# The ANNABELL system

Version 2.0.9

MANUAL

Bruno Golosio[a], Angelo Cangelosi[b], Olesya Gamotina[a], Giovanni Luca Masala[a]

[a]POLCOMING Department, Section of Engineering and Information Technologies,
University of Sassari, Italy

[b]Centre for Robotics and Neural Systems, School of Computing and Mathematics,
University of Plymouth, United Kingdom

June 6, 2015

## Contents

## 1 Introduction

**ANNABELL** is a cognitive architecture, entirely based on a large-scale neural network, which models the neural processes that underpin early language development. The system is able of learning natural language starting from tabula rasa, only by communicating with a human through a text-based interface. The global organization of the system os based on Baddeley's working memory model, combined with some ideas from Cowan and Oberauer models. At the lower level, the system is entirely composed by interconnected artificial neurons. The learnable connections among the neurons are updated by a discrete version of the Hebbian learning rule. The system was tested on a database of 1587 input sentences, and produced 521 output sentences.
A description of the model and of the database used for its validation is provided in our article "A cognitive neural architecture able to learn and communicate through natural language". A detailed description of the neural architecture is provided in the technical document "The ANNABELL system architecture".
This document provides instruction for downloading, installing and using the software package.
As is true for all non-trivial software, ANNABELL is not free of bugs. We try and fix bugs in every released version. In order to do this, we would like to ask all users to send us their bugreports, which should consist of the input-files that provoke the bug. Send reports to golosio@uniss.it.
ANNABELL is released under the terms of the GPLv3.

## 2 Installation instructions

### 2.1 Requirements

In order to compile the software from source, you need a C++ compiler with OpenMP support (v3.0 or higher), which is freely available for most common platforms.In addition, your PATH must contain basic build tools such as make.
NOTE ABOUT GPU (CUDA) VERSION:
This distribution includes a mixed C++/CUDA version of the software, which is partially implemented for GPU systems. It requires CUDA libraries version >=5.5 and CUDA Toolkit version >=5.5. The CUDA installation can be done through the bash script make_cuda.sh, which uses the nvcc compiler. However, be aware that currently the CUDA version installation and usage is undocumented.

### 2.2 Downloading the source code

The source code can be downloaded from the **ANNABELL downloads** repository:
http://www.neuralsystems.net/annabell/files

### 2.3 Extracting the source

Extract the source using any program able to extract/uncompress tar-gz archives.
For instance, in Linux, Unix and Unix-like systems you can extract the source using the tar utility, by typing the command:

```
tar -xvzf annabell-x.x.x.tgz
```

where x.x.x refers to the version of the program that you are installing. This will create a new directory annabell-x.x.x under the current directory containing the source code for the distribution.

### 2.4 Building and installing the program

There are two main methods of installing ANNABELL from source: system wide installation and personal installation. The former requires root privileges on most systems. Both ways are described below.
System wide installation

After unpacking the distribution, go to the directory "annabell-x.x.x" (where x.x.x refers to the version of the program that you are installing) and type the command:

```
./configure
```

You can then build the program by typing,

```
make
```

The installation can be performed with the command

```
make install
```

The default installation directory prefix is /usr/local.
The installation directory can be changed with the --prefix option to configure. Consult the "Further information" section below for instructions on installing the library in another location or changing other default compilation options.
Try running ANNABELL:

```
annabell
```

## 2.5 Personal installation

After unpacking the distribution, go to the directory "annabell-x.x.x" (where x.x.x refers to the version of the program that you are installing) and type the command:

```
./configure --prefix=DIRNAME
```

where DIRNAME is the name of the directory where you want to install the binary executable. You can then build the program by typing,

```
make
```

The installation can be performed with the command

```
make install
```

This command will install the executable in the installation directory.
You should add the installation directory to the PATH environmental variable.
For instance, if you use the bash shell you can add the following lines to the file .bashrc (or bashrc_private if you use it) in your home directory:

```
export PATH=${PATH}:DIRNAME:
```

If you use the tcsh shell you can add the following lines to the file .cshrc
(or cshrc_private if you use it) in your home directory:

```
setenv PATH ${PATH}:DIRNAME:
```

Try running ANNABELL:

```
annabell
```

## 2.6 Platforms known to compile and run ANNABELL

Linux (32-bit/64-bit)
In principle, it should be possible to compile and run ANNABELL in any platform that supports a C++ compiler and OpenMP version >=3.0.

