BERICHT
aus dem
PSYCHOLOGISCHEN INSTITUT
DER UNIVERSITÄT HEIDELBERG

Matthijs Kadijk

Plotting Activations in Neural Networks

Oktober 1992
Institutsbericht Nr. 73

6900 HEIDELBERG 1, HAUPTSTR. 47-51, TEL. 06221/54 73 46

Plotting Activations in Neural Networks

Matthijs Kadijk*

September 1992

Abstract

A simple procedure to generate plots of unit's activation levels over time is described. It is designed for neural network simulations with the Rochester Connectionist Simulator, using the Mathematica or Gnuplot plotting facilities on Unix workstations. A sample program that simulates harmony theory networks is available from the author.

Artificial Neural Networks, also known as connectionist- or PDP-models, are of growing importance in psychological modeling (Rumelhart and McClelland, 1986; McClelland and Rumelhart, 1986). A connectionist model is built from simple computational units that are highly interconnected. The overall behavior of the network is determined by (1) the pattern of weighted interconnections between the units, along which activations are propagated, and (2) each unit's *activation function*, that defines how incoming activations are combined into a new activation level of this unit.

Using the Rochester Connectionist Simulator (RCS) (Feldman, Fanty, Goddard and Lynne, 1988; Goddard, Lynne and Mintz, 1988), both network characteristics (interconnection pattern and activation functions) can be defined by means of a user defined C program (Kernighan and Richie, 1978). This feature makes the RCS a highly flexible and powerful tool for designing and simulating a wide class of connectionist networks within the Unix environment. A graphic

*This research was supported by Grant Al 205/3 to D. Albert of the Deutsche Forschungsgemeinschaft. I would like to thank Jan Peter de Ruiter for extensive testing and some adaptations of harmony.c. Requests for reprints should be sent to Matthijs Kadijk (b35@vm.urz.uni-heidelberg.de), Psychologisches Institut der Universität Heidelberg, Hauptstraße 47-51, 6900 Heidelberg, Germany.

interface is available but it is not adequate for studying unit's activations over time.

In order to study the dynamic behavior of units in a connectionist model only a minor addition to the user defined unit activation functions is needed. The new code simply allows for saving the activation levels in an appropriate *log file*. The data in these log files can be post-processed and plotted using existing data analysis programs like SAS or Gnuplot.[1] In the following programming example the Mathematica[2] (Wolfram, 1991) `ListPlot` command is used for data visualization.

How it is Done

The changes in an existing RCS C program necessary to produce log files that contain a plot command with the form `ListPlot[`*ListOfActivations, Options*`]` are described. It is also shown how the actual plots are viewed. The proposed procedure is simple but effective because of its generality and flexibility. The programming example (listings 1 – 4) constitutes a complete RCS network simulation program. It may be very instructive for readers interested in neural network simulation but unfamiliar with the RCS.

Additions to the C Program

For each unit in the network a log file is created with a unique name e.g., `activation.<unit_name>` where `<unit_name>` is the name of a unit. Therefore a global array of file pointers and a function that takes care of naming and opening the files are defined (see listing 1). After calling `openlogfiles` the file pointer `logunit[i]` refers to the log file for the unit with index `i`. This file pointer is be used to log the new activation level each time it is updated.

In the RCS the updating of a unit's activation level (i.e. potential) is done by its unit function (e.g., `UFbox` as shown in listing 2). We simply add a line to the relevant unit function definition that logs the new activation level to the correct log file. Instead of the potential of a unit we could, of course, also log some other calculated value of interest (see figure 1).

To make the log file contain a correct Mathematica command a head and a

[1] Gnuplot is distributed by the Free Software Foundation, and is available on the internet

[2] Mathematica is a registered Trademark of Wolfram Research Inc.

Listing 1. Global variable definitions and an `openlogfiles()` function

```
#include "sim.h"    /* simulator definitions */
#define MAXLOG 100
FILE *logunit[MAXLOG];  /* array of file pointers for each unit */

void openlogfiles(){    /* name and open the log files */
   char logfile_name[80];
   int i;
   for(i=0; i<NoUnits; i++){
      sprintf(logfile_name,"activation.%s", IndToName(i));
      logunit[i] = fopen(logfile_name,"w");
/* Mathematica head: leave out for Gnuplot */
      fprintf(logunit[i], "ListPlot[{ {0, %f}",
                          (float)UnitList[i].potential/1000.0);
   };
}; /* end openlogfiles */
```

tail are needed.[3] The function `openlogfiles` writes the head of the command, just after opening the file. In the function `closelogfiles` (listing 3) we write the tail of the command. Here some options concerning the styling of the plot can be set. In this example the plot is supplied with a label, the points will be connected by a line, the autoscaling of the Y-axis is suppressed (for better comparability of plots), and the axes are labeled appropriately.

To put this all to work a function that builds a network with the modified unit functions is necessary. Listing 4 shows an example of a `build` function that constructs a simple fully interconnected network with random connection weights and random initial activations. In simulations it can be observed that some units stabilize over time while others show some kinds of oscillatory behavior.

[3] For a Gnuplot style log file no head and tail are needed.

