Neural Network Models of Cognitive Conflict Paradigms

Dissertation

Zur Erlangung der Würde eines Doktors der Philosophie des Fachbereiches Pädagogik der Universität der Bundeswehr Hamburg

Vorgelegt von

Aquiles Luna-Rodriguez

Aus Hamburg

Hamburg, 2008

This thesis is dedicated to all those indispensable helpers
who make a thesis possible,
but have no thesis dedicated to them.

Neural Network Models of Cognitive Conflict Paradigms

Abstract

In recent years, some areas of cognitive psychology have proposed formal models in the form of computer simulations, using Back-Propagation Artificial Neural Networks (BP-ANNs). Such models represent an improvement in plausibility, and they allow quantitative results to be compared with empirical data.

After learning, BP-ANN's cells exhibit a fixed input/output behavior. Using the black box method, this is shown to be a fundamental problem. Cells without an internal state to represent short-time memory cannot account for the sequence of stimuli, nor for the time elapsed between stimuli. As a consequence, BP-ANNs -as well as other neural networks without state-dependant input/output- are inadequate as models of some important cognitive processes. These include classical conditioning, operant conditioning, and sequence effects in cognitive control.

Another major methodological problem is the use of free parameters. BP-ANNs cognitive models frequently use arbitrary amounts of cells, amounts of layers, connection structure, learning parameter value and other characteristics, without giving a theoretical justification.

Three methods are proposed here to solve these problems: first, the black box method is used to produce a cell's input/output behavior more similar to that of neurons. Second, the reverse engineering method is used to simulate as many neural features as possible. And, third, a genetic algorithm is used to eliminate arbitrary free parameters.

The use of these methods is illustrated through a series of spiking neural network models, implementing state-dependant input/output, spikes, refractory period, temporal summation, axon delay and synchronization of neuron groups. A genetic algorithm is used to choose the parameter values in another series of models.

Finally, the feasibility of following this research strategy using parallel computer hardware is discussed.

CONTENTS **page**

Preface -an informal history of how this work actually began

Many scientific works use the order of theory, method, results and discussion. This gives the impression of first having a theoretical question, then choosing a method to answer it, then collecting empirical data, and finally reaching a conclusion about the theory. Of course, in real life this rarely happens. Most experiments are never published, and the ones which are -specially the surprising ones- were sometimes performed before finding a theory which could explain the results. Since the preface is not formally a part of a scientific work, and I shall take here the opportunity to write in first person about how this work came to be.

During the season's vacation in 1983/84, I was working as assistant in the behavior laboratory in the Universidad de Costa Rica, training rats in standard classical and operant conditioning. It was clear that while the theory is straightforward, real rats behave in sometimes unpredictable ways, and expert conditioning is almost an art. I was worried that my students would have trouble (they did) and decided to write software to let the students practice with a computer-simulated rat first. Programming a rat in a Skinner box, using Burroughs-mainframe BASIC, rapidly proved unfeasible. But it was too late: my interest in behavior modeling was awaken, and persists to this day.

Early in 1984, I read an article in the mathematical games column in Scientific American (Hayes, 1984). It presented a promising theoretical framework that can be used to simulate neural systems. Nothing could be further away from the intentions of the author, for the column's subject were Cellular Automata. These are little-known systems, used mainly for entertainment by mathematicians and programmers, though they would eventually play a pivotal role in chaos research.

The normal use of the word "chaos" implies a disorderly, random behavior. Mathematicians mean rather a fully deterministic process without random components, where the behavior is complex but not necessarily disorderly. Many non-mathematicians have heard from the "butterfly effect", and tend to believe that chaotic behavior is always due to the exponential grow of very small differences in the initial state. This is not the case in cellular automata, where the initial state is absolutely precise, and all calculations are performed with integer numbers, avoiding the inherent inaccuracy of floating-point arithmetic.

The canonical Cellular Automaton is John Horton Conway's Game of Life. It consists of a two-dimensional array of identical cells, much like a chess board. The cells may have one of two possible states, alive or dead. To begin the game, a pattern (random or not) of living cells is placed in the array. Each cell has 8 neighbors around, like the squares a chess' king may reach in one move. The rules ("formula") are: if a cell is dead and exactly three neighbors are alive, the cell will be alive the next time step ("generation"). If a cell is alive, and two or three of its neighbors are alive, the cell stays alive on the next generation. In all other cases, a cell is dead on the next generation.

Despite being an extremely simple, discrete and fully deterministic system, the game of life develops an extremely complex behavior, and there is a mathematical proof (technically: Turing's halting problem) that it is ultimately unpredictable. In this paradoxical combination of determinism and unpredictability lies the interest for chaos research. For the rest of us, it is just fun to watch. It looks like a myriad of microbes moving under a microscope, that is why it is called game of life.

More seriously, Cellular Automata offer a close analogy of the brain: the cells are neurons, the neighbors are other neurons connected through synapses, and living cells are neurons firing. The behavior of Cellular Automata is defined by their formula, a simple table which combines the activity of the neighbors and the cell's own state. This is similar to a neuron combining pre-synaptic excitatory and inhibitory input with its own state (e.g., refractory period and calcium concentration) to determine whether to produce a spike or not. There are other similarities: Cellular Automata are not serial systems like a normal (Von Neumann architecture) computer, but parallel systems like the brain. Alas, the fact that neurons come in a confusing variety of forms and neurotransmitters, seems to contradict the complete uniformity of Cellular Automata.

Then, Scientific American came to the rescue again (Goodman & Bastiani, 1984). Their article explained how grasshopper embryonic neurons differentiate, depending on the neighbors around and their own state of development. After all, a human brain is only the ultimate product of a single cell with a single -digital- genetic code. An inhibitory neuron is like an automaton's cell that becomes trapped in a subset of its formula, with no combination of input and state producing a new state located outside this subset. This is fortunate, otherwise inhibitory neurons could begin to produce insulin or hydrochloric acid, which are also coded in their nucleus.

The ever-more-complex automata I developed following this insight, would be called by other researchers Artificial Neural Networks, but at the time I was unaware that such systems existed. One major difference persists when following the cellular-automaton-as-brain approach. Most conventional neural networks use cells with a single state. In other words, their output depends exclusively on the input, and lack a memory of what they were doing a time-step before. This single state is a major computational disadvantage. The mathematical model of computers, the Turing Machine, can be redefined in this context as a 1-cell, 0-dimensional cellular automaton, with the tape substituting the neighbors. To put it in drastic words: if Turing machines had only one state, they would not be the mathematical model of computers; and if neurons had no states, then there would not be short term memory nor temporal summation. It is surprising that cognitive modeling is still usually performed with single-state neural networks.

Over the years, these automata were coming always closer to the complexity of neurons, loosing in the process one of the most appealing features of automata: their simplicity. Stephen Wolfram, a pioneer in this field, published in 2002 an ambitious book on cellular automata (Wolfram, 2002), proposing them as a tool to understand the complexity of the universe, as developing from very simple systems. His approach had a somewhat controversial reception in the scientific community, but nevertheless it generated a short-lived renaissance in the field. Taking advantage of a fashionable trend, a paper describing some results was subjected to a cognitive simulation congress (Luna et al, 2003). It was a bit out of place at the congress, where most of the other authors presented models developed with traditional Artificial Intelligence tools, like ACT-R. Besides, the paper described jellyfish's locomotion, hardly a relevant subject for cognitive modeling at first sight. Maybe it was accepted simply because of curiosity about an exotic method. Hopefully, readers of the present work will share that curiosity.

When I started my doctoral work, the original goal was to develop more elegant models than the ones dominating the cognitive control field, trying to simulate various effects simultaneously within a single model. The more I studied these models, the more unsatisfied I became with them. For a long time I could not put in words the reason for this, and when I did, the unavoidable conclusion was that conventional BP-ANNs are a dead-end street for cognitive modeling. I have opted to argue for a paradigm change, and try to give a glimpse of how future models should look like, and which methods can be used to develop them. Looking back at the history of neural models, a paradigm change is a very ambitious goal, and I have been advised more than once how difficult this approach is. I have chosen to make the effort, because I could not bring myself to develop easy-but-scientifically-doomed models with a quiet conscience.

1.1 Goals and hypotheses of this work

The general goal of this work is to analyse which neural features should be considered relevant for models of cognitive conflict, and propose methods to improve the plausibility and usefulness of such models. The main concern is methodological, focusing on the research strategies, techniques and assumptions involved in the development and validation of cognitive models.

The first hypothesis claims that to account for sequence and latency effects, the cells in neural network (ANN) models of cognition should store information not only in the form of long-term synaptic weights, but also have a separate, internal, short-term state that influences their input/output behavior.

The second hypothesis claims that such an internal state can be implemented by simulating only simple and well understood input/output properties of neurons, avoiding abstract programming devices.

The third hypothesis claims that a biologically detailed simulation is practical using modest computer resources, by applying programming techniques from cellular automata, like look-up tables indexed by state and input.

The fourth hypothesis, concerning not single neurons but rather groups of them, claims that cognitive contents (perception of input, active task, motor output, etc.) can be represented in a model by groups of cells spiking in temporal close proximity, i.e., synchronized.

1.2 Structure of this work

The models presented in this work belong to the field of cognitive neuroscience. This is an interdisciplinary field, where research in cognitive psychology, neurobiology, computer science and other sciences are combined. One of the forms that such research takes, is the development of formal models, mainly in the form of computer simulations.

Due to the interdisciplinary character of this work, no optimal order of the different parts of the theory (see part 2) exists, as it would depend on the particular area of expertise of the reader. The chosen order is a compromise, it has some abrupt changes of subject and sometimes it does not follow the conventional order. For example, the description of the black box method is found at the start of the theory and not in the method part, otherwise the nature of the problems that this work tries to solve cannot be easily understood.