# 3 User guide

## 3.1 Getting started

The present tutorial is complementary to the general description of the system and to the description of the database, which are provided in the Supplemental Material document. You should read that document before reading this tutorial.
It can be useful to start working with simple examples, as the ones described in Sect. 6. First, run the example as it is and display the results of the simulation. Then you can try to modify the example.

## 3.2 The interface

The human-agent communication is achieved through an user interface between the human interlocutor and the system. The interface converts words into input patterns and submits them one by one to the system, extracts output patterns and convert them to words. It also sends reward signals to the system when prompted by the human. The interface includes a monitor tool that can be used to display the content of the SSMs that compose the system.
The following conventions should be used when typing phrases and/or commands through the interface:
use lowercase letters, with no punctuations, no special characters; uppercase should be used only for the first character of a proper noun; a sentence should NOT start with a capital letter (unless it starts with a proper noun);
by convention, questions should START with a question mark, as: "? how old are you";
words with suffix must be splitted in the form: base -suffix; e.g. animals → animal -s, writing → write -ing, etc.
note that the article "an" is automatically translated to "a" by the interface.
a phrase is defined as a set of tokens (words) that ends with a carriage return (newline) excluding commands;
generally a phrase is a short sentence or a clause; the limit is 10 tokens;
a long sentence should be split in more phrases; the division is arbitrary, however keep in mind that the system learns from the way of splitting that you use;
The following is a list of the main commands that can be executed through the interface. The complete list is reported in Sect. 7. All commands start with a dot. Many commands have an abbreviated form, which is indicated besides the extended form between parenthesis.

`.phrase(.ph) phrase`
Suggests a phrase to be retrieved by the association process.

`.word_group(.wg) word group`
Suggests a word group to be extracted from the working phrase.

`.partial_reward(.prw)`
Sends the word group to output and produces a partial reward for the past state-action sequence.

`.reward(.rw)`
Sends the word group to output and produces a (conclusive) reward for the past state-action sequence.

`.exploit(.x) phrase`
Starts an exploitation phase. At the end of this phase, the system is expected to respond to the input phrase with an appropriate output phrase. Without argument, the exploitation is related to the previous input phrase.

`.file(.f) file_name`
Loads phrases and/or commands from the file file_name.

`.monitor(.m) command`
Sends a command to the monitor

`.logfile file_name`
Writes the output to a log file.

`.logfile off`
Closes the log file.

`.quit(.q)`
exits the program or the current input file

## 3.3 The monitor

The monitor is a tool that can be used to display the state of selected SSMs, in every operating mode or in selected operating modes. For SSMs that represent phrases, the monitor maps the correspondence between the neuron indexes and the corresponding words, so that it can display the SSM content in a easily interpretable form, i.e. readable words rather than neuron indexes and states. Similarly, the state of action neurons is displayed using the names that by convention are given to the corresponding actions. The monitor commands are the following:

`.monitor console on`
Switches on the monitor output to the console.

`.monitor console off`
Switches off the monitor output to the console.

`.monitor display mode_change on`
Displays a message when the operating mode changes.

`.monitor display mode_change off`
Stops displaying a message when the operating mode changes.

`.monitor display SSM_name always on`
Displays the content of SSM_name in all operating modes.

`.monitor display SSM_name always off`
Never display the content of SSM_name.

`.monitor display SSM_name operating_mode on`
Displays the content of SSM_name in the specified operating mode.
Operating mode can be acquire, associate, explore, exploit or reward.

`.monitor display SSM_name operating_mode off`
Do not display the content of SSM_name in the specified operating mode.
Operating mode can be acquire, associate, explore, exploit or reward.

## 3.4 Simple examples

The following two examples can be found in the directory simple_examples .
After launching the main program:

```
.annabell
```

they can be executed by issuing the commands

```
.f example1.txt
```

for the first example, and

```
.f example2.txt
```

for the second example. Note that the training procedure for both examples can be simplified using macro commands. The use of macro commands will be described in Sect. 8.