Listing 2. A simple unit function that logs activation values.

```
int UFbox(up) /* brain-state-in-a-box activation function */
    Unit     *up;
{
    up->potential += SiteValue("NetInput", up->sites);
    if(up->potential < -1000) up->potential = -1000; /* min */
    if(up->potential >  1000) up->potential =  1000; /* max */
    up->output = up->potential;

/** line added to log the activations:  **/[a]

    fprintf(logunit[UnitIndex(up)], ",\n {%d, %f}", Clock,
                                    (float)up->potential/1000.0);

}; /* end UFbox */
```

[a] For a Gnuplot style log file the two numbers are logged without brackets and comma: `"\n%d %f"` instead of `"\n,{%d, %f}"`

Listing 3. In the function `closelogfiles()` the tail of the `ListPlot` command is constructed.

```
void closelogfiles(){ /* write tail and close log files */
  int  i;
  for(i=0; i<NoUnits; i++){
     fprintf(logunit[i],
           " }, PlotLabel->\"Unit %d\", %s, %s, %s]\n", i,
           "PlotJoined->True", "PlotRange->{-1,1}",
           "AxesLabel->{Cycle,Activation}" );
     fclose(logunit[i]);
  };
};
```

Listing 4. Example of a `build()` function.

```
#define RANDOM  500-random()%1001 /* between -500 and 500 */
#define NETSIZE 12   /* number of nodes  ... */

build(){ /* build fully connected random network */
   int  i,j,index;
   char unit_name[20];
   srandom(getpid());
   AllocateUnits(NETSIZE);

/* make units with random initial activation: */
   for(index=0; index < NETSIZE; index++){
      MakeUnit("output",UFbox,0,RANDOM,0,0,0,0);
      AddSite(index,"NetInput",SFweightedsum,0);
      sprintf(unit_name,"unit_%d",index);
      NameUnit(unit_name,SCALAR,index,0,0);
  };
/* make links with random weights between all units */
  for(i=0; i<NoUnits; i++)
      for(j=0; j<NoUnits; j++)
        if(i != j)
           MakeLink(i,j,"NetInput",RANDOM,0,NULL);
        /* links not symmetric: most networks do NOT stabilize */
  openlogfiles(); /* must be called before first GO !! */
};
```

Viewing the Graphs

After a simulation session with the RCS we have a separate file for each unit (artificial neuron) which contains a full Mathematica plot command. To execute this command we simply read it in into a Mathematica session using the << command[4] e.g., type:

```
<<"activation.unit_1"
```

to produce a plot of the activations of unit number 1.

Variations

This principle can be used to produce more complex data visualization commands by changing the head and the tail of the command. An example of this is to smooth the data using a running average procedure before plotting (see figure 2).

Instead of activations of units the same procedure can be used to produce plots of weight changes during a learning session (e.g., using the Back Propagation learning rule (Rumelhart, Hinton and Williams, 1986)).

An Example: harmony.c

A program is available from the author that implements the harmony theory model of Smolensky (1986). Harmony theory is closely related to the Bolzmann machine model and is used for solving constraint satisfaction problems. The binary valued network units are updated according to a stochastic updating rule. The method of simulated annealing (Kirkpatrick, Gelatt and Vecchi, 1983) is used to search for a global optimum of a harmony function, that measures the "degree of agreement" between units.

Description

The program provides a `build` function that reads a harmony network and an annealing schedule from a definition file. Simulations are run with the usual RCS commands. Commands for opening and closing the log files are provided and can be called during a simulation session. The user can chose between

[4]Gnuplot users can view the recorded data with the `plot` command e.g., `plot [] [-1:1] "activation.unit_1" with lines`.

constructing a Mathematica plot command or a plain Mathematica list of data points.

Plot commands are constructed for the overall value of the harmony function (see figure 2) and for each unit in the network (see figure 1). Note that not the actual activation (one of two states: ON or OFF) but the more interesting "probability of the unit being ON" is plotted. As a second feature curve smoothing using a running average transformation[5] is provided.

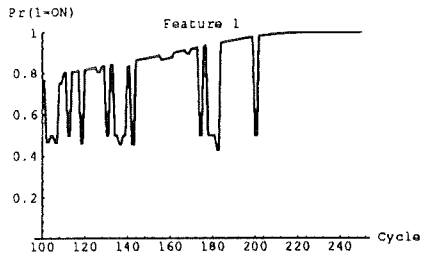

Figure 1. Example of a plot produced by harmony.c and Mathematica: "the probability of being ON" for a feature unit (representing the number 1 in a seven segment decoder network) is plotted for cycles 100 - 250.

Availability

A listing of the harmony.c program and basic documentation can be obtained from the author by e-mail (send requests to b35vm.urz.uni-heidelberg.de).

[5]Curve smoothing can be controlled by setting the Mathematica variable `Width`.

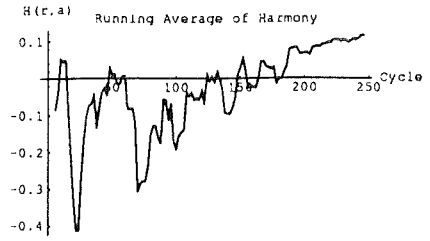

Figure 2. Example of a plot produced by harmony.c and Mathematica: the curve of the overall harmony in a seven segment decoder network, smoothed by a running average of 5 cycles.

REFERENCES

Feldman, J. A., Fanty, M. A., Goddard, N. H. & Lynne, K. J. (1988). Computing with structured connectionist networks. *Communications of the ACM*, *31*, 170–187.