The main focus of this work is the methodology used to develop models of cognitive processes. The goal is to analyse the methods and techniques commonly used, find flaws in them, and propose new methods for future models. Within this context, the models presented here should be understood simply as *examples* of the use of methods, and not as models of definite cognitive processes. Thought if may seem contradictory, it can be argued that the proposed methods themselves are the most important results of this work.

In the following, various versions of computer programs are provided. Most of the parameters that define a specific model are not part of the program itself, instead they are defined by the user in the form of two human-readable data files, loaded at the start of each simulation. Within this context, the term "program" refers to the simulation software, i.e., an executable program with the extension .EXE. A "model" consists of the two data files (STRUC.TXT and PARAM.TXT) defining respectively the connection structure of the network, and the input/output behavior of the cells. To use in this way the word “model” for a set of parameters, does not necessarily implies that it is a model of a real-world object, unless explicitly stated. Finally, the term "simulation" is used here to describe the results of the software running, after it has loaded the data files. A simulation may consist of results of the program running several times, when it is run only once it may be called simply a "run".

The programs used to perform the simulations are the result of continuous development, in principle they are successive versions of a single program that has been changed mostly in small, incremental steps. Between changes, many models were built and tested, of which only a small fraction will be presented. The process has been continuous, alternating phases of program and model development.

The simulations are presented in two series. The first one consists of simulations of the most simple neural processes. The reasons to choose those processes and the specific implementation of neural features are explained, followed by the results and a discussion.

In the second series, the theoretical grounds to simplify the simulation from neurotransmission to ions-only is explained. Using this model as basis, a genetic algorithm framework is proposed, its theory and methodological advantages explained. This is followed by the methods used to produce a prototype program, the selection procedures used, the statistical problems of measuring synchronization, and the development of a fitness function. Many of these issues are illustrated by simulations.

Finally, there is a general discussion, bibliography, appendices and a CD-ROM that includes all versions of the program and all models, which the reader may run him/herself.

2 THEORY

2.1 The black box method in psychology

The black box is a method used to perform input/output analysis of systems, and it is most frequently employed in engineering. It is known to psychologists mainly as the theoretical background used in experimental analysis of behavior (radical behaviorism). The Turing test is also an example of this method, and a behaviorist may be tempted to describe it as behavior analysis.

The box is "black" because its inner components are either not available for inspection, or simply ignored. To build a model -or a copy- of the system being analysed, how the box actually functions is of no interest, as long as the model can reproduce exactly the input/output behavior of the original.

This method underlies most of present work, it used to analyse neural network models of cognitive conflict, and to propose solutions to flaws found in these models. A simple example of the method, adapted from an argument of philosopher of science John Searle (1983) follows.

The system to be analysed is a soft-drink vending machine, selling bottles for a price of one dollar. For an input of one dollar coin, the output is one bottle (ignoring the case of a sold-out machine), and for this, the machine needs no internal states, it could be a single-state machine. However, the machine also accepts half-dollar coins, and after an input of a single one, there is no output. After the second half-dollar, the output is one bottle. The unavoidable conclusion is that the machine has at least two different states, one that produces an output after an input of 50 cent, and one that does not. This is described as state-dependant input/output.

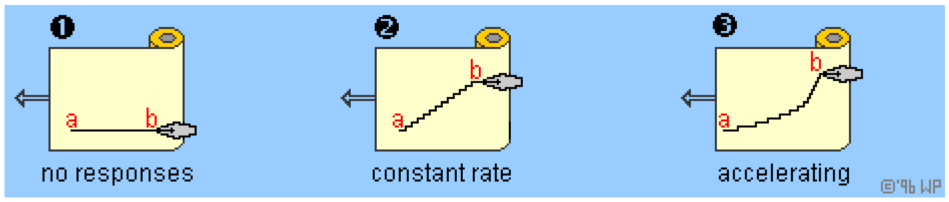
Suppose now that the machine also accepts quarters (25 cent coins). The next input for the same machine has the sequence half-dollar, quarter, quarter, and the output is a bottle. The next input/output sequence is quarter, half-dollar, bottle. In this case, there should be at least four internal states: "empty", "25 cent", "50 cent" and "75 cent", but the state "25 cent" seems to be missing, and the first coin probably sets the machine to state "50 cent" even if the coin was a quarter. One may try other sequences to test this hypothesis, yet the black box method already indicates an error in the design or functioning of the machine, because the order of the inputs should play no role for the output of a vending machine. Notice that for this analysis it is completely irrelevant if the machine is purely mechanical or microprocessor-controlled.

In psychology, the sequence of input stimulus, and the time lapsed between them, often has a critical influence over the output. The description of classical and operant learning below, is based on chapter 8 of the textbook from Zimbardo and Gerrig (2008). In the learning paradigm of classical conditioning (popularly known as Pavlov's dog) an organism learns a relationship between two inputs, the conditioned stimulus and the unconditioned stimulus. The canonical example is that of a dog that repeatedly hears a bell a minute before receiving food. After this is done a number of times, the dog begins to salivate after hearing the bell, but before receiving the food. Before conditioning, salivating is the unconditioned response to the food, after learning salivation is produced by the sound of the bell and is called a conditioned response.

The sequence in which the two stimuli -bell and food- are presented, determines the magnitude of the conditioned response, and its resistance to extinction (extinction consists in repeatedly presenting the bell alone without food, and eventually leads disappearance of the conditioned response). If the bell and food are presented simultaneously, the conditioning is weak at best and is rapidly extinguished. If the bell is presented one minute *after* the food, no conditioning occurs. If the bell sounds one hour before food is presented, no conditioning occurs. Obviously, classical conditioning is highly dependant both on the sequence of inputs and the latency (time elapsed between the two stimuli). It follows that a neural network model of classical conditioning needs to account for input sequence and time. In other words, besides the long-term memory represented as the changes in the connection weights, it also needs some form of short-term memory. This point will be discussed in detail later, within the description of BP-ANNs.

The other learning paradigm in experimental analysis of behavior, is operant conditioning. The main difference is that in classical conditioning one stimulus is followed by a response (bell -> salivation), while in operant conditioning a behavior is followed by a consequence. The canonical example here, is a rat that learns to press a lever to operate a mechanism which provides the rat with a small portion of food. The food is in this case a "positive reinforcer", because it increases the probability of the rat to show a given behavior (pressing the lever). Critical for this conditioning is the latency, defined in this case as the time lapse between a behavior and its consequence. Short latencies need few trials to condition a behavior, medium latencies need more trials, and too long latencies produce no conditioning.

The behavior required for a reinforcement can be manipulated with different reinforcement schedules. Pressing the lever once, followed by food, is a reinforcement schedule named "fixed ratio one". If the rat must press the lever ten times, it is called "fixed ratio ten". If the rat receives a reinforcement after pressing the lever ten times *on average*, it is on a "variable rate ten" schedule. Another class are "period" schedules, where a single behavior is reinforced only after a given period of time following the last reinforcement, and no behavior is reinforced before. If this period is one minute, the schedule's name is fixed period one minute. If the period is varies from trial to trial, with an average of one minute, the schedule's name is variable period one minute. The four basic reinforcement schedules, fixed ratio, variable ratio, fixed period and variable period, produce different rates of behaviors by unit of time. They also show characteristic patterns of behavior, which are virtually undistinguishable over a wide variety of organisms that includes humans, rats an pigeons.



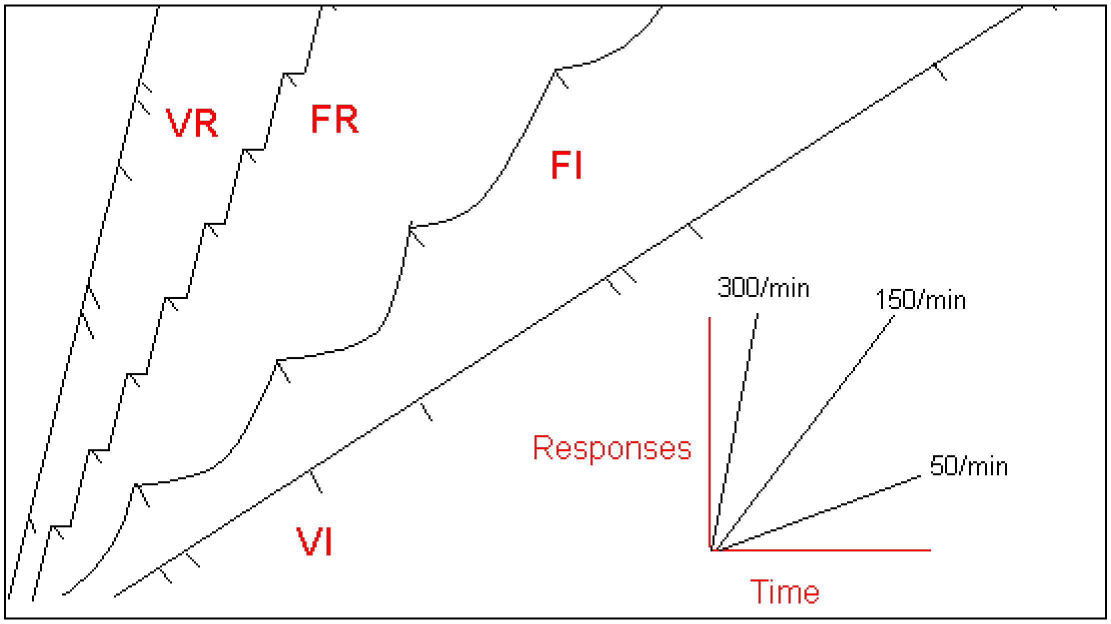


Fig. 1 Above: schema of a cumulative recorder, used to record behavior units (usually lever presses) in behavioral laboratories.
Below: Schedules of reinforcement. VR: variable ratio. FR: fixed ratio. FI: fixed interval. VI: variable interval. The short diagonal lines mark reinforcements. The step-like horizontal lines in FR are periods without activity. Taken from:
http://www.schouppe.net/wpl/psychologie/persoonlijkheid/theorie/gedragstheorie/operante.htm

From the viewpoint of the organism, the world is a black box where the sequence of inputs (lever presses) and outputs (food portions) are dependant on both sequence and timing. Analysing the organism itself as a black box, the input sequence and timing (reinforcement schedule), and reinforcement latency, play a critical role to determine the output (the organism's behavior). A very interesting example of this is superstitious behavior (Skinner, 1947), where the organism is given food at fixed time intervals, regardless of the organism's behavior. If the timing is right (around 15 seconds for pigeons) one particular behavior will increase in probability of appearance despite having no connection whatsoever with a consequence, and this behavior differs widely for different individuals (pecking in one corner, twisting the head, wing flapping ,etc.).