### 3.4.1 First example: categorization.

This example uses the animal classification to show the categorization ability of the system. A first very simple test can be made by launching the program and typing phrases as:

```
the turtle is a reptile
the eagle is a bird
the dog is a mammal
...
```

all lowercase, without punctuations, mixed to other phrases, as for instance

```
fish -es live in the water
reptile -s have cold blood
the turtle is slow
...
```

The order in which the phrases are submitted is not important. It is possible to display what happens inside the system during and after the acquisition through the monitor. For instance, by typing.

```
.logfile log.txt
.monitor display mode_change on
.monitor display Header acquire on
.monitor display IW acquire on
.monitor display InPhB acquire on
.monitor display PhI acquire on
.monitor display AcqAct acquire on
.monitor display ElAct acquire on
```

it is possible to monitor the content of the input-word buffer, the input-phrase buffer, the phrase index, the acquisition actions and the elaboration actions during the acquisition operating mode. Note that the first two lines are used to to keep track of the output also in a logfile and to display changes of the operating mode.
Similarly, by typing the commands

```
.monitor display Header associate on
.monitor display InPhB associate on
.monitor display WkPhB associate on
.monitor display WGB associate on
.monitor display PhI associate on
.monitor display WGI associate on
.monitor display AcqAct associate on
.monitor display ElAct associate on
```

it is possible to monitor the content of the input-phrase buffer, the working phrase buffer, the word-group buffer, the phrase index, the word-group index, the acquisition actions and the elaboration actions during the association operating mode.
Therefore, by typing a phrase, as for instance

```
the dog is a mammal
```

it is possible to see that during the acquisition phase the phrase index grows from zero to four, while the input words are copied from IW to the corresponding slots of the input phrase buffer. During the association operating mode the system extracts all possible groups of consecutive words (with maximum four words in the group) from the working phrase, and builds the associations between the word groups and the working phrase.
The teacher could now ask the system an animal belonging to one of the categories that he used before, e.g.

```
tell me a mammal
```

Clearly at this point the system has no idea of the meaning of this phrase, because it started from a blank condition (tabula rasa). However, it can use this phrase to start an exploration phase, during which the system can retrieve phrases memorized by the association mechanism and build new phrases through partially-random action sequences. The basic action sequence used during the exploration operating mode is the following:

- W_FROM_WK: initializes the phrase index (PhI) to zero, to prepare the extraction of words from the working-phrase buffer;
- NEXT_W(N1 times): skip N1 words of the working phrase buffer;
- FLUSH_WG: clear the content of the word-group buffer;
- GET_W(N2 times): copies N2 consecutive words from the working phrase buffer to the word-group buffer;
- RETR_AS: retrieve a phrase associated to the word group by the association mechanism.
  Additionally, the system can eventually retrieve the starting phrase of the same context of the working phrase:
- GET_START_PH
  and retrieve sequentially the phrases belonging to the same context
- GET_NEXT_PH
  N1 and N2 are random integer numbers. N1 can eventually be null, while N2 must be greater than or equal to one. The range of N1 and N2 depends on the maximum phrase size (ten words in the

current implementation).
The teacher can suggest to the system a target phrase or a target word group. The exploration process is terminated when it produces the target phrase / target word group, or when the number of iterations becomes greater than a predefined limit.
For instance, if the teacher types the command:

```
.wg mammal
```

the system starts an exploration phase, which terminates when the target word group "mammal" is extracted from the working phrase buffer. Therefore, the command:

```
.ph the dog is a mammal
```

starts another exploration phase that is terminated when the working phrase, which is retrieved from the word group through the association mechanism, becomes equal to the target phrase. At this point, the teacher can type the command:

```
.wg dog
```

The system will start a new exploration phase, which terminates when the target word group "dog" is extracted from the working phrase buffer. The word group corresponds to a good output, so the teacher can reward the system using the command

```
.rw
```