Goddard, N. H., Lynne, K. L. & Mintz, T. (1988). *Rochester Connectionist Simulator*, User manual.

Kernighan, B. W. & Richie, D. M. (1978). *The C Programming Language.* London: Prentice-Hall.

Kirkpatrick, S., Gelatt, C. D. & Vecchi, M. P. (1983). Optimization by simulated annaeling. *Science*, *220*, 671–680.

McClelland, J. L. & Rumelhart, D. E. (Eds.) (1986). *Parallel distributed processing: Explorations in the microstructure of cognition: Vol. II. Psychological and biological models.* Cambridge, Massachusetts: MIT Press/Bradford Books.

Rumelhart, D. E. & McClelland, J. L. (Eds.) (1986). *Parallel distributed processing: Explorations in the microstructure of cognition: Vol. I. Foundations.* Cambridge, Massachusetts: MIT Press/Bradford Books.

Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986). Learning internal representations by error propagation. In D. E. Rumelhart & J. L. McClelland (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition: Vol. I. Foundations* (pp. 318–362). Cambridge, Massachusetts: MIT Press/Bradford Books.

Smolensky, P. (1986). Information processing in dynamical systems: Foundations of harmony theory. In D. E. Rumelhart & J. L. McClelland (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition: Vol. I. Foundations* (pp. 194–281). Cambridge, Massachusetts: MIT Press/Bradford Books.

Wolfram, S. (1991). *Mathematica, A System for Doing Mathematics by Computer. Second Edition.* New York: Addison-Wesley.

Appendix I
Harmony.c User Manual

Harmony.c: A Program that Simulates Harmony Theory Networks

User Manual

Matthijs Kadijk

How to use Harmony.c

Harmony.c is a RCS program that implements Smolensky's (1986) Harmony theory. In this basic documentation it is described how to use the program.

Authors

Harmony.c was written by Matthijs Kadijk (b35@vm.urz.uni-heidelberg.de) and adapted by Jan Peter de Ruiter (JanPeter@mpi.kun.nl).

Making the Simulator

Use harmony.c to build a RCS simulator using the command makesim:

```
makesim -f harmony.c
```

Be sure to make a floating point version of the simulator. The executable simulator can be run as usual with "sim".

Building a Harmony Network

In the RCS window use the command

```
call build <network.har>
```

to build the network as defined in the <network.har>. In this Network Definition File (NDF), the following parameters are defined:

I. annealing schedule:

   temperature at start
   temperature at end
   kappa at start
   kappa at end
   number of time steps

II. Network Architecture:

number of feature nodes (F)
names of feature nodes (one per line)
number of knowledge atoms (A)
knowledge atoms (one per line) in the form:

name sigma (float > 0) knowledge vector

A knowledge vector consists of F numbers (-1, 0, or 1) defining a negative, zero, or positive connection between this knowledge atom and all the feature nodes in the model (see Appendix A).

N.B. Knowledge vector values are -1, 0, or 1 after Smolensky, but this program allows for continues values between -1 and 1. This appears to be a useful extension of Smolensky's theory to build certain types of networks.

Only spaces and tabs may be used to separate the numbers.

Each node in the net network has to have a unique name.

Network Structure

There are 3 types of nodes:

1. feature nodes (potential = -1 or 1) indexed 0..F-1
2. knowledge atoms (potential = 0 or 1) indexed F..A+F-1
3. extra nodes:
   - temperature node (index A+F) pot = current temp. connected to all other nodes to send the value of temp.
   - kappa node (index A+F+1) pot = current kappa connected to all knowledge atoms to send value of kappa.
   - Harmony node (index A+F+2) pot = current Harmony connected to all knowledge atoms to receive partial harmonies.

The extra nodes are there to implement global variables needed for the annealing process. Use the standard RCS functions to examine the status of the nodes.

The state of a node can be 0 (free running) or 1 (clamped). To state a completion task to the network, just clamp some feature nodes to values of -1 or 1 (representing a feature being ON or OFF in the environment, the input) and let the rest of the feature nodes free (the output). Now simulate some time steps (GO) until the network cools down and the configuration on the free running nodes stabilizes. This configuration represents the answer of the model.

Plotting Activations

At the point of (simulated) time you want the plot to start type

```
-> call initlog <logfile> <logtype> <start#>
```

where <logfile> is the base file name of the logfiles to be opened, <logtype> is "m" for a full Mathematica ListPlot statement or "l" for just a (Mathematica) list of data points, and <start#> is the number the logcounter should start.
EXAMPLE:

```
-> call initlog plot m 1
```

results in the creation of a file "plot.<node>.1" for each node in the network with <node> is the name of the node as defined in the NDF (*.har).
Now set correct input configuration by clamping, and simulate as much time steps as required.
Now end the plot by issuing the closelog command:

```
-> call closelog
```

To start another plot with the same name but a higher number just call nextlog:

```
-> call nextlog
```

Do some more simulation and call closelog to end.
The plots can be visualized in Mathematica using the << command: just read in the file that contains the Mathematica ListPlot statement of that unit you want to see the plot. It is important first to read the definition of the RunningAverage2 function and to the set the curve smoothing parameter Width to the appropriate value before plotting (default value is 1).
EXAMPLE:

```
In[1]:= << RunningAverage2.m
In[2]:= Width = 5
In[3]:= << plot.harmony.1
```

If an error occurs you may have forgotten to close the logfiles with the closelog command.