It is obvious that a reasonable model of operant behavior, should account for sequence and timing of both input and output. However, despite the existence of a class of neural networks using an unsupervised learn algorithm named "reinforcement learning", no neural network known to the author is able to reproduce the characteristic patterns of behavior for the various reinforcement schedules or superstitious behavior. Reinforcement schedules can be mixed, and virtually unlimited combinations of sequences and timings could be used to test the validity of a model. Yet, to put it bluntly, from the viewpoint of black box analysis there is simply no model of operant conditioning.

Learning is a subject of great interest both for psychology and neural networks, but is not the subject of this work, which is constrained to cognitive conflict. Within this field there is usually no new learning involved beyond the initial phase of practice, which is almost never analysed in the results. To use an hypothetical example, in an experiment the subject is instructed to press a given key on the computer keyboard when he/she sees a green letter, and a different key when a red letter is shown. A practice phase is started, until the experimental subject has reached a given average reaction time and/or a given maximum error rate. Only after this phase of practice, the proper experiment starts.

One task frequently used in the study of cognitive conflict, is the Stroop task, named after a classical experiment (Stroop, 1935). In this task, the experimental subject must name the color in which a word is printed, but the word itself is the name of a different color. Because reading is a highly trained task, the word interferes with the task of naming the color in which the word is printed, and produces slower reaction times compared with naming the color of geometrical figures. One of the possible causes of this and other types of cognitive conflict, is the simultaneous activation of incompatible responses. Additional variations can be used, like compatible, incompatible and neutral conditions, respectively "RED" in red color, "RED" in blue color, and "DOG" in red color. In the normal configuration of this task, the word used as irrelevant stimulus in one item is never used as the color to be named in following item.

Another way to study cognitive conflict is to use the negative priming effect. In this experimental configuration, two stimuli are combined, one relevant and one irrelevant to the task, as it is done in the Stroop task. The irrelevant stimulus in item n-1 is used as target for stimulus n. This is a sequential effect that slows down the reaction time to the target, and can be used likewise in compatible, incompatible and neutral conditions. Combining the Stroop and the negative priming effects, a series of items could be: "BLUE" in red color, "GREEN" in blue color, "YELLOW" in green color, "GREEN" in yellow color, and so on.

Most cognitive psychologists treat these two effects as separate ones, and try to develop theories or models for each one separately, following a general strategy informally known as "divide and conquer". Human cognition is deemed to be so complex, that to eventually understand it we should try to understand first all the different processes individually. It may be the case that the Stroop effect and the negative priming effect are due to unrelated cognitive subsystems, but sooner or later cognitive psychology will be forced to merge all the subsystems if we expect to produce a comprehensive theory or model of cognition. In any case, for a black box analysis it is evident that the negative priming effect alone is evidence of state-dependant input/output, and requires multiple internal states of the box. Again, when the Stroop and the negative priming effects are combined, it makes no sense to treat them as separate boxes, since both stimuli and task are the same. A proper model of cognitive conflict should account for the actual input/output of the system, regardless of which input sequence is used.

2.2 What is a model?

It is very difficult to give a general definition of the concept "model" because it is used with very different meanings in different fields or even within a given field. It may designate a physical object, a set of mathematical formulas, a common language description of a process, or a computer program. The characterization and taxonomy of models is a task not for a particular scientific discipline, but rather for epistemology or the philosophy of science in general.

According to Stachowiak (1973), a model is always an image, example or representation of something (the "original"), that has the following attributes:

1) The representation attribute. Models are images, representations of a natural or artificial original, that itself may be a model (another way to refer to this attribute is isomorphism - "equal form", or homophormism -"same form").

2) The reduction attribute. Models do not have all properties of the original, but only those considered to be relevant for a restricted purpose.

3)The pragmatical attribute. Models are not uniquely assigned to an original. They substitute an original for a specific subject, in a specific time, for specific mental or material operations. (pp. 131 - 133). In the case of scientific models, this attribute means that the usefulness of a given model depends on the purpose of the researcher, usually understanding or predicting some behavior of the original.

Within theoretical psychology, Dörner (1994) proposed two additional distinctions. First, when a model is not static but represents a process, it is called "simulation" and is usually performed with a computer. Second, he points out that in the social sciences the word model is used with confusingly diverse meanings, and without a clear distinction between theory and model. Referring to Braithwaite (1963 pp. 224-341), Dörner suggests that the word "model" is used when:

"*1) A theory is so small and special, that the word "theory" sounds too ambitious.*

2) Half or completely formalized theories are very rare in the social sciences. One calls such theories "models", in order to distinguish them from not formalized theories.

3) A theory is not complete but rather is only a precursor form with missing parts." (Dörner 1994 p. 368)

Of particular interest for the computer models presented in this work, is the second point. Within this context, formalized means an unambiguous description, accurate enough to produce quantitative results. It requires an artificial language, like the ones used in mathematics, logic or computer programming.

A good example for the crucial formal/informal distinction is Hebb's rule, one of the most frequently used concepts in neural networks: *"When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased."* (Hebb, 1949 p. 62). This rule is neither a theory nor a model, it is only a non-formal hypothesis -that neurophysiology has proven to be true (Koch, 1997 p.207)- described using common language. Most psychologists may think that Hebb is clear enough and his rule is easy to understand, but for the purpose of making a formal model, this is not true.

There are many computer simulations that claim to use Hebb's rule, but they are implemented in different ways and deliver different quantitative results. This happens because the computer enforces a complete formalization of the rule. A computer is unable to accept a description like "near enough", "some growth process" or "one or both cells". The programmer is forced to disambiguate all such descriptions, and must choose exact values for all variables (e.g., "increased": by how much?) and constants. Since Hebb did not explained in detail how to do this, programmers end up with formal models that disagree with each other.

In the context of this work only formal models will be proposed, because the validation of a model is easier to perform when a model produces quantitative results, that can be fitted to data from experiments with humans. Before discussing which attributes should be represented in cognitive models, the methodological question of how to validate models is addressed below.

2.3 Evaluation of models in psychology

According to Ueckert (1983 p.587), for computer-simulated models "The question of the validity of simulation models and their applicability is basically nothing else than the question about the relationship between theory and empirical reality, the same that applies generally for sciences as a methodological problem". Atkinson (1961 p. 46) puts it in a more blunt way: "At this point it would be nice if we could refer to a list of criteria and a decision rule which would evaluate the model and tell us whether this specific development or similar mathematical models are of genuine value in analysing the phenomena of interest to psychologists. Of course, such decision procedures do not exist."

While it is true that computer models carry no lower burden of proof than other forms of theory, the objects studied -models- are not part of the empirical reality but artificial ones. Herzog (1984, p. 94-96) argues on this ground that a model cannot be empirically proven or unproven, the relevant judgement is: useful or not. This is ultimately dictated by the pragmatic attribute of models, i.e., a model is useful (or not) only in relation with its purpose, which in Herzog's case (p.84-85) is theoretical insight (Erkenntnistheoretische Funktion).

Perhaps, instead of trying to find criteria to determine if a model is valid, it is more practical to look for criteria to declare a model invalid. Simon & Newell (1963), in analogy to inferential statistics' errors of type I and type II, propose two types of modelling error. Error of omission designates a relevant feature of the original, that is missing in the model. Error of commission designates a model's feature that is not present in the original. These errors are failures of the isomorphism attribute, an make model and original different. At the same time, omission and commission errors break the reduction attribute, by missing an original's relevant feature or including a non-existent feature.

There is no way to know in advance which features of the original are actually relevant. In the case of cognitive neuroscience models, this is an extremely difficult problem. Neurobiology has delivered an enormous amount of facts, to the point that a detailed model of a single neuron is a major challenge. Cognitive psychology has also produced scores of empirical data and effects, frequently without comprehensive theories to relate the many empirical facts. Finally, computer science faces the problems of the vast computer power required for the simulations, and also the non-trivial task of writing complex software without programming errors.

The result of this almost unmanageable complexity, is that models have to be extremely simplified representations of very constrained originals, and there is a difficult trade-off between the isomorphism and reduction attributes. Which of the original's features are deemed relevant has remained a problem without a definitive answer, and this work concentrates in identifying -and correcting- errors of omission in BP-ANN models. The original being modelled in this work is described in 2.5, how this model is formalized is described in section 3.1.

2.3.1 The problem of free parameters

Computer programs are a way to formalize a model, but they tend to be much more complex than mathematical formulas. Programs have many variables, and if arbitrary values are used to define them, the resulting model has many free parameters.