During the reward phase, the system retrieves the state-action sequence that led to a good output. The association between each state of the sequence and each corresponding action is rewarded by changing the link weights of the state-action association SSM through a discrete version of the Hebbian learning rule.
It is possible to use the monitor tool to display the sequence of actions and the corresponding system states that led to the desired output. For instance the following commands could be used before the reward:

```
.monitor display Header reward on
.monitor display InPhB reward on
.monitor display WkPhB reward on
.monitor display WGB reward on
.monitor display OutPhB reward on
.monitor display StActI reward on
.monitor display ElAct reward on
```

Finally, the teacher can ask the system to say an animal belonging to a category different from the one used for training it, e.g.

```
tell me a reptile
```

and start the exploitation operating mode through the command

```
.xr
```

During the exploitation phase, the state-action association SSM receives as input the state of a subset of the system SSMs, and it is updated through the k-winner-take-all rule, i.e. the k neurons with the highest activation state are switched on, while all the remaining neurons are left off. In the current implementation, a value of k=5 was chosen. The state-action association SSM sends its output to the elaboration-actions SSM, which is updated through the winner-take-all algorithm: the neuron with the highest activation state is switched on, while all the others are left off. In this way a single elaboration action is selected, the one that is more represented among the output of the k winners of the state-action association SSM.
At the end of the exploitation phase, the system will respond with a correct output.

### 3.4.2 Second example: verbs and personal pronouns

In this example the system should combine the use of some verbs with that of personal pronouns.
The teacher can start by typing the two phrases:

```
the personal pronoun for a male person is he
the personal pronoun for a female person is she
```

then he should type phrases as

```
Susan is a female name
Susan is a doctor
Susan is drive -ing the car
Tim is a male name
Tim is a student
Tim is read -ing a book
Elisabeth is a female name
Elisabeth is a secretary
Elisabeth is write -ing a letter
Max is a male name
Max is an actor
Max is go -ing to the theatre
...
```

and other similar phrases with different names. Again, the order in which the phrases are submitted is not important, and they can be mixed with other kinds of phrases.
Now the teacher can ask the question

```
? what is Max do -ing
```

starting with a question mark, without other punctuations, and following the rule discussed previously for words with suffixes. Then he can suggest target word groups and target phrases that lead to the correct answer. Since he would like that the system uses the personal pronouns, the first part of the output should be the word "he", which can be obtained through the following word-group extractions and associations:

```
.wg Max
.ph Max is a male name
.wg male
.ph the personal pronoun for a male person is he
.wg he
```

The word group obtained at the end of this exploration phase is the first part of the output. It should be rewarded, however the system should be warned that the output phrase is not complete. The command that the teacher should use in this case is

```
.prw
```

The only difference between the partial reward and the complete reward is that the first is terminated by a CONTINUE action, while the latter is terminated by a DONE action.
To complete the answer, the teacher can suggest the following word-group extractions and associations:

```
.rip # retrieve the input phrase
.wg Max
.ph Max is go -ing to the theater
.wg is go -ing to the theater
```

The last word group is the second and final part of the desired output, therefore it should be rewarded:

```
.rw
```

Now the teacher can test if the system is able to answer to similar questions, as for instance:

```
? what is Tim do -ing
.x
```

The system will answer:

```
he is read -ing a book
```

Note that it is not necessary to train the system with an example using a female name: the system will be able to use correctly both personal pronouns according to the gender. For instance, if the teacher asks the question:

```
? what is Elisabeth do -ing
```

the system will answer correctly, being able to use correctly both personal pronouns according to the gender.

## 3.5 Interface Commands

The following is the complete list of the commands that can be executed through the interface. All commands start with a dot. Many commands have an abbreviated form, which is indicated besides the extended form between parenthesis.

`.phrase(.ph) phrase`
Suggests a phrase to be retrieved by the association process.

`.word_group(.wg) word group`
Suggests a word group to be extracted from the working phrase.

`.partial_reward(.prw, .po)`
Sends the word group to output and produces a partial reward for the past state-action sequence.

`.reward(.rw, .o)`
Sends the word group to output and produces a (conclusive) reward for the past state-action sequence.

`.exploit(.x) phrase`
Starts an exploitation phase. At the end of this phase, the system is expected to respond to the input phrase with an appropriate output phrase. Without argument, the exploitation is related to the previous input phrase.