Miscellaneous functions: Debugging

If you want to change harmony.c for your own purposes, you might want to set the debugging flag that causes the program generate same extra (useful ?) information. You can call debug_on and debug_off to turn debugging on or off, and the functions xdebug_on and xdebug_off to turn of or of extended debugging.

Appendix A: Commented NDF for a 7 segment decoder network

This is the file 7segment.har:

```
# N.B. all lines starting with # are comments:
#they cannot be processed by harmony.c !! strip them before using !

0.15  # temperature at start
0.0000001 # temperature at end
0.8  # kappa start
0.8  # kappa end
100  # steps

17  # features

s1  # the segments
s2
s3
s4
s5
s6
s7
0  # the numbers 0..9
1
2
3
4
5
6
7
8
9

10  # atoms

#name sigma  <------------ knowledge vector ----------->
#      <from the segments>    <- to the numbers ->
_0    1.0    1  1  1 -1  1  1  1    1 0 0 0 0 0 0 0 0 0
_1    1.0   -1 -1  1 -1 -1  1 -1    0 1 0 0 0 0 0 0 0 0
_2    1.0    1 -1  1  1  1 -1  1    0 0 1 0 0 0 0 0 0 0
_3    1.0    1 -1  1  1 -1  1  1    0 0 0 1 0 0 0 0 0 0
_4    1.0   -1  1  1  1 -1  1 -1    0 0 0 0 1 0 0 0 0 0
_5    1.0    1  1 -1  1 -1  1  1    0 0 0 0 0 1 0 0 0 0
_6    1.0   -1  1 -1  1  1  1  1    0 0 0 0 0 0 1 0 0 0
_7    1.0    1 -1  1 -1 -1  1 -1    0 0 0 0 0 0 0 1 0 0
_8    1.0    1  1  1  1  1  1  1    0 0 0 0 0 0 0 0 1 0
_9    1.0    1  1  1  1 -1  1 -1    0 0 0 0 0 0 0 0 0 1
```

Appendix B: Example session with 7segment

```
unix% sim
```

wait for the graphics interface to pop up ...

```
-> read 7segment
```

You will see the feature nodes representing the 7 segments, temp., kappa, and harmony nodes, knowledge atoms 0..9 and the feature nodes 0..9.

Go to custom mode and use the right mouse button to clamp a feature to ON (white) and the middle mouse button to clamp a feature to OFF (black). If the all of the 7 segments are thus turned ON or OFF to represent a number, simulate until the configuration is stabile (GO). In the end the only the knowledge atom and the feature node corresponding to the inputted number should be ON!

Now RESET the network (right mouse button) and try another number. It is also possible to clamp the 'output' features (0..9) to generate the image of a number on the 7 segment display.

N.B. There are several versions of 7segment.har some of them work, some of them don't! Having a look at those other versions may be very instructive. Since 7segment.har.noway did not work, I allowed for continues knowledge vector elements to come to this version of 7segment.har.

<MRK 1992>

Appendix II
Harmony.c Source Code

```
/****************************************/
/* HARMONY.C                            */
/* written by Matthijs Kadijk.          */
/* adapted by Jan Peter de Ruiter       */
/****************************************/


/* This program can be used to implement Smolensky's harmony theory */
/* in a Rochester Connectionist Simulator. */
                                                                    10

#include "/utils/rcs/include/sim.h"
/***************** global vars ************************/

#define MAXLOG 50

/* The two values the global variable FlipFlop can have */
#define AtomValue    0
#define FeatureValue 1
                                                                    20

/* GLOBAL VARIABLES */

FILE    *logunit[MAXLOG];           /* array of filepointers for each unit */
int     Gfeature_index;             /* number of opened feature logfiles, used incloselog() */
int     Gatom_index;                /* number of opened atom logfiles,
                                          used in openlog() and closelog() */
int     Gharmony_index;             /* idem concerning harmony logfile */

int     logging_on      = 0;        /* global variable, 1=on,  0=off */          30
int     MathPlot;                   /* global variable,        1= mathematica formula,
                                                               0 = list */
char  CurrentLogFile[40];           /* the current filename of the log-files */
int   FileCounter       = 0;        /* the extension counter of the logfiles */
int   DebugOn           = 0;        /* Debugging on/off switch */
int   XDebugOn          = 0;        /* eXtended Debugging on/off switch */
int   FlipFlop          = AtomValue; /* Flipflop var for the atom/feature sequencing */
/* int   RealClock      = 1;          Because of the weird 2-layer update,
                                          this Clockvalue is better fit for correct logging */
                                                                    40

/*************** utils *********************************/

#define newarray(TYPE,LENGTH)   (TYPE *)malloc((LENGTH+1)*sizeof(TYPE))
#define between(C,A,B)          (A <= C && C <= B) || (B <= C && C <= A)

#define FREERUNNING 0
#define CLAMPED     1

static int rnd()                                                    rnd
{                                                                   51
        return(500-random()%1000); /* between  -500 and + 500 */
};
```