"Quantitative theories with free parameters often gain credence when they closely fit data. This is a mistake. A good fit reveals nothing about the flexibility of the theory (how much it cannot fit), the variability of the data (how firmly the data rule out what the theory cannot fit), or the likelihood of other outcomes (perhaps the theory could fit any plausible results), and a reader needs all 3 pieces of information to decide how much the fit should increase belief in the theory. ... A better way to test a theory with free parameters is to determine how the theory constrains possible outcomes" (Roberts & Pashler, 2000, p. 358).

While this criticism applies in general to psychological theories, it is particularly important for computer models, and indeed the authors use a BP-ANN model as an example. If a model has many free parameters, one may fine-tune their values until they fit the empirical data of a particular experiment. With another data set, the parameters may be fine-tuned again with different parameter values, and fit the data. However, the model has not predicted the results of two experiments. It has only produced two ex-post-facto explanations, and the explanations are different for both cases. Programmers have a saying for this criticism: "you can simulate anything", which does not mean that you have found a way to solve a general problem.

Roberts and Pashler (2000) propose that the models should show not only that a given configuration fits the data, i.e., make a prediction that is empirically tested. They demand an explicit range of behavior predicted, as well as behavior ruled out by the model. This demand can be fulfilled only if a range of values for each parameter are simulated separately, and a reason to choose the range's bounds is given. Besides the fact that for most BP-ANNs this would mean thousands of simulations, it can be expected that most of the model's behavior does not fit data.

The logical consequence of this argument is that there must be no free parameters, but rather each parameter value should have a explicit reason to be chosen, and this reason cannot be that it "fits the data".

The simulations performed in this work are implemented using a construction-set program, that by necessity has many more free parameters than usual ANNs. Two methods will be proposed later to respond to the many-free-parameters criticism. First, the laboratory approach of keeping all variables constant -however artificial they may be- except two or more values of a single independent variable. The second strategy is to use a genetic algorithm, that starts with random values for all free parameters, and changes this values according to an explicit fitness function.

2.4 A case example: astronomical models

One way to understand how models are developed, and which criteria are used to evaluate them, is to study a series of different models that represent the same original. It is also important that the purposes of these models, and the original's features represented, are the same or at least comparable.

Such requisites are difficult to fulfil in psychology, so first some examples from astronomy, the oldest science, will be used here before examining an example from cognitive psychology. One big advantage in solar system models, is that the features to be represented are clear from the beginning, i.e., the position and movements of celestial bodies. The purpose of the models is to calculate the positions in the future (like calendars or eclipse prediction) or the past (e.g., horoscopes). Another goal in some cases, is a deeper understanding of the universe in general.

The criteria used to judge models are not absolute, and each model is a compromise in this respect. Of the various criteria generally accepted in astronomy -and shared by psychology- four had been arbitrarily chosen, to illustrate the fact that each one of them has been ignored by an accepted astronomical model:

1) Plausibility, another word for the representation attribute, i.e., how isomorph is the model with respect to the original. This is unfortunately an inherently subjective judgement, and tends to change with the scientific *zeitgeist*.

2) Simplicity. This criterion corresponds to the reduction attribute, but is used here with an additional epistemological meaning. Following Occam's razor, it means that if two models describe a system with equal accuracy, the most simple one is to be preferred.

3)Explanatory power, i.e. how many of a system's observed behaviours are accounted by the model. This is equivalent to the pragmatical attribute, and in the case of astronomical models it has grown from celestial bodies' positions to include the fall of apples on the earth, and eventually the nature of space-time.

4)Prediction accuracy. This is a test used for scientific theories in general, and applies to models in the sense that if a model's predictions do not fit the observation data from the original, the isomorphism requirement is broken.

The first formal model of the solar system (actually of the whole universe, at the time), is due to Eudoxus of Cnidus, motivated by his teacher Plato. He described the universe as a series of transparent, concentric spheres, with the various objects attached to them. The spheres revolve within each other on axes pointing in various directions. Eudoxus' formal model is usually considered as the first one in astronomy, but its predictions were only approximations. Unfortunately, the details of this models were lost to history, and what survived was the improved version of Aristoteles.

The next breakthrough is due to Claudius Ptolemaeus. He refined the spheres model letting some of them revolve not around the earth, but around points in other spheres. The plausibility suffers, since no explanation is given why the spheres go trough each other without breaking. Moreover, despite its name, his system is not geocentric at all:

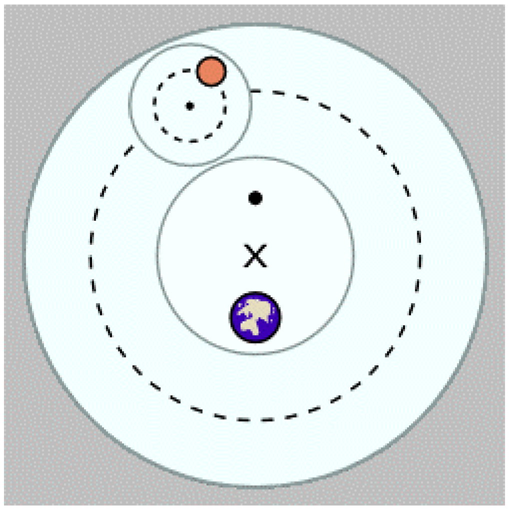


Fig. 2 The geocentric model. The big interrupted line is the main component of the orbit, called deferent. The small interrupted line is the epicycle, in this case a single one. The deferent revolves around the marked with an X, halfway between the earth and the equant (black dot). Taken from: http://en.wikipedia.org/wiki/Geocentric

Notice that the deferent is not centred on the earth as in Eudoxus' model, it is an eccentric orbit. Ptolemaeus admitted that making planets revolve around an empty, arbitrary point in space was implausible. He defended his model only on the grounds that it allows accurate predictions. In this case, prediction is more important than plausibility.

One very important aspect of the geocentric model, is that its accuracy can be improved by adding more epicycles. As the model becomes more accurate, its complexity grows until it becomes impractical to calculate. "Adding epicycles" has become the pejorative way to describe repairing a faulty model, it corresponds exactly to the problem of having too many free parameters, as discussed above.

Nicolaus Copernicus published his heliocentric model in 1543. However, Copernicus model was deemed by many of his contemporaries as implausible, since it required the assumption that the earth revolves around itself and around the sun. Moreover, though Copernicus created a new paradigm, his model was no better at fitting the data than Ptolomaeus. Its main effect was rather to unleash new research by other astronomers.

Galileo Galilei, in his book Dialogue Concerning the Two Chief World Systems, argued in favor of the Copernicus model (at least in public) because the calculations it requires for practical purposes are more simple, though geocentrism remains more accurate, if enough epicycles are used. In this case, simplicity is more important than plausibility or prediction accuracy.

It was Johannes Kepler who improved the predictive accuracy of the heliocentric model. Kepler tried for years to improve on Copernicus, using eccentric orbits. His model fit Tycho Brahe's observation data very well, with the exception of three observations of the planet Mars.

Kepler decided to trust Brahe's data, and disposed of simple assumptions about planetary orbits. Instead of circular orbits, he proposed elliptical ones. More "un-simple" and implausible yet, he proposed changing speeds for the planets. Kepler's model fitted the data better than the geocentric model, uses formulas easy to compute and has fewer parameters. In this case, prediction and simplicity are more important than plausibility, but the purposes of the model have been expanded. Now, the issue is not simply how to calculate planet positions, but what the solar system actually looks like, i.e., improve scientific understanding. More important from the methodological point of view, Kepler's model drastically reduced the number of free parameters.

Kepler's model rapidly became the accepted one, but it was Isaac Newton who reduced the number of assumptions to universal gravitation and the laws of motion. Newton's model was as accurate as Kepler's, and more simple. It has a single free parameter, the gravitational constant. Maybe more important however, was its extraordinary explanatory power, for it applied to apples as well as planets.

Newton's laws predict with absolute accuracy the orbits of two bodies, but has fundamental problems with more than two. This is known in physics as the "three body problem". Despite a number of exact solutions (notably Lagrange's points) there is no way to predict the positions of more than two bodies for arbitrary times in the future. The amount of feedback loops grows very fast as the number of bodies modelled increases, and makes the system behavior chaotic and ultimately unpredictable. The same problem is found in neural simulation.

After Uranus was discovered, small anomalies in its orbit were detected that contradicted Newton's laws. Urbain Le Verrier assumed that there must be another planet beyond Uranus, and he predicted the position of Neptune, which was soon confirmed by observation. This is considered as a triumph in astronomy's history. Less well known, is that Le Verrier also tried to explain by the same method another observation that contradicted Newton: Mercury's perihelion precession. But planet "Vulcan" could not be found. This illustrates that models are not abandoned despite isomorphism failures, unless a better model is available.

Mercury's anomaly remained the only challenge to Newton's model, and not until Einstein's general relativity theory could it be explained. However, relativity made such implausible assumptions, and was so complex to calculate, that most astronomers refused to adopt it. Einstein himself did not expected that his model would be validated in his lifetime. Only after a relativity's implausible prediction (that the apparent positions of stars would be affected by gravitation during a solar eclipse) was confirmed, the relativity model was accepted. This is a good example of the difference between explaining and predicting, when it comes to the subjective judgement of plausibility.

Beyond scientific understanding, practical models must be easy to use. This is the reason why NASA still uses Newton's model to calculate the vast majority of spacecraft trajectories.