`.file(.f) file_name`
Loads phrases and/or commands from the file file_name.

`.clean_exploit(.cx) phrase`
Clear the goal stack and starts an exploitation phase. At the end of this phase, the system is expected to respond to the input phrase with an appropriate output phrase. Without argument, the exploitation is related to the previous input phrase.

`.exploit_random(.xr) phrase`
Starts an exploitation phase using a random ordering for the WTA rule. At the end of this phase, the system is expected to respond to the input phrase with an appropriate output phrase. Without argument, the exploitation is related to the previous input phrase.

`.exploit_memorize(.xm) phrase`
Starts an exploitation phase.
The output phrase is memorized by the system.

`.best_exploit(.bx) n_iter phrase`
Analogous to .x, but it iterates the exploitation process n_iter times and selects the best output phrase.

`.push_goal(.pg)`
Insert the working phrase and the current word group in the top of the goal stack.

`.drop_goal(.dg)`
Removes the goal phrase and the goal word group from the top of the goal stack.

`.get_goal_phrase(.ggp)`
Copies the goal phrase from the top of the goal stack to the working phrase.

`.sentence_out(.snto)`
Sends to the output the next part of the working phrase, after the current word, and all subsequent phrases of the same context until the end of the context itself.

`.search_context(.sctx) phrase`
Searches a phrase in the current context of the working phrase.

`.retrieve_input_phrase(.rip)`
Copies the input phrase to the working phrase.

`.get_input_phrase(.gi) phrase`
Gets the input phrase provided as argument without building the associations between the word groups and the phrase.

`.build_association(.ba)`
Builds the associations between the word groups and the working phrase.

`.action(.a) action_name`
Executes single elaboration action

`.time(.t)`
Displays the execution time

`.stat`
Displays some statistical information.

`.null(.n)`
Executes a system update with a NULL action.

`.monitor(.m) command`
Sends a command to the monitor

`.save file_name`
Saves all variable-weight links to a file.

`.load file_name`
Loads all variable-weight links from a file.

`.logfile file_name`
Writes the output to a log file.

`.logfile off`
Closes the log file.

`.speaker speaker_name(3 characters)`
Displays the speaker names using the CHAT format.

`.speaker off`
Stops displaying the speaker names.

`.period`
Displays a period at the end of the output sentences.

`.answer_time(.at)`
Records the answer time.

`.evaluate_answer_time(.eat)`
Evaluates the answer time.

`.auto_save_links(.asl)`
Periodically save variable-link weights in files with progressive numbers.

`.quit(.q)`
exits the program or the current input file

## 3.6 Simplification of the training procedure

The training procedure can be simplified by using macros, which execute some standard sequences of training commands.
All macros have the format:

```
command /cue/phrase/word group/
```

or

```
command /first cue & second cue/phrase/word group/
```

where command is one of the following commands:

`.ph` suggests a phrase to be retrieved by the association process;
`.wg` suggests a word group to be extracted from the working phrase;
`.pg` insert the working phrase and the current word group in the top of the goal stack;
`.po` sends the word group to output and produces a partial reward for the past state-action sequence;
`.o` sends the word group to output and produces a (conclusive) reward for the past state-action sequence;
`cue` is a word group that must be extracted from the current working phrase and used as a cue, phrase is the phrase that must be retrieved from the long-term memory using cue, and word group is an optional group of words extracted from phrase.
If phrase is omitted, the standard form of a phrase:

```
cue , word group
```

is retrieved from the long term memory. This form can be used to perform simple substitutions, e.g. to translate the second person to the first person

```
you , I
```

An optional secondary cue can be used after the main one using the "&" symbol.
An asterisk (*) next to the commands ".wg", ".po" and ".o" (which thus become ".wg*", ".po* or ".o*")* indicates that the working phrase stored by the system before the beginning of the macro should be retrieved at the end of the macro itself.
For instance, the training procedure of the example described in Sect. 6.1 can be simplified by using, after the training question-like imperative sentence:

```
tell me a mammal
```

the following macro:

```
.o /mammal/the dog is a mammal/dog/
```

Similarly, the training procedure of the example described in Sect. 6.2 can be simplified by using, after the training question

```
? what is Max doing
```

the following macros:

```
.ph /Max/Max is a male name//
.po /male/the personal pronoun for a male person is he/he/
.rip
.o /Max/Max is go -ing to the theater/is go -ing to the theater/
```

The two examples above can be found in the directory simple_examples .
After launching the main program:

```
annabell
```

they can be executed by issuing the commands

```
.f simplify1.txt
```

for the first example, and

```
.f simplify2.txt
```

for the second example.