sigmoid

```
static float sigmoid(x)
        float   x;
{
        float   s;
        float   exp();

        s = exp(-1.0 * x);
        s = 1.0 / (1.0 + s);
        return(s);

}; /* end sigmoid */


/****************** Site Functions ************/

/* Site function that calculates the input for a feature node
    I_i = 2 * sum_over_alfa { W_{i,alfa} a_alfa } .
        where W_{i,alfa} = (k_alfa)_i (sigma_alfa /norm(k_alfa) )
*/
int SFfeature(up,sp)
        Unit    *up;
        Site    *sp;
{
        float    sum;
        Link    *lp;

        for(lp = sp->inputs,sum = 0.0; lp != NULL; lp = lp->next)
        {
                sum += (*(lp->value) * lp->weight);
                if (XDebugOn)
                        fprintf(stderr, "val(%s)=%f,wght()=%f;\n",
                                up->name, *(lp->value),lp->weight);
        };
        sp->value = 2.0*sum;
        if (XDebugOn) fprintf(stderr, "new value = %f;\n",sp->value);
}; /* end SFfeature */
```

SFfeature

```
int SFget(up,sp)
        Unit    *up;
        Site    *sp;
{
        sp->value = *(sp->inputs->value);
}; /* end SFget */


int SFatom(up,sp)
        Unit    *up;
        Site    *sp;
{
```

SFget

SFatom

```
        float     sum;
        Link     *lp;

        for(lp = sp->inputs,sum = 0.0; lp != NULL; lp = lp->next) {
                sum += (*(lp->value) * lp->weight);                    110
                if (XDebugOn) fprintf(stderr, "val()=%f,wght()=%f;\n",
                                *(lp->value),lp->weight);
                }
        sp->value = (sum - up->data * SiteValue("Kappa",up->sites));
        if (XDebugOn) fprintf(stderr, "new value = %f;\n",sp->value);

}; /* end SFatom */

int SFharmony(up,sp)                                          SFharmony
        Unit     *up;                                                  120
        Site     *sp;
{
        float     sum;
        Link     *lp;
        Unit     *fromAtom;      /* points to unit from wich
                                        activation comes over the link */

        for(lp = sp->inputs,sum = 0.0; lp != NULL; lp = lp->next)
        {
                fromAtom = &(UnitList[lp->from_unit]);                 130

                sum += (fromAtom->output) * SiteValue("Input",fromAtom->sites);
        };
        sp->value = sum;
}; /* end SFharmony */


int SFhold(up,sp)                                                SFhold
        Unit     *up;
        Site     *sp;                                                  140
{
        sp->value = sp->value;
        sp->data  = sp->data;
}; /* end SFhold */

/********************** Unit Functions ********************************/

/*
        The state field can be FREERUNNING (0) or CLAMPED (1).
        The potential field of a knowledgeatom can be  0.0 or 1.0.     150
        The potential field of a feature node  can be -1.0 or 1.0.
        The output of a unit is always its potential.

        The data field of a knowledge ateom unit is used to represent
        the strength parameter sigma_alfa!
        As a (more elegant?) solution the sigma could be stored in a Site:
        AddSite(index,"Sigma",SFhold,0);
```

```
        . UFsigmoidFeature() UFsigmoidAtom() decide whether or not to update
        the nodes potential and output values acourding to the global FlipFlop
        value. A better solution would be to create an extra node, that fips
        between to states every RCS cycle, and propagates this value to all
        other nodes in the network.
*/

int UFsigmoidFeature(up)          /* Feature Node takes values from {-1,1} */   UFsigmoidFeature
        Unit *up;
{
        float   InpTempFraction, prob1;  /* probability that activation = 1 */
        int     random();                /* from libc */
        int     StoredRandom;

        if((FlipFlop == FeatureValue) && (up->state == FREERUNNING))
        {
             InpTempFraction = SiteValue("Input",up->sites) /
                               SiteValue("Temperature",up->sites);
             prob1 = sigmoid(InpTempFraction);
             if(SiteValue("Temperature",up->sites) == 0.0)
                  prob1 = 0.5;
             StoredRandom = random()%1000;
             if( ((float)(StoredRandom) / 1000.0) < prob1 )
                   up->potential =  1.0;
             else  up->potential = -1.0;
             up->output     = up->potential;

 if(DebugOn)
        fprintf(stderr,"probability for %s  = 1 is sigmoid(%f / %f) = %f. Random = %i\n",
                 up->name, SiteValue("Input",up->sites),
                 SiteValue("Temperature",up->sites), prob1, StoredRandom);


        }
        else { /* probability should be calculated for logging purposes */
                    InpTempFraction = SiteValue("Input",up->sites) /
                                      SiteValue("Temperature",up->sites);
             prob1 = sigmoid(InpTempFraction);

        } /* else */

/*  log  prob1 values for use in a mathematica listplot !! */

        if (logging_on /*&& (FlipFlop == FeatureValue)*/)  {
                fprintf(logunit[UnitIndex(up)],",\n {%f, %f}",(float)Clock,prob1);
         } /* if */

}; /* end UFsigmoidFeature */


int UFsigmoidAtom(up)  /* Knowledge Atom takes values from {0,1} */      UFsigmoidAtom

        Unit *up;
```

```
{
        float   InpTempFraction, prob1;  /* probability that activation = 1 */
        int     random();                /* from libc */

        if((FlipFlop == AtomValue) && (up->state == FREERUNNING))
        {
            InpTempFraction = SiteValue("Input",up->sites) /
                              SiteValue("Temperature",up->sites);
            prob1 = sigmoid(InpTempFraction);                                   220
            if(SiteValue("Temperature",up->sites) == 0.0)
                prob1 = 0.5;
            if( ((float)(random()%1000) / 1000.0) < prob1 )
                up->potential = 1.0;
            else  up->potential = 0.0;
            up->output    = up->potential;

 if(DebugOn)
        fprintf(stderr,"probability for %s  = 1 is sigmoid(%f / %f) = %f\n",
            up->name, SiteValue("Input",up->sites),                             230
            SiteValue("Temperature",up->sites), prob1);

        } /* if */

        else { /* even if unit is not updated, we should calculate the probability for logging purposes */
            InpTempFraction = SiteValue("Input",up->sites) /
                              SiteValue("Temperature",up->sites);
            prob1 = sigmoid(InpTempFraction);
        }; /* else */
                                                                                240
/* test ... log  prob1 values for use in a mathematica listplot !! */

        if (logging_on /*&& (FlipFlop == AtomValue)*/) {
            fprintf(logunit[UnitIndex(up)],",\n {%f, %f}",(float)Clock,prob1);
        }
}; /* end UFsigmoidAtom */

int UFanneal(up)                                                    UFanneal
        Unit *up;
{                                                                               250
        float   start,stop,steps, new;


            start = SiteValue("Start",up->sites) ;
            stop  = SiteValue("Stop", up->sites) ;
            steps = (float)SiteValue("Steps",up->sites);

            if(up->potential == stop)
                    up->output = up->potential = stop;
            else    {                                                           260
                    new = start + ((float)Clock * (stop - start) / steps);
                    if(between(new, start, stop))
                            up->potential = up->output = new;
                    else    up->potential = up->output = stop;
```

```
                }
}; /* end UFanneal */


int UFharmony(up)                                                    UFharmony
        Unit *up;                                                              270
{
        up->potential = SiteValue("Sigma",up->sites);
        up->output = up->potential;

        if (logging_on && (FlipFlop == FeatureValue)) {
        /*  log  harmony values for use in a mathematica listplot !! */
        fprintf(logunit[Gatom_index + 1],",\n {%f, %f}",(float)Clock,up->potential);