The solar system is not easy to model, but at least it is clear what the original is, (positions of celestial objects) and what is irrelevant. For example, an astronomer does not need to know anything about the geology of the moon to calculate its orbit. In other systems, it is not clear in advance what features of the original are relevant, even when the dependant variable(s) are clear enough. Compare, for example, solar system models to the task of developing a computer simulation to predict climate change. There are more dependant variables, like temperature, precipitation, sea level, etc. But in this case, so many independent variables are known to influence weather, that the researcher is forced to choose which ones can be represented, and can only rely on a-priory plausibility to choose those. Additionally, trade-offs are forced between spatial and temporal resolution on one hand, and computer resources on the other. Finally, there is not enough accurate data from the past to validate the model by running it backwards, and of course no data at all from the future.

Neural simulations face a problem more similar to climate change than to the solar system. There are computer resources limits for how many neurons and synapses can be simulated. Thousands of facts about neural systems are known, and is not clear in advance which ones are the most relevant. And finally, only a very incomplete description of neural behavior is available for systems bigger than a sea-slug.

2.5 Basic features of neural behavior

Neural-based models of cognitive behavior must start by choosing which characteristics of the original should be represented. Given the many facts known about nervous systems, judging in advance which features are most relevant and should be implemented, is a very difficult task. As it will be shown later, this is probably one of the main problems in the design of ANNs. In the following, some of the most basic neural features will be shortly described, and these will define the original that this work's simulations represent. There are two reasons why these particular features have been chosen. First, they all are important for the input/output of neurons. And, second, they are well understood since decades, and are discussed in introductory biopsychology textbooks. The short description bellow is based on parts II and III of Kandel et al (2000).

2.5.1 Neural input/output

The most important form of input/output in neurons is the action potential, a discrete electrochemical pulse that varies very little in duration or amplitude. Because of its appearance in a record of electrical activity, it is also called "spike".

Human neurons have a resting potential of around -70 mV. When presynaptic excitatory synapses fire, neurotransmitter-gated ion channels open in the postsynaptic membrane. Once enough Na+ gets inside a neuron, the firing threshold (about -65 mV) is reached, and a series of voltage-gated ion channels open and close in a precise sequence in which is called an action potential or spike. First, channels open that let Na+ ions in, making the neuron more positive until a peak of around +50 mV is reached. At this point, Na channels close, K channels open and the neuron becomes more negative, overshooting the resting potential at around -75 mV. This process takes about a millisecond, and is followed by the **refractory period (ref. per.**). The first part of it, some 2-3 ms., is called the absolute ref. per.. No spike will be produced when the neuron is stimulated using an electrode during this period, even if the stimulation is considerably greater than the normal amount that would produce a spike otherwise. The following is called partial refractory period, during which electrode stimulation is greater than normal in order to produce a spike, until the neuron returns to its original resting potential around 10-20 ms afterwards (depending on neuron type and previous activity among other things).

The precise values of the resting potential, threshold, negative overshoot, duration of absolute and partial refractory periods, as well as ion concentrations, vary across the different animal species and neuron types within an organism, but the mechanism is essentially the same and is found also in many other cells like, e.g., heart cells. The research that described these processes was performed in the early 1950s using mainly electrodes, and has lead computer simulations to make a number of assumptions about which are the relevant features that should be represented in a model, and which ones not.

When probed with electrodes, one of the results is that the absolute ref. per. determines the maximum spike rate. Different values of the constant electrode stimulation below this level, result in frequency modulation (FM, like in radio). Strong values trigger spikes earlier in the partial ref. per. and lead to hi-frequency spiking. Weak stimulation produces spikes at later points in the ref. per., when the neuron is more sensitive, and produces a low-frequency spike rate. Individual neurons are "noisy", meaning that individual spikes are equally strong (no AM, amplitude modulation) but the stimulation required to produce a single spike is not constant, it varies from one spike to the next, sometimes by a considerable amount. This has led most researchers outside biology to consider the average frequency, and not the timing of individual spikes, to be the relevant feature to be accounted for.

Another basic input/output neural behavior is the so-called temporal summation. A neuron integrates its input by adding (maybe multiplying, see Koch, 1997) all presynaptic excitatory spikes multiplied by their respective synaptic weights, subtracting all presynaptic inhibitory spikes (again, multiplied by their respective synaptic weights) and finally triggering a spike when the integration reaches the threshold at the axon's hillock. Crucially, the electrical effect of incoming spikes lasts for some milliseconds before returning to the resting membrane potential. A given neuron that would fire if ten of its neighbours spike simultaneously, will also fire if only five neighbours spike twice each within a couple of milliseconds. A neuron integrates how many spikes it receives, and also how temporally close those spikes are. This means that neural integration is affected by a short-term internal state, and the output of neurons is not a fixed function of its input, it depends also on its recent history. This fact is one of the oldest known about neural integration, and its mechanism is well understood, perhaps that is the reason why it is so easily overlooked.

Another input/output behavior that changes depending on recent activity is the so-called long-term potentiation (LTP) in hippocampus' neurons, a process known to be crucial for memory. After a period of intense stimulation, some neurons remain much more active for minutes or even hours. A similar phenomenon is found on most cortical neurons. After above-average stimulation, neurons become more sensitive to input for a short period of some seconds. Again, the output of neurons depends on a combination of the input and the internal state.

2.5.2 The binding problem and neural synchronization

Gray et al. (1989) showed that synchronization of individual neurons plays a decisive role in an old cognitive problem. The binding problem poses the question of how perception is able to bind together different parts of an object, while keeping it separate from its background. The original paper is written in a language too technical to quote it here, instead an explanation of the discovery, written later by one of the authors will be used:

"How does the brain bind all these individual characteristics together into a single impression of a green grasshopper?

Twenty years ago Christoph von der Malsburg, a computer scientist and brain theorist, now at the Ruhr University in Bochum, Germany, suggested a solution. By synchronizing their activities, nerve cells could join into effectively cooperating units-- so-called assemblies. Subsequently, a number of research teams, among them the group at Wolf Singer's laboratory at the Max Planck Institute for Brain Research in Frankfurt, have demonstrated that this "ballet of neurons" in fact exists. Peter Koenig, Singer and one of us (Engel) carried out an especially decisive experiment at the end of the 1980s. We presented a cat with various targets to observe. When we showed it a *single object, neurons in its visual system responsible for analysing characteristics synchronized their activities in a pronounced way. When we gave the animal two separate objects to look at, however, the common rhythm broke down. The synchronization changed to a pattern of rapid oscillatory fluctuations at characteristic frequencies between 30 and 100 hertz, a region that brain researchers call the gamma band."* (Engel et al., 2006)

Although the study of neural synchronization has meanwhile become a major field of research in neuroscience, it has had little effect in cognitive simulation. Neural networks using binary output, and non-supervised hebbian learning rules, date as far back as the Boltzmann Machine (Hinton and Sejnowski, 1983). Yet it was the discovery of the role of neural synchronization in perception, that inspired a new type of ANN called Spiking Neural Networks, the type used in the present work. Though these networks have attained a modest expansion, they are mostly restricted to engineering applications like pattern recognition. This is not surprising since the original studies concerned sensor visual systems, and tracking synchronization in association cortex is much more difficult.

2.5.3 New research on spike timing

ANNs rarely represent neuron's spikes, and opt for a so-called "firing rate", i.e., an average of the neuron's firing rate over many spikes. This is odd, given that the binary and discrete nature of spikes is one of the oldest know input-output features of neurons. We will examine later the reasons why traditional ANN models have chosen not to represent spikes, but first let us examine some of the facts that have been discovered recently.

There is now plenty of literature in neuroscience indicating that information processing at the neuron level is quite sophisticated. A good overview article is Beardsly (1997). From one of its original sources, itself a widely quoted review article (Koch, 1997):

"Over the past decades, neural networks have provided the dominant framework for understanding how the brain implements the computations necessary for its survival. At the heart of these networks, are simplified and static models of nerve cells. But neuroscience is undergoing a revolution, and one consequence is that the picture of how neurons go about their business has altered." (p.207)

"Experimentally, the best-studied aspect of increasing connection strength is long-term potentiation, which can be conceptualized as an increase in synaptic weight lasting for days or even weeks. It is induced by simultaneous activity at the pre- and postsynaptic terminals (Fig. 1), in agreement with Hebb's rule. [...] These neural-network models assumed that, to cope with the apparent lack of reliability of single cells, the brain makes use of 'firing-rate' code. Here, only the average number of spikes within some suitable time window, say a fraction of a second, matters. The detailed pattern of spikes was thought to be largely irrelevant " (p. 207)

The author reviews other research, that has shown that the timing of individual spikes indeed carries about 10-40 per cent of the theoretical information maximum, only the rest being actual noise (p. 209). If this is the case, the coding used for the output of neurons is not Frequency Modulation (FM) as most neural networks assume, but rather Ultra Wide Band (UWB), where the timing of discrete pulses is used to carry information. Interestingly, UWB is a descendant of a forgotten technique used in the oldest radio transmissions, which used electrical sparks -not unlike neural spikes (the German word for radio comes from Funke, "spark").

Not only the timing is important, but also the *order* in which presynaptic spikes are received with respect to the postsynaptic ones. These facts contradict the view that individual spikes are an unreliable way to carry information, the main reason why they are not represented in ANNs (Koch, 1997 p.209). The author concludes:

"Only very little of this complexity is reflected in today's neural-network literature. Indeed, we sorely require theoretical tools that deal with signal and information processing in cascades of such hybrid, analog-digital computational elements. " (p. 210)

Taking together basic neural input/output, the role of synchronization in neural binding, and the way information is carried by individual spikes, the black box method suggests that neural networks require an implementation of the actual input/output of neurons, at least for cognitive simulation purposes. This implementation must be much more detailed than the ones currently used by BP-ANNs.