## 3.7 The database and the files used for training/validation

The database, as well as the procedure used for cross validation, are described in the document Supplemental material.

### 3.7.1 People dataset files

Declarative sentences:

```
crossvalidation/template/people/people.txt
crossvalidation/template/people/people_howto.txt
```

Training input files:

```
crossvalidation/round<i>/people/train<j>.txt
```

Test input files:

```
crossvalidation/round<i>/people/test<j>.txt
```

`<i>`: cross validation round index (1-4)
`<j>`: file index

### 3.7.2 Parts of the body dataset files

Declarative sentences:

```
crossvalidation/template/body/body_what.txt
crossvalidation/template/body/body_where.txt
crossvalidation/template/body/body_howto.txt
```

Training input files:

```
crossvalidation/round<i>/body/train<j>.txt
```

Test input files:

```
crossvalidation/round<i>/body/test<j>.txt
```

`<i>`: cross validation round index (1-4)
`<j>`: file index

### 3.7.3 Categorization dataset files

Declarative sentences:

```
crossvalidation/template/skills/categories_list.txt
crossvalidation/template/skills/adjectives_list.txt
crossvalidation/template/skills/categories_howto.txt
```

Training input files:

```
crossvalidation/round<i>/skills/train<j>.txt
```

Test input files:

```
crossvalidation/round<i>/skills/test<j>.txt
```

`<i>`: cross validation round index (1-4)
`<j>`: file index

### 3.7.4 Communicative interactions dataset files

Declarative sentences:

```
crossvalidation/template/childes/david_sentences.txt
crossvalidation/template/childes/david_sentences3.txt
```

Training input files:

```
crossvalidation/round<i>/childes/david_train<j>.txt
```

Test input files:

```
crossvalidation/round<i>/childes/david_test<j>.txt
```

`<i>`: cross validation round index (1-4)
`<j>`: file index

### 3.7.5 Virtual environment dataset files

Declarative sentences and training input file:

```
crossvalidation/round<i>/virtual/bring_howto.txt
crossvalidation/round<i>/virtual/train<j>.txt
```

Test input files:

```
crossvalidation/round<i>/virtual/test_bring_files/test_bring_<j>.txt
```

`<i>`: cross validation round index (1-4)
`<j>`: file index

## 3.8 Running the cross validation and displaying the results

The following commands and bash scripts can be used in linux/unix platforms to run the training and the cross validation on the whole database and to check the results.
TRAINING:

```
cd crossvalidation/round1
./train_all_rand.sh
cd ../round2
./train_all_rand.sh
cd ../round3
./train_all_rand.sh
cd ../round4
./train_all_rand.sh
```

TEST:

```
cd crossvalidation/round1
./test_all_rand.sh
cd ../round2
./test_all_rand.sh
cd ../round3
./test_all_rand.sh
cd ../round4
./test_all_rand.sh
```

DISPLAY RESULTS:

```
cd crossvalidation/round1/randomize_check
./check.sh
cd ../../round2/randomize_check
./check.sh
cd ../../round3/randomize_check
./check.sh
cd ../../round4/randomize_check
./check.sh
```

CHILDES RESULTS:
in the directories:
`round1/childes, round2/childes, round3/childes, round4/childes`
check the file `log_all.txt`
You can compare it with `log_all.txt.correct`
VIRTUAL ENVIRONMENT INCREMENTAL RESULTS:
TRAINING:

```
cd crossvalidation/template/virtual_incremental_update/train_bring_run
./run.sh
```

TEST:

```
cd crossvalidation/template/virtual_incremental_update/test_bring_check
./run.sh
```

DISPLAY RESULTS:

```
cd crossvalidation/template/virtual_incremental_update/test_bring_check
./stat.sh
```