        }
        if (FlipFlop) {
                /* RealClock++; */                                               280
                /* FreezeFeatures(); UnFreezeAtoms(); */
                FlipFlop = 0;
                }
        else {
                /* FreezeAtoms(); UnFreezeFeatures(); */
                FlipFlop = 1;
                };
}; /* end UFharmony */

/***************** building the network ************************/                290

static int MakeFeatures(fp, temp_start)          /* build feature nodes */   MakeFeatures
        FILE    *fp;                             /* network desciption file */
        float    temp_start;
{
        Site    *sp;
        int      i, features, index;
        char     fname[50];

        fscanf(fp,"%d", &features);                                            300
        fprintf(stderr,"MAKEFEATURES: features to read: %d\n",features);
        AllocateUnits(features);            /* feature units needed */

        for (i=1; i <= features; i++)
        {
           fscanf(fp,"%s",fname);
           fprintf(stderr,"MAKEFEATURES: feature read: %s\n",fname);

           index = MakeUnit("Feature Node",UFsigmoidFeature,
                        0.0, 0.0, 0.0, 0.0, FREERUNNING, FREERUNNING);          310
           AddSite(index,"Input",SFfeature,0.0);
           sp = AddSite(index,"Temperature",SFget,0.0);
           sp->value = temp_start;
           NameUnit(fname,SCALAR,index,0,0);
        };
        return(index);
```

```
}; /* end MakeFeatures() */

                                                                        320
static int MakeAtoms(fp,features,kappa_start,temp_start)          MakeAtoms
/* build knowledge atoms */
        FILE    *fp;                    /* network desciption file */
        int      features;              /* number of feature nodes */
        float    kappa_start,
                 temp_start;            /* innitial annealing values */
{
        Site    *sp;                    /* new created site */
        int      i,f, atoms, index;
        float   *k_alfa;                /* generalized knowledge vector !! */      330
        float    sigma,
                 w;                     /* weight */
        int      norm;
        char     fname[50];

        fscanf(fp,"%d", &atoms);

        AllocateUnits(atoms);      /* atom units needed */
        fprintf(stderr,"MAKEATOMS: atoms to read: %d\n",atoms);
                                                                        340
        for (i=1; i <= atoms; i++)      /* units for knowledge atoms */
        {
           fscanf(fp,"%s %f",fname, &sigma);
           fprintf(stderr,"MAKEATOM: making atom: %s\n",fname);
           index = MakeUnit("Knowledge Atom",UFsigmoidAtom,
                 0.0, 0.0, sigma, 0.0, FREERUNNING, FREERUNNING);
           AddSite(index,"Input",SFatom,0.0);
           sp = AddSite(index,"Kappa",SFget,0.0);
           sp->value = kappa_start;
           sp = AddSite(index,"Temperature",SFget,0.0);                 350
           sp->value = temp_start;

           NameUnit(fname,SCALAR,index,0,0);

           norm = 0;
           k_alfa = newarray(float,features);      /* allow for float weights in knowledge vector */
           for (f=0; f < features; f++)            /* read knowledge vector */
           {
                fscanf(fp,"%f", &k_alfa[f]);
                norm += (k_alfa[f] != 0.0) ? 1 : 0;                     360
           };
           for (f=0; f < features; f++)            /* normalise the weights  */
           {
                w = (float)k_alfa[f]*(sigma /(float)norm);
                MakeLink(index,f,"Input", w, 0.0, NULL);
                MakeLink(f,index,"Input", w, 0.0, NULL);
           };
        };

        return(index);                                                  370
```

```
}; /* end MakeAtoms() */

static int MakeAnnealing(feature_index,atom_index,
                         T_start,T_stop,k_start,k_stop,steps)
        int     feature_index,atom_index;
        float   T_start,T_stop,k_start,k_stop;
        int     steps;   /* cooling schema */
{
        int     i,temp_node,kappa_node;                                    380
        Site    *sp;     /* pointer to new created Site */

        AllocateUnits(2);

        temp_node = MakeUnit("Temperature node ",UFanneal, T_start,T_start,0.0,T_start,1,1);
        sp = AddSite(temp_node,"Start",SFhold,0.0);
        sp->value = T_start;
        sp = AddSite(temp_node,"Stop", SFhold,0.0);
        sp->value = T_stop;
        sp = AddSite(temp_node,"Steps",SFhold,0.0);                        390
        sp->value = (float)steps;

        kappa_node = MakeUnit("Kappa node",UFanneal, k_start,k_start,0.0,k_start,1,1);
        sp = AddSite(kappa_node,"Start",SFhold,0.0);
        sp->value = k_start;
        sp = AddSite(kappa_node,"Stop", SFhold,0.0);
        sp->value = k_stop;
        sp = AddSite(kappa_node,"Steps",SFhold,0.0);
        sp->value = (float)steps;
                                                                           400
        NameUnit("temperature",SCALAR,temp_node,0,0);
        NameUnit("kappa",SCALAR,kappa_node,0,0);

        for (i=0; i <= feature_index; i++)        /* feature nodes must know T */
        {
                MakeLink(temp_node,i,"Temperature", 1.0, 0.0, NULL);
        };

        for (i=feature_index+1; i <= atom_index; i++)
        {                         /* knowledge atoms must know T and kappa */  410

                MakeLink(temp_node,i,"Temperature", 1.0, 0.0, NULL);
                MakeLink(kappa_node,i,"Kappa",      1.0, 0.0, NULL);
        };

}; /* end MakeAnnealing */

static int MakeHarmony(feature_index,atom_index)                  MakeHarmony
        int     feature_index,atom_index;
{                                                                          420

        int     i,harmony_node;
```

```
        AllocateUnits(1);

        harmony_node = MakeUnit("Harmony node",UFharmony, 0.0, 0.0 ,0.0, 0.0,1,1);
        AddSite(harmony_node,"Sigma",SFharmony,0.0);

        NameUnit("harmony",SCALAR,harmony_node,0.0);
                                                                              430

        for (i=feature_index+1; i <= atom_index; i++)
        {                           /* need links from all knowledge atoms */

                MakeLink(i,harmony_node,"Sigma", 1.0, 0.0, NULL);
        };

        return(harmony_node);
}; /* end MakeHarmony() */
                                                                              440

build(argc,argv) /* call build file */                                  build
        int argc;
        char * argv[];
{
        FILE    *dataf;
        int      feature_index, atom_index, harmony_index;

        float    T_start,T_stop,k_start,k_stop;
        int      steps;    /* cooling schema */                           450

        DeclareState("freerunning",FREERUNNING);
        DeclareState("clamped",CLAMPED);

        fprintf(stderr,"BUILD: building the network from file: %s\n",argv[1]);

        if(argc != 2)
        {
                fprintf(stderr,"BUILD: Usage: build <filename>\n");
                return;                                                       460
        };

        dataf = fopen(argv[1],"r"); /* open the network discription file */

        fprintf(stderr,"BUILD: reading annealing schema:\n");
        fscanf(dataf,"%f", &T_start);
        fscanf(dataf,"%f", &T_stop);
        fscanf(dataf,"%f", &k_start);
        fscanf(dataf,"%f", &k_stop);
        fscanf(dataf,"%d", &steps);                                           470
        fprintf(stderr,"BUILD: Kappa %f -> %f  in %d steps\n",k_start,k_stop,steps);
        fprintf(stderr,"BUILD: Temp. %f -> %f  in %d steps\n",T_start,T_stop,steps);

        feature_index = MakeFeatures(dataf,T_start);
        atom_index = MakeAtoms(dataf, feature_index + 1, k_start, T_start);
```

```
        MakeAnnealing(feature_index,atom_index,
                                T_start,T_stop,k_start,k_stop,steps);

        harmony_index = MakeHarmony(feature_index,atom_index);                  480

/* a few globals are updated over here : */
        Gfeature_index = feature_index;
        Gatom_index = atom_index;
        Gharmony_index = harmony_index;

}; /* end build */

initlog(argc, argv) /* initlog starts a series of logfiles */                  initlog
        int argc;                                                               490
        char * argv[];
{
/* error check */
        if(argc != 4) {
                fprintf(stderr, "INITLOG: Usage: call initlog <filename> <m/l> <start#>\n");
                /* option can be <m> or <l>.   m = mathematica statement,
                                               l = a simple (mathematica) list */
                /* The ball game is that when another (wrong) character is entered, m will be assumed
                /* <start#> is an integer that specifies at what number the logcounter must start */
                return;                                                         500
        }
        if (argv[2][0] == 'm') MathPlot = 1;      /* Global MathPlot indicates what kind of logging is o
                                                        for the closelog function */
        else MathPlot = 0;

        FileCounter = atoi(argv[3]);

        strcpy(CurrentLogFile,argv[1]);
        openlog(CurrentLogFile,FileCounter); /* a Global, mind you! */
        fprintf(stderr, "INITLOG: Just opened logfiles %s.*.%d;\n", CurrentLogFile,FileCounter)$10
};
/* end initlog */

nextlog()                                                                       nextlog
{
        if(logging_on)  {
                fprintf(stderr, "NEXTLOG: ERROR: First call closelog to close previous files...\n"
        }
        openlog(CurrentLogFile,++FileCounter);
        fprintf(stderr, "NEXTLOG: Just opened logfiles %s.*.%d;\n", CurrentLogFile,FileCounter)$20
};
/* end nextlog */


openlog(FileName, Number_of_Logfile) /* open a specified log file */           openlog
char * FileName[40];
int Number_of_Logfile;
```

```
{
        int i;                                                                  530
        int atom_index;
        int harmony_index;
        char command_string[40];
        atom_index = Gatom_index; /* Gatom_index is a global */
        harmony_index = Gharmony_index; /* Gharmony_index too */


        for(i=0; i<= atom_index; i++)
        {
                sprintf(command_string,"%s.%s.%i",FileName,IndToName(i),Number_of_Logfile);   540
                logunit[i] =
                        fopen(command_string,"w"); /* open the log files */
                if(MathPlot)
                        fprintf(logunit[i],"ListPlot[RunningAverage2[Rest[{{0,0}");
                else
                        fprintf(logunit[i],"Rest[{{0,0}");
        };

        sprintf(command_string,"%s.%s.%i",FileName,IndToName(harmony_index),Number_of_Logfile);
        logunit[atom_index+1] = fopen(command_string,"w"); /* open harmony logfile*/   550
        if(MathPlot)
                fprintf(logunit[atom_index+1],"ListPlot[RunningAverage2[Rest[{{0,0}");
        else
                fprintf(logunit[atom_index+1],"Rest[{{0,0}");
        logging_on = 1;

}; /* end openlog() */

closelog() /* should be called by user at the end of a logging session ! */      closelog
        /* closelog() is also called in nextlog() */                            560
{
        int     i;

        for(i=0; i<= Gfeature_index; i++)
        {
        if(MathPlot)
                fprintf(logunit[i], " }],Width], %s, %s->\"Feature %s\", %s, \%s->{%s,\"Pr(%s=1)\"} ]\n
                        "PlotJoined->True", "PlotLabel",
                        "PlotRange->{0,1}", "AxesLabel",
                        IndToName(i),"Cycle",IndToName(i));                     570
        else fprintf(logunit[i], " }]");

        fclose(logunit[i]); /* close the log files */
        };
        for(i=Gfeature_index+1; i<= Gatom_index; i++)
        {
        if(MathPlot)
                fprintf(logunit[i], " }], Width], %s, %s, %s->\"Atom %s\", %s->{%s,\"Pr(%s=1)\"} ]\n",
                                "PlotJoined->True", "PlotRange->{0,1}", "PlotLabel",
                                "AxesLabel", IndToName(i), "Cycle",IndToName(i));   580
        else fprintf(logunit[i]," }]");
```

```
        fclose(logunit[i]); /* close the log files */
        };
        i = Gatom_index+1;
        if(MathPlot)
        fprintf(logunit[i]," }], Width], %s, PlotLabel->\"%s\", AxesLabel->{%s,\"%s\"} ]\n",
                "PlotJoined->True",
                "Running Average of Harmony",
                "Cycle","H(r,a)");                                                    590
        else fprintf(logunit[i], " }]");
        fclose(logunit[i]); /* close the log files */

        fprintf(stderr, "CLOSELOG: Just closed logfiles %s.*.%d;\n", CurrentLogFile, FileCounter);

        logging_on = 0;
};

debug_on() /* when called, it sets the global DebugOn to 1 */                  debug_on
{                                                                                   600
        DebugOn = 1;
        fprintf(stderr, "DEBUG_ON:Debugging turned on. Just you wait...\n");
};

debug_off() /* when called, it sets the global DebugOn to 0 */                 debug_off
{
        DebugOn = 0;
        fprintf(stderr, "DEBUG_OFF:Debugging turned off.\n");
};
                                                                                    610
xdebug_on() /* when called, it sets the global XDebugOn to 1 */               xdebug_on
{
        XDebugOn = 1;
        fprintf(stderr, "XDEBUG_ON: eXtended Debugging turned on. Just you wait...\n");
};

xdebug_off() /* when called, it sets the global XDebugOn to 0 */              xdebug_off
{
        XDebugOn = 0;
        fprintf(stderr, "XDEBUG_OFF: eXtended Debugging turned off.\n");          620
};

/***
ResetClock()
{
        RealClock = 1;
};

****/
```