2.6 Neural models in cognitive science

Modern digital computers first became widely know to the general public at the early 1950's, most notably as a result of their use in the 1952 USA presidential election. The most widely used analogy to describe them at the time was "electronic brain". Perhaps, this comparison was the result of a weak form of the Turing test: if computers can do arithmetic and play chess, and humans can too, they must be similar in some way. This analogy is far from perfect, as John Von Neumann -one of the most important computer pioneers- pointed out in his last book (Von Neumann, 1958). The conceptual framework eventually adopted in cognitive sciences, was a more abstract view of nervous systems as "information processing" systems. This concept is still widely used in symbolic Artificial Intelligence and cognitive psychology.

The first attempt to formally analyse the similarity between brains and computers was performed by McCulloch and Pitts (1943), who used simplified neuron-like components to implement logic gates, the building blocks of computer hardware. They are almost always mentioned in texts about the development of cognition models. This is a misunderstanding, and many cognitive researchers fail to realize that while McCulloch and Pitts showed how to build computers out of neurons, they did not show how to build brains out of transistors.

The first real attempt to model cognition -at its most primitive level- was Rosenblatt's "perceptron" (1957), a simplified retina model constructed with electronic hardware. The perceptron did not use the serial processing (Von Neumann architecture) of digital computers, it copied the parallel structure of retinas and the brain's cortex, using cells and synapses. This first type of artificial neural network, became the subject of extensive research, mainly in the field of pattern recognition.

Eventually a book by Minsky and Papert (1969) proved that a perceptron cannot recognize some very simple patterns (e.g., the XOR problem). Their analysis took the form of a formal mathematical proof, this probably led most psychologists to view perceptrons as a dead-end paradigm, and research in this field came almost to a halt for many years. However, a mathematical proof does not necessarily has this ultimate meaning. It is true only within the constrains of the assumptions that underlay the proof. In Minsky and Papert's case, one crucial assumption was that neural networks have only two layers, one for input and one for output. Instead of simply describing a fundamental problem, they could have also have proposed the solution (three or more layers; even mammalian retinas have three layers). To their credit, this seems obvious only in retrospective, and it took the cognitive modelling community a long time to understand this.

In the 1980's three independent teams discovered the three-layer Back-Propagation Artificial Neural Networks, made popular by what may be the most influential book on cognitive modelling, Parallel Distributed Processing (1986) usually abbreviated as PDP. By the simple device of using three layers, BP-ANNs had abandoned Minsky and Papert's (1969) implicit assumption of only two layers, and could solve, among others, the XOR problem. This type of ANNs -and their variations- are still today the most commonly used, and are also widely used outside cognitive research. ANNs brought a paradigm shift in the original sense of Thomas Kuhn (1962), changing the basic assumptions about the way to develop cognitive models, reinterpreting the empirical facts, and promoting a wave of research in the subject.

Though still embedded in the information processing concept, the main difference of the PDP approach is to abandon the serial nature of standard computers. It uses instead structures and processes similar to the ones studied in neurobiology, like cells (neurons), weighted connections (synapses), activation functions (neural integration) and parallel processing of information.

PDP has delivered models that are biologically more plausible, but the analogy implicit in the name "neural network" is often not very close. The massive amount of facts known about neurons and nervous systems, and the vast body of research in cognition, have forced ANN models to make extreme simplifications.

It is difficult to put a date to the point when neural networks became an accepted way to model cognitive processes, but without doubt one of the early successes was the paper *From rote learning to system building* (Plunkett & Marchman; 1993). Using a BP-ANN with 18 input units, 30 hidden units and 20 output units, they achieved a very good fit to the data produced by children when learning the past tense of English verbs. The model not only fitted the data, but made a prediction about the evolution of error rates that was inconsistent with the theory of the day, and the prediction proved correct. The model was a good one when it comes to represent the distributed storage of cognitive contents in a distributed way, following the Hebb's rule. Strikingly, with less than hundred cells a neural network can model some cognitive processes that take 100 billion neurons in humans.

One would expect that such a success in modelling learning would be rapidly followed by models of much more basic behavior, like classical and operant conditioning, but this did not happend. A BP-ANN can represent the number of words recognized and error rates as a function of the number of loops of the learning algorithm, but it lacks a representation of timing and order of events. Both classical (i.e., like Pavlovs's dog) and operant learning (e.g., like a rat in a Skinner box), are very sensitive to the order of stimulus and behavior, and the time between them (latency). A BP-ANN can adequately represent the synaptic changes of long-term memory, but not the short-term effects of working memory. To put it in other words, when the dependent variable has a time component, BP-ANNs lack a mechanism to represent it due to the non-changing nature of the cell's input/output function.

A rarely mentioned attempt to model simple cognition, are Braintenberg's vehicles (1993). These are *gedankenexperiments*, simple robots using only a few computing elements like Grey Walter's robotic turtles (Walter, 1950). While Walter used analogue electronic valves, Breitenberg proposed "neurodes", hypothetical devices that produced spikes. Neurodes produce more spikes per second when a sensor input (e.g., photocell) is strong than when it is weak. The stepping motors of the vehicles move when more than one spike is received at the same time from the neurodes. The vehicles thus simulate neural integration based on semi-random spikes, and are capable of autonomous behavior and even simple forms of social behavior. While such simulations are relatively well known by robot researchers, stochastic-spiking systems are mostly ignored by the cognitive research community (see 2.7.1 and SIMULATIONS II-Measuring synchronization).

2.7 A conventional ANN

At this point, it is necessary to explain what an ANN is, and how it works, even before the reasons why they are designed this way are analysed. There are many different types of ANNs, and explaining even the most common ones would be beyond the scope of this work. The reader should be aware that for almost every feature in the following example, there is an ANN type that does not implement it. Fortunately, there is a family of networks that dominates research and makes up most of the models used in cognitive psychology.

The canonical three-layer Back-Propagation Artificial Neural Network (BP-ANN) has the following structure:

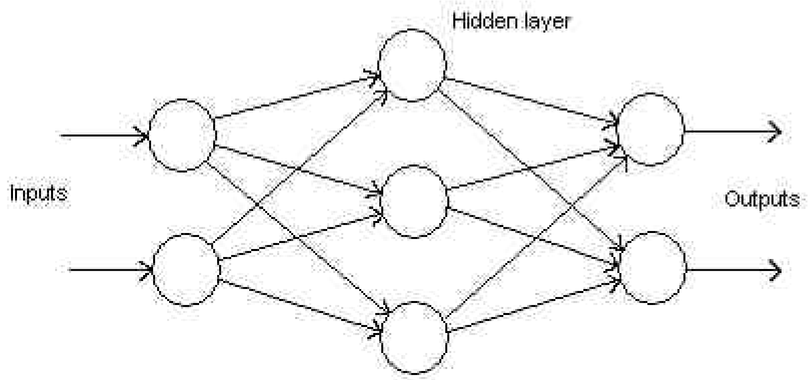


Fig. 3 Three layer feed-forward network. Taken from:
http://www.seattlerobotics.org/encoder/nov98/neural.html

At the left is the input layer, that receives activation from the outside world, e.g., a pattern to be recognized. The middle layer is called "hidden" since it has neither input nor output connections outside. There may be more than one hidden layer. Finally, to the right, the output layer expresses the output of the network in the form of the activation of its cells. The cells may have connections (synapses) within their own layer, or output connections with the next one. Information flows only from left to right, this is called "feed-forward". It lacks feedback connections (recurrent) between the layers.

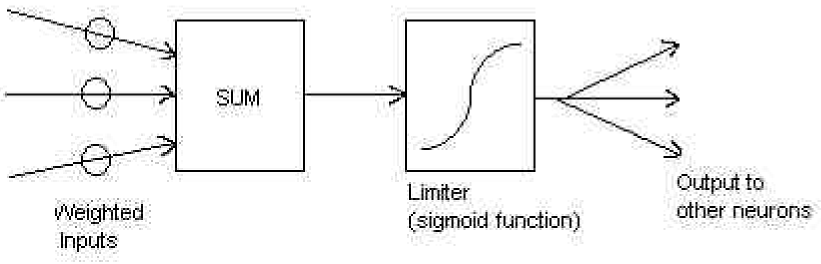


Fig. 4 Processing in a single cell. Taken from
http://www.seattlerobotics.org/encoder/nov98/neural.html

Each cell's input has a "weight", a number representing the functional effect of the connection, much like Hebb's rule (1949). The connections may have positive values (excitatory) or negative (inhibitory). They are added together, which represents neural integration. This value is then transformed using a sigmoid function, to simulate the non-linear behavior of neurons. Finally, the output is fed to the next cell(s).

The BP learn algorithm data consists of one set of stimulation patterns, and a corresponding set of expected output patterns. At start, the weights of the cells' connections are set to random values. The input and resulting cells' activation proceeds then from the input to the hidden to the output layer. The output layer's activity pattern is then compared with the expected pattern, and an error measure is computed out of their difference. This error signal is used to change a fraction of the weight of the hidden-to-output connections, in order to diminish the error the next time. The changed weights themselves, are used to compute the error on the hidden layer, and change accordingly the weights between the input and the hidden layers. In this way, the weights' changes propagate backwards through the network, giving its name to the learn algorithm, and the first loop (iteration) is complete.

The network is "trained" using one pair of input-output patterns after another, changing the weights slowly and mixing them. After the whole set of input-outputs has been run, it starts again, usually with a different order of the training input/output pairs. The order of the pairs is changed ***to eliminate sequence effects***. The effect of the previous pair on the present one, behaves as noise and averages to 0 -on the long term- when successive pairs do not share similarities. And if similarities exist, those should be learned. The whole set is run many times, until the mean error for the input-output pairs has reached a barely changing value.

Once the network has been trained, typically for some thousands of iterations, the weights are fixed. To test it, new, previously unused input-output pairs are presented, and if successful, the network delivers the expected output in what is called "generalization" i.e., learning.

It is important to note that the network does not simulate reaction times -the dependent variable of many cognitive experiments. Each test pattern is run only once -in three steps- from input to hidden to output layer. It is an instant result, that does not require iterations and takes exactly the same time -with no time variation whatsoever- for every input-output pair once the training phase is complete. The output shows likewise no quantitative variation at all for a given input, unless random noise is added.

The back-propagation learn algorithm is one example the "supervised" type, which are known to be biologically implausible. Each connection needs additional information to change its weight, that is not available to either the presynaptic nor the postsynaptic cell. In biological terms, this would require that each synapse has an additional synapse under the control of an *homunculus,* who tells the synapse when and how much it should change.

Another problem concerning learning, is the question if the neural network actually learns anything not already known to the program as a whole. It can be argued that what is to be learned is already stored in the training pairs, though not in a explicit form. The learning process can be viewed as simply storing this implicit knowledge in a distributed way, using the connection weights. The advantage of a BP-ANN when compared to e.g., an expert system (a type of symbolic Artificial Intelligence program) is that the developer does not need to make the knowledge available in the form of explicit if-then rules. Indeed, the developer does not even need to understand how the problem can be solved, as long as the training pairs can still be generated, and a small fraction of wrong training pairs can be tolerated by the algorithm.

2.7.1 Typical model's assumptions

Some of the assumptions about nervous systems on which BP-ANNs are grounded, were reasonable at the time when they were first created, before the role of spikes in neural processing was better understood. The assumptions have not yet been updated according to the results of neurobiological research. Some of these will be examined in the following.

McLeod et al. (1998), in their cognitive modelling textbook, explicitly name five assumptions on which connectionist (PDP) models are based:

"1) Neurons integrate information. [...]

2) Neurons pass information about the level of their input. [...] In the classical neuron, information about the level of stimulation of the sending neuron is coded by the rate at which it fires. [...]

3) Brain structure is layered. [...]

4) The influence of one neuron on another depends on the strengths of the connections between neurons. [...]

5) Learning is achieved by changing the strengths of connections between neurons." (pp. 11-15).

Assumption 1 is correct, but as shown in 2.5.1, temporal summation requires a short-term internal state, representing the recent input. The standard "integrate and fire" cells used in ANNs do not implement *neural* integration.

Assumption 2 is flawed, as research in neural synchronization and the information carrying ability of individual spikes has shown. Besides disregarding actual neuron's input/output, another reason may play a role in the decision to implement output in the form of frequency modulation. The fire rate expressed as a floating-point variable represents the average number of spikes of a number of neurons over a period of time. Given the assumption that neurons are noisy (more about this below), the period represented should be long enough for the noise to be averaged to a small proportion of the signal. How long the averaging period is, or how many neurons are being simulated by a cell, is not specified in the models. One is forced to speculate in this case, probably the period is between 100 to 200 milliseconds and 1 to 2 seconds, and the neurons simulated may be those of a cortical column (10000), or, to be conservative, at least 100. It follows that implementing spikes instead of fire rate, not only would make the model much more complex, it would also slow down the simulation by several orders of magnitude.

Assumption 3 is partly true, but misleading. There are some layered structures in the central nervous system that feed information forward only, like the retina. Mammalian cortex also shows a cytoarchitecture of six layers, and in general information flows from the primary sensory areas, to the secondary sensory areas, then the association areas, and finally to the motor areas. But information does not flows in the cortex only forward. There are an enormous amount of feedback connections, not only between cortex areas but also among the various cortex layers, basal nuclei, cerebellum, etc. Feed-forward connections are neither the only structure nor -probably- the most important connection feature of brains. The reason why feed-forward layered structures are standard in models, is because the back-propagation learning algorithm works only in feed-forward networks. This is an implementation issue in computer models, but lacks plausible anatomical grounds.

Assumptions 4 and 5 are well supported by neurobiological research.

The successor to the standard reference book in cognitive simulation, *Parallel Distributed Processing* (McClelland and Rumelhart, 1988), is arguably *Computational explorations in cognitive neuroscience* (O'Reilly and Munakata, 2000). The last one is the main reference book used for this work, and it provides an unusually detailed description of neural physiology. To illustrate the effects of assumptions made about the decision to represent individual spikes vs. mean spiking rate, and related issues, some quotes of the last source follow.

"A neuron integrates information from many different sources (input) into a single real-valued number that reflects how well this information matches what the neuron has become specialized to detect, and then sends an output that reflects the results of this evaluation. This is the well-known integrate-and-fire model of neural function." (O'Reilly and Munakata, 2000 pp. 23, 24)

Notice that "integrate-and-fire" refers to a 1-bit discrete output, the spike. However, the output has been previously described as a "real-valued number" i.e., the mean spiking rate. The authors appear oblivious to this contradiction.

"However, most of our models use a rate code approximation to discrete spiking. Here the output of the neuron is a continuous, real-valued number that reflects the instantaneous rate at which an otherwise equivalent spiking neuron would produce spikes. [...] The principal computational advantage of the rate code output is that it smoothes over the noise that is otherwise present in discrete spiking. [...] This kind of noise would tend to be averaged out with thousands of neurons, but not in the smaller-scale models that we often use." (O'Reilly and Munakata, 2000 p.42)

Here the authors interpret the varying timing of individual spikes simply as noise, not information. The use of a rate code is proposed to simulate the assumed noise-reduction effect of actual brains, which have many more neurons that the model can simulate in a reasonable time. As previously mentioned in 2.5.2 and 2.5.3, while neurons are indeed noisy, not *all* of the spike variation is noise. For a computer programmer is easier to add the noise with a random number generator if desired, because the randomness can be kept under control at the desired level. If the noise is due to intrinsic properties of the system, chaotic behavior may ensue, specially in systems with many feedback loops (this is the case on this work's simulations).

"Time averaging is also important when discrete spiking is used for having a window of temporal summation, where spiking inputs that arrive around the same time period will produce a larger excitatory effect than if those same inputs were distributed across time." (O'Reilly and Munakata, 2000 p.43)

Interestingly, the authors are aware that individual spiking plays an important role in a neuron's temporal summation, but fail to implement it. Again, they appear to dismiss this property as irrelevant.

O'Reilly and Munakata (2000 p. 47) claim that a rate code can simulate regularly spiking -simulated- neurons, and that rate code plus noise simulates noisy neurons. This is true only in the case that the irregular spiking of neurons is noise and not information. This may be impossible to prove, since one of the results of information theory (Shannon's information entropy) is that perfectly non-redundant information has exactly the same statistical properties as white noise. Besides, thermal noise is known to play a role in spontaneous spiking of input-less neurons, so the authors are making a plausible assumption here, one that recent research shows to be probably false after all.

The main problem with rate code, is that it accounts for the spiking frequency of neurons, but not for the ***phase***. Consider the following example: two groups of cells are firing at the same frequency, say, 10 ms wavelength. The first group fires at times 10, 20, 30, 40 ms. and so on. The second fires at times 5, 15, 25, 35 ms. and so on. Each group is synchronized by itself, but out of synchrony with respect to the other group by half a wavelength. The two groups' activity is may be unrelated and represent different cognitive contents, but for a rate code there is no measurable distinction at all.

One simulation showing the relevance of phase effects was performed by Raffone and Wolters (2001). They implemented an (artificial) spiking neural network that simulated the electrical effects of Potassium ions only. The network could maintain up to four different subgroups of cells synchronized within themselves, all having the same main frequency but four different phases. They published this result as a proposed mechanism to account for the fact that visual short-time memory can handle at most four chunks (a chunk is a unit of information in short time memory, itself composed of various details. For example, to keep a telephone number in short-time memory, one may "chunk" two digits into one unit). The simulated cells produced discrete output, i.e. spikes. Despite the stable frequency of all subgroups, individual cells had continuously changing wavelengths. If one of the cells spiked too late, it would be affected by the too-early spiking cells of the following subgroup of cells, and would "jump wave" backwards, i.e., synchronize with the following subgroup's phase. The same can also happen forwards. So the limiting factor for how many visual chunks can be maintained, probably depends on the frequency and the variation of cells' wavelengths around that frequency, which together determine how many different phases can coexist without interference. This result was a surprise for the researchers themselves (Wolters 2006, personal communication).

Finally, another critical assumption, is the lack of relevance of the ref. per.:

"This refractory period effectively results in a fixed maximum rate at which a neuron

can fire spikes (more on this later)." (O'Reilly and Munakata 2000, p. 29. the same point is also made on page 48, and on page 70).

The ref. per. is not described in detail, and no clear distinction is made between the absolute and relative phases of it. In the whole book it is mentioned only these three times, of which only the first appears in the subject index. For the simulation, it is only relevant as an upper saturation limit that justifies the use of a sigmoid (S-shaped) activation function. This is surprising, since the relative ref. per. is the cause of nonlinearity of the input-output behavior of neurons, which shows also an S-shaped (unsymmetrical) form. Nonlinearity is frequently used in hidden layers of artificial neural networks because of mathematical considerations (a series of linear equations can be reduced to a single one, nonlinear ones cannot), but rarely is the ref. per. mentioned at all.

One can only speculate about the reason why the ref. per. is not given as justification for sigmoid functions. Either it has been overlooked, or implementing the ref. per. would force simulating individual neurons, spikes, internal states, and a much higher time resolution of the simulation. This all requires of course more computer and programming resources, and make the models more complex.

To resume the role of neural assumptions in the design of neural networks, many of them are based on oversimplifications of neural input/output, and some of them are simply false. For a technical application this is not necessarily a problem. A BP-ANN may not represent the effects of input sequence, but this makes it inmune to the slow-down of response time in case of negative priming. A software engineer can hardly justify developing complex and slow neural networks, with the purpose of making the response time even slower after negative priming input.

In cognitive modelling however, the goal is not to deliver an optimal and easy solution to a given problem, but rather to faithfully replicate the way cognitive systems function, including the case when they fail to function in an optimal way. It follows that cognitive simulations should abandon the extremely simplified BP-ANNs, and start to include more neural features to produce a more realistic input/output of the cells.

2.8 BP-ANNs and models of cognitive conflict

Formal models are rare in cognitive psychology, and even more rare are series of different models of the same original, like it was the case with solar system models in astronomy. Fortunately, one specific series of models has produced a good example, though the differences between models are relatively small (within the series). The origins of this chain of models can be directly traced back to the ANN paradigm shift, as can be seen by the authors' names. The authors of the PDP book were McClelland and Rumelhart. The first model of the Stroop effect was realized by Cohen, Dunbar and McClelland (1990). It was followed by another model proposed for the same subject, (Cohen and Huston, 1994). Finally, a series of more comprehensive models covering in addition other cognitive conflict effects, were presented by Botvinick, Braver, Barch,Carter, and Cohen.(2001).

These models were incrementally improved, using a conservative approach. The authors take a model, and change it as little as necessary to improve performance or adapt it to represent other cognitive phenomena. Here, only the last models (Botvinick et al., 2001) will be examined, since they are the most advanced.

One of the cognitive effects modelled is the so-called Gratton Effect. This effect shows itself on the Eriksen task (Eriksen & Eriksen, 1974). In an often used version of this task, one central letter (H or S) is presented to the experimental subject, who presses one of two keys accordingly. The central target letter is shown surrounded by other letters, which may be the same (HHHHH or SSSSS in the compatible case, no conflict) or the other one (SSHSS or HHSHH, incompatible case, conflict).

Cognitive conflict, or more specifically the compatibility effect, is measured as the difference in reaction time between the compatible and incompatible cases. Gratton et al (1992) found that the cognitive conflict varied depending on whether the previous item had been compatible or not.

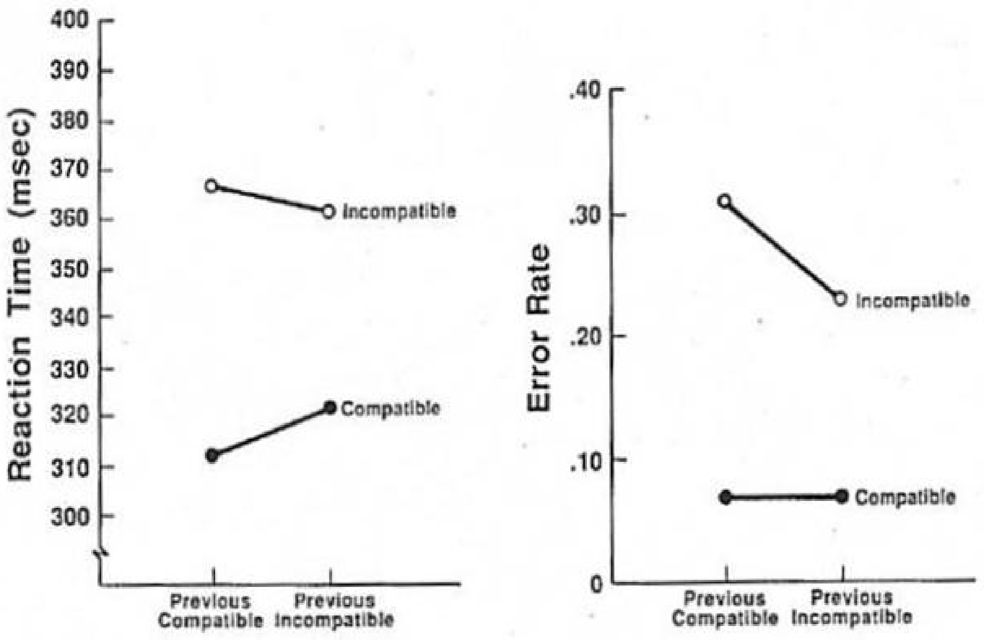


Fig. 5 from Gratton et al, 1992.

The original experiment was interpreted by the authors as evidence that cognitive control processes had adjusted the attention after each trial. According to this interpretation, after an incompatible trial, attention would be more focused in the target and mask out the flankers, after compatible trials the flankers would be taken into account more strongly -despite irrelevance- and speed up the answer for compatible item (or slow down the answer of incompatible ones). This cognitive control instance would then constantly monitor conflict, and adjust processing selectivity constantly. Earlier, in part due to functional magnetic resonance studies, it had been suggested that this control instance is anatomically located in the Anterior Cingulate Cortex.

New research (Wendt et al 2006, Hommel et al 2004) strongly suggests that this "control hypothesis" -a short-hand name for lack of an accepted one- is premature, and more simple memory effects are at least partly responsible for a great part of the Gratton Effect. To put it in blunt words, the apparent adjustment of attention may not be an effect at all, but an artifact produced by sequence effects. When the number of targets is increased from two to four, the Gratton effect is usually much smaller or disappears completely. No literature is available that directly compares two vs. four stimulus while eliminating from the analysis item repetitions, this would be a good opportunity to test the predictions of the model. However, lacking a representation of input sequences, the model will fail to show differences between two and four stimulus runs.

The theory that describes better the Gratton's effect is Bernhard Hommel's feature integration (2004). A short description of it, is that all features of a trial -relevant or not- are bound together in memory as a chunk. When the next item is identical, response is speeded-up. If the next item shares no features with the previous one, response time is unaffected (note that this is seldom the case; even if both target and irrelevant features change, the expected response may be the same). If only some of the features repeat, this recalls a chunk in memory, and because the given response is part of the chunk, it may slow-down the present response. Hommel's own diagram:

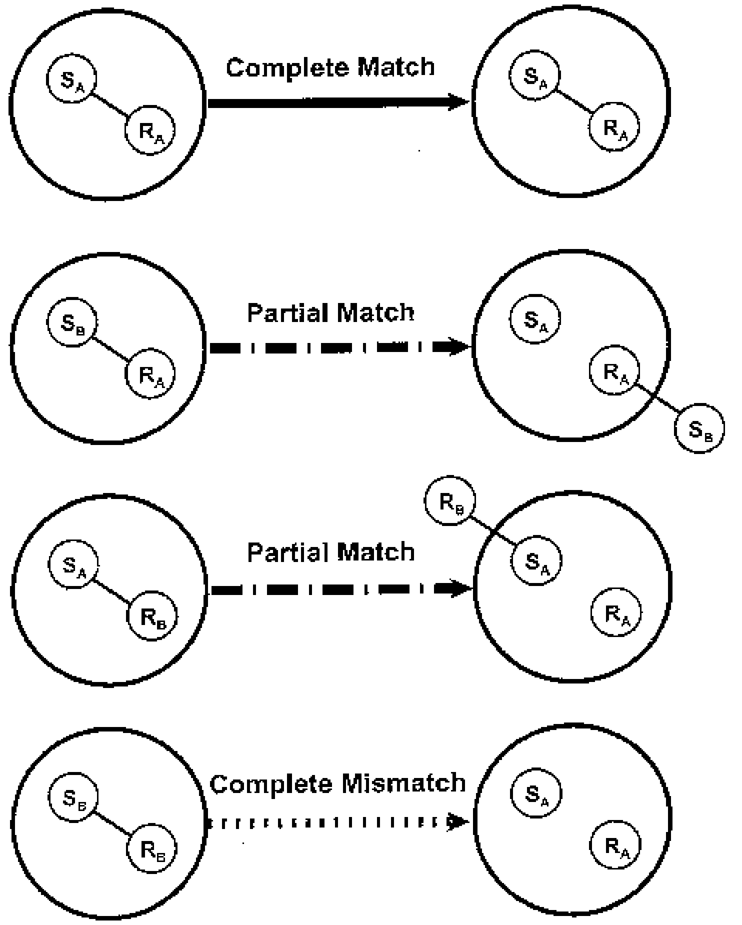


Fig. 2 The four possible combinations resulting from the repetition or alternation of binary stimulus and response features. In the example, the second part of the trial (*right column*) requires response R_A to stimulus S_A, following a stimulus-response pair with the same or different features (shown in *left column*). The critical assumption is that the codes of preceding stimulus (features) and response (features) are bound so that activating one code tends to activate the code it is integrated with. Note that (only) if there is a partial match does this lead to activation of wrong and misleading stimulus or response features

Fig. 6 The effect of trial to trial features' change or repetition. Taken from Hommel et al (2004).

Meanwhile, the control hypothesis has produced formal models not only of the Gratton effect, but also of the Stroop effect and others (Botvinick et al, 2001). Though these effects come from different experimental paradigms, in most of them reaction time is slowed down due to the presence of incompatible stimuli-irrelevant to the task, (or speed up in compatible case). The models used previously developed BP-ANNs, so no new learning was needed and the connections' weights remained fixed. The only change in the connection structure was to add a "conflict monitoring unit".

The first model to be consider, is the model for the Stroop task (1935). This is based in the Stroop test, where words for various colors are printed in an ink of a different color (e.g., "RED" written in blue ink). When subjects are asked to name aloud the *colors* in which the words are printed, they do so slower than when asked to name the colors of bars. The long training in reading seems to produce an interference with color-naming, this is the source of cognitive conflict in this case. Let's examine the model of this effect:

