a new Control method which feeds the input from the sensor into myBrain and then sets the motors using the network146s output.

```
virtual void Control()
{
    This.myBrain.SetInput(0, sensors["angle"]->GetOutput());

    This.myBrain.Fire();

    This.Controls["left"] = This.myBrain.GetOutput(0);
    This.Controls["right"] = This.myBrain.GetOutput(1);
}
```

Compile and run the now-brainy Mouse class and the mice should exhibit exactly the same behaviour as their non-neural predecessors. So much for neural nets, but at least we know it works!

12.12.3 Making Mice Picky

Some mice will eat anything, as long as it's cheese. Some mice prefer gruyere and won't touch anything else and some other mice won't touch the fancy stuff and stick to plain old cheese. We're going to set up this simulation now, to demonstrate the use of inheritance in constructing simulations and also the difference between IsA and IsKindOf.

First, we'll create a couple of new cheeses by inheriting from the cheese base class.

```
class Stilton : public Cheese
{
    Stilton()
    {
        This.SetColour(0.2f, 1.0f, 0.2f);
    }
    virtual ~Stilton(){}
};

class Gruyere : public Cheese
{
    Gruyere()
    {
        This.SetColour(1.0f, 0.5f, 0.0f);
    }
    virtual ~Gruyere(){}
};
```

Both of these new classes are identical to cheese except for their colour and their type. The diameter and initRandom flag are set automatically because when a class is constructed, all of its base class constructors are called automatically.

Now we'll create two new kinds of mouse: `PickyMouse` and `OldFashionedMouse`.

```
class PickyMouse : public Mouse
{
public:
    virtual ~PickyMouse(){}

    virtual void OnCollision(WorldObject* obj)
    {
        Gruyere* cheese;

        if (IsA(obj, cheese)) {
            cheese->Eaten();
        }

        This.Animat::OnCollision(obj);
    }
};

class OldFashionedMouse : public Mouse
{
public:
    virtual ~OldFashionedMouse(){}

    virtual void OnCollision(WorldObject* obj)
    {
        Cheese* cheese;

        if (IsA(obj, cheese)) {
            cheese->Eaten();
        }

        This.Animat::OnCollision(obj);
    }
};
```

This time, only OnCollision needs to be overridden. In both cases IsA is used, rather than IsKindOf. Thus, even though Gruyere is a kind of cheese, OldFashionedMouse won't have anything to do with it. Either IsA or IsKindOf could have been used in the code for PickyMouse because Gruyere is the only class in the world which is a kind of Gruyere.

Add four more groups to your simulation - one for each of the new classes - and adjust the numbers accordingly, then see what sort of chaos ensues! It might also help to add constructors to the different mice to set their colours so that it's clear which type is which.

12.12.4 Modifying Sensors

But... there is a problem. Your mice might know what kind of cheese they like but they have no way of recognising which is which from a distance. They all still have the same sensor which simply senses any cheese in the hierarchy, so they charge towards the nearest piece of cheese and if it's not the right kind, they just carry on charging towards it because it's still the nearest cheese.

What we need to do is modify the sensor so that it detects particular types of cheese.

Sensors in the simulation environment are very flexible and can be combined and mixed up in various ways. There are four elements to every sensor:

- A sensor class which decides which individuals in the world to look at:
  - The basic Sensor class will consider individuals in the whole world.
  - AreaSensor can take on a specified shape and only looks at individuals in that area.
  - TouchSensor will only consider individuals touching the sensor's owner.
  - BeamSensor casts a ray of a certain angle scope and radius out from a particular point. A scope of 0 makes a laser which only sees dead ahead, a scope of 2 x pi (360 degrees) will detect an object within range in any direction.
- A matching function which decides whether or not an object is of interest to the sensor.
- An evaluation function which obtains certain information about the object, e.g. how far away it is, what the angle to it is, how big it is or perhaps just keeps a count of the number of objects sensed.
- A scaling function which takes the output of the evaluation function and modifies it to represent suitable output. For example, the NearestAngleSensor's scaling function takes the angle evaluator's output and scales it from the range +/- pi to +/- 1, which is easier to work with. A distance sensor usually has to scale depending on the range of the sensor and in the case of the ProximitySensor used in tutorial one, the sensor output is inverted so that smaller distances result in greater output. The scaling function might also incorporate features such as random noise to simulate real sensors.

Sensor functions aren't really functions at all, but instead are function objects (or functors) - classes with the parenthesis operator (or operator()) overloaded so that an object can appear to be used as a function. For more details on how sensors work, look in the Sensors section of the documentation.

So there is really no such thing as a NearestAngleSensor, it's just a bunch of function objects attached to a class. NearestAngleSensor is really a function that sets up a Sensor with the correct objects and returns a pointer to that sensor:

```
template <class T>
Sensor* NearestAngleSensor()
{
    Sensor* s = new Sensor(Vector2D(0.0, 0.0), 0.0);
    s->SetMatchingFunction(new MatchKindOf<T>);
    s->SetEvaluationFunction(new EvalNearestAngle(s, 1000.0));
    s->SetScalingFunction(new ScaleLinear(-PI, PI, -1.0, 1.0));

    return s;
}
```

So to change the matching function of `PickyMouse`, simply add this line to the `PickyMouse` constructor:

```
sensors["cheese sensor"]->SetMatchingFunction(new MatchExact<Gruyere>);
```

The old matching function will be deleted and a new Gruyere matcher will replace it. A similar thing can be done for the `OldFashionedMouse`

12.12.5 Things to try

- Set the fourth parameter of the constructor for myBrain to true so that a sigmoid activation function is used rather than a threshold. See if you can adjust the neural net weights to compensate.
- Duplicate the Eaten method of Cheese inside PickyMouse and change OldFashionedMouse so that he eats PickyMouse instead of Cheese.

12.13 Tutorial 3: Introducing the Genetic Algorithm

In this tutorial we will:

- Replace the hand-configured neural net from tutorial 2 with an equivalent net, developed using a GA.
- Set up a predator-prey simulation and co-evolve more complex controllers.

In tutorial two, we replaced a hand-coded controller with a neural equivalent. It didn146t allow us to do anything new, but did prove the neural net works. In the first part of this tutorial, we146ll get rid of the hand-configured neural network and evolve a new one using a Genetic Algorithm. Again we won146t be doing anything new, but it should be a reasonable test for the GA.

12.13.1 Using the GA

The Genetic Algorithm class provided with the simulation environment has numerous features, only a few of which will be useful in any given simulation. The default configuration corresponds as closely as possible to the original GA developed by John Holland. It uses roulette-wheel selection and single-point crossover. The mutation operator has different defaults depending on whether your genes are binary, integer or real valued. If you create a GA which works on the bool type, a simple NOT function is used. To evolve neural networks however, we need to work with real values, in which case the default mutation operator is a normally distributed random function, so genes may be mutated by very high values occasionally, but usually smaller changes will occur.

GeneticAlgorithm is actually a class template which takes for its template parameters the type of the individuals to be evolved (and also the type of the mutation function, although we can ignore that parameter as we will be using its default). The class specified for the first template parameter needs to contain certain methods before the GA can work on it:

- It must have a GetGenotype method, which returns a vector of genes (genotype) corresponding to the object146s configuration (phenotype).
- It must have a SetGenotype method, which takes a vector of genes for its argument and configures an individual accordingly.
- It must have a GetFitness method, which returns a float corresponding to the individual146s fitness, the higher the better. For roulette wheel selection, fitness scores must be positive. (If your individuals don146t always produce positive fitness scores, the GA can clamp or scale them, see GeneticAlgorithm::SetFitnessFix for details).

There are a number of other features required by the GA for evolvable individuals, so to make things as easy as possible, an abstract base class called Evolver is supplied. If you inherit from this class and provide suitable GetGenotype, SetGenotype and GetFitness methods, the other details will be filled in automatically.

But wait! You don146t even need to inherit your own Evolver! An Animat class with a built in feed-forward network and GA compatibility is already provided, called EvoFFNAnimat. EvoFFNAnimat returns the neural net146s configuration vector as its genotype, it initialises the net with one input per sensor and one output per control (i.e. the right and left wheels in the basic Animat) and there is also an automated Control method.

The code for the evolving feed-forward net mouse looks like this:

```
class EvoMouse : public EvoFFNAnimat
```

```
{
public:
    EvoMouse(): cheesesFound(0)
    {
        This.Add("angle", NearestAngleSensor<Cheese>());
        This.InitRandom = true;
        This.InitFFN(4);
    }

    virtual void OnCollision(WorldObject* obj)
    {
        Cheese* cheese;

        if (IsKindOf(obj,cheese)) {
            cheesesFound++;
            cheese->Eaten();
        }

        This.EvoFFNAnimat::OnCollision(obj);
    }

    virtual float GetFitness()const
    {
        return cheesesFound > 0 ? static_cast<float>(cheesesFound) / static_cast<float>(powerUsed) : 0;
    }

    virtual string ToString()const
    {
        ostringstream out;
        out << " Power used: " << powerUsed;
        return out.str();
    }

private:
    int cheesesFound;   // The number of cheeses collected for this run.
};
```

The constructor sets up the sensor and initRandom as before, but now also initialises cheesesFound to 0 and calls InitFFN to set up a feed-forward network with two hidden nodes. The number of inputs is one, since there is only one sensor and the number of outputs is two, one for each wheel. This 1-2-2 setup matches the hand-configured network so there is good reason to believe that the GA will be able to find a suitable network.

OnCollision is exactly as before except that each cheese eaten increments cheesesFound by one, and EvoFFNAnimat::OnCollision is now called at the end.

Finally, GetFitness returns the number of cheeses found, divided by powerUsed which is the total activation of all the controls for the duration of the assessment. This penalises mice that latch onto the unrealistic tactic of going as fast as possible in one direction, picking up more cheese simply due to the greater area covered.

We also need to set up the simulation. This time, instead of a Group, the mice will go into a Population. This is a similar container to Group but also has an associated GA which is used on the population at the end of each assessment.

```
class MouseSimulation : public Simulation
{
    Population<EvoMouse>        theMice;
    GeneticAlgorithm<EvoMouse>  theGA;
    Group<Cheese>               theCheeses;

public:
    MouseSimulation():
    theGA(0.7f, 0.05f),
```

```
    theMice(30, theGA),
    theCheeses(50)
    {
        This.theGA.SetSelection(GA_RANK);
        This.theGA.SetParameter(GA_RANK_SPRESSURE, 2.0);

        This.Add("Mice",        theMice);
        This.Add("Cheeses",     theCheeses);
    }
};
```

The Population of 'evomice' is configured with a reference to a GA which works on EvoMouse objects, this GA will therefore be used at the end of each generation.

Rather than using roulette wheel selection, rank selection has been specified with a selection pressure of two. This means that at the end of the generations the individuals' chances of reproducing depend on how they rank in the overall population. Selection pressure ranges between one and two, one means that all individuals have an equal chance of going on to the next generation (not very useful!) whereas two means that in a population of twelve, the sixth individual has half as much chance of reproducing as the best, the third has three quarters the chance of the best, the ninth has one quarter the chance of the best and so on.

For information on the other options and selection methods available, see the GeneticAlgorithm documentation.

Compile and run the simulation same as usual. This time the animats start off wandering about with no idea about cheese or its uses. At the end of a certain period (the time limit defaults to 1000 time steps) the animats will be taken out of the world and put through the GA. Their offspring then appear as the next generation. To make this process go faster, go to the Simulation menu and click on High Speed. The display will stop updating and the simulation environment will go at the highest speed your machine is capable of. After a few more generations, click cancel and see if your mice are doing anything useful. You should expect to see good results for this particular simulation after about 200 generations.

12.13.2 Coevolutionary Simulations

Now we'll try creating something more complex, a coevolutionary simulation where two populations are evolving and the fitness of each is dependent in some way on the fitness of the other. In this case, we'll evolve predators and prey. The `Predator` class will be similar to `Mouse`, in that it has to seek out prey and receives a point for every `Prey` object it munches on. `Prey` will be penalised each time they get munched. To make things interesting, we'll give predators two long-range, forward facing proximity sensors, a little like the eyes of a hawk or cat, and the prey will have short-range, sideways-facing proximity sensors like fish and rodents - if it makes sense in nature, it should make sense here.

Here is the code for both the `Predator` and `Prey` classes.

```
// Forward declaration for Prey
class Predator;

class Prey : public EvoFFNAnimat
{
public:
    Prey():timesEaten(1)
    {
        This.Add("right", ProximitySensor<Predator>(PI/1.05, 100.0, -PI/2));
        This.Add("left", ProximitySensor<Predator>(PI/1.05, 100.0, PI/2));

        This.InitFFN(4);
```

```
        This.InitRandom = true;
        This.MinSpeed = 0;
        This.MaxSpeed = 100;
    }

    void Eaten()
    {
        This.timesEaten++;
        This.Location = myWorld->RandomLocation();
    }

    float GetFitness()const
    {
        return 1.0f / static_cast<float>(This.timesEaten);
    }

private:
    int timesEaten;
};

class Predator : public EvoFFNAnimat
{
public:
    Predator():preyEaten(0)
    {
        This.Add("left", ProximitySensor<Prey>(PI/5, 200.0, -PI/20));
        This.Add("right", ProximitySensor<Prey>(PI/5, 200.0, PI/20));

        This.InitFFN(4);
        This.InitRandom = true;

        This.MinSpeed = 0;
        This.MaxSpeed = 100;
        This.Radius = 10.0;
    }

    void OnCollision(WorldObject* obj)
    {
        Prey* ptr;

        if (IsKindOf(obj,ptr)) {
            This.preyEaten++;
            ptr->Eaten();
        }

        This.FFNAnimat::OnCollision(obj);
    }

    float GetFitness()const { return preyEaten; }

private:
    int preyEaten;
};
```

Note that there is a forward declaration of `Predator` above the `Prey` class, which simply states that there is a class called 'Predator'. This is to get round the problem of classes which refer to each other - one must be defined after the other, and both contain methods which use pointers to the other, so one is bound to find itself referring to a type which has not yet been defined. The forward declaration simply lets the compiler know that the type `Predator` does exist and so there's no need to get upset when `Prey` starts talking about `Predator` objects which aren't yet defined. By the time `Predator` is defined, the compiler knows all about `Prey`, so it's safe for `Predator` to manipulate `Prey` objects and call member functions.

The only other unusual feature worth mentioning in the code above is the fitness function for `Prey`: are more fit the fewer times they are eaten, so it would make sense to subtract one from

the `Prey's` fitness each time it is eaten, however the standard GA using roulette wheel selection is unable to work with negative scores. Two solutions are available:

- Set up the GA to adjust the scores using `SetFitnessFix`, which can be set to one of three options:
  - `GA_IGNORE` which is the default setting, no change is made to fitness scores.
  - `GA_CLAMP` which sets any scores below zero to zero.
  - `GA_FIX` which linearly scales the scores up so that the lowest score becomes zero.
- Ensure that GetFitness returns a positive value, in this case by returning the inverse of the number of times the individual has been eaten, so higher numbers become lower but remain positive.

Setting up a GA to do coevolution is very easy, all you need is two GAs. They don't need to know anything about each other, and the fitness functions of the two types of individuals are entirely unrelated. `ChaseSimulation` is therefore set up just like previous simulations, but with two populations (and no groups).

I've chosen population sizes of 30 for speed but feel free to experiment with larger numbers depending on the speed of your computer and how much time you have. One thing we don't want is a simulation with 60 individuals vying for space in our environment - this would reduce the effectiveness of the simulation because the chances of randomly bumping into another individual, regardless of whether that individual is effective at chasing or evading (depending on whether it is predator or prey), are increased the more crowded the simulation becomes. One answer is to simply increase the arena size using World::SetWidth and World::SetHeight, but then they'd be too small to see.

Instead, I've opted for a more complex kind of simulation where rather than simulating every individual at once, a selection of each Population go into the arena and are assessed, their scores are then stored and a new selection go in. The selections are made depending on the `Population` object's team size, which is set using SetTeamSize. The default is for the team to be the whole population, but team sizes as low as one can be used, making individual assessments possible.

```
class ChaseSimulation : public Simulation
{
    GeneticAlgorithm<Predator> gaPred;
    GeneticAlgorithm<Prey> gaPrey;
    Population<Predator> popPred;
    Population<Prey> popPrey;

public:
    ChaseSimulation():
    gaPred(0.7f, 0.1f), gaPrey(0.7f, 0.1f),
    popPred(30,gaPred), popPrey(30,gaPrey)
    {
        This.gaPred.SetSelection(GA_RANK);
        This.gaPred.SetParameter(GA_RANK_SPRESSURE, 2.0);

        This.gaPrey.SetSelection(GA_RANK);
        This.gaPrey.SetParameter(GA_RANK_SPRESSURE, 2.0);

        This.popPred.SetTeamSize(5);
        This.popPrey.SetTeamSize(10);
        This.SetAssessments(30);

        This.Add("Predators", popPred);
        This.Add("Prey", popPrey);
    }
};
```

In this simulation, 30 assessments are being made for each generation, so each `Predator` will have had 5 assessments and each `Prey` will have had 10 assessments, and also every `Prey` will have been preyed on by every `Predator`.

If you set this simulation going in the usual way and leave it for a few hundred generations you will start to see some quite convincing predator/prey behaviour.

12.13.3 More things to try...

- Replace the EvoFFNAnimat used in the predator/prey simulation with EvoDNNAnimat and see if the dynamical network-based solution is any 'cleverer'.
- Give the predators an additional DensitySensor, a beam sensor which counts the number of individuals in its range. More successful predators should be able to use this sensor to concentrate on areas where there are more than one prey nearby. Giving the same kind of sensor to prey will give the prey chance to avoid running into large groups of predators.
- Add sensors allowing predators to sense other predators and prey to sense other prey. Do the prey consequently avoid each other, or huddle together? Do the predators work together or look for their own territory? (Hint: if you find that adding all these sensors slows things down too much, change some of them into nearest angle sensors which are just as effective and much faster for the simulation environment to process.)
- Introduce a third coevolving class which can eat predators but is eaten by prey (yes, I know it's silly but it might be funny to watch).

12.14 Tutorial 4: More Advanced Simulations

In this tutorial we will look at some of the more advanced options available in creating new simulations. In particular:

- The GAVariant gene type and custom genes.
- Creating more complex evolving animats.
- Creating more complex worlds.
- Creating more complex sensors.

12.14.1 The GAVariant Type

Supposing you want to run an evolutionary simulation using some genotype other than a simple list of values - perhaps your genes are a mixture of real values, integers and booleans, or perhaps they are some completely different type of your own devising. There are two options available to you in this situation:

- Create a mutation operator for your custom gene type and configure your Evolvers, Population and GeneticAlgorithm to use custom genes (specified in the Evolver template declaration) and your custom mutation operator. More information on this is available in the GeneticAlgorithm documentation.
- For many simulations you will just need to be able to work with more than one of the basic types available in C++. This is where the GAVariant type comes in.

If you have programmed in BASIC, LISP or other high-level languages, you will already be familiar with the concept of a variant type. A variant is a data type which is able to act as a range of different types, performing conversions between them as required. A general purpose variant might allow you to add two strings containing numbers together and output the result as a float. GAVariant is simpler than this since it only deals with five basic types: int, double, float, char and bool.

12.14.1.1

12.15 Todo List

Class BEAST::DNNAnimat Review brain ownership/destructor

Member BEAST::DynamicalNet::Serialise(std::ostream &) const Serialise input/output channel/node config

Member BEAST::DynamicalNet::Unserialise(std::istream &) Unserialise input/output channel/node config

Class BEAST::MutationOperator< GAVariant > serialise/unserialise

Member BEAST::SelfSensor::GetOutput() const Sensing of fitness, perhaps adjusting SelfSensor into a template which can be made to sense any detail about the owner.

Member BEAST::Sensor::~Sensor() Shared functors?

Member BEAST::Group::Serialise(std::ostream &) const Finish!

Member BEAST::Group::Unserialise(std::istream &) Add general object unserialiser

Member BEAST::Group::Unserialise(std::istream &) Rather than removing only objects from this group, all objects with the same type as this group are removed.

Member BEAST::GeneticAlgorithm::Generate() Account for odd numbered output population sizes

Member BEAST::GeneticAlgorithm::ToString() const store time data

12.16 Deprecated List

Member BEAST::PolarVector(double l, double a) Probably of no use to anyone unless they have a strange fascination with polar coordinates.

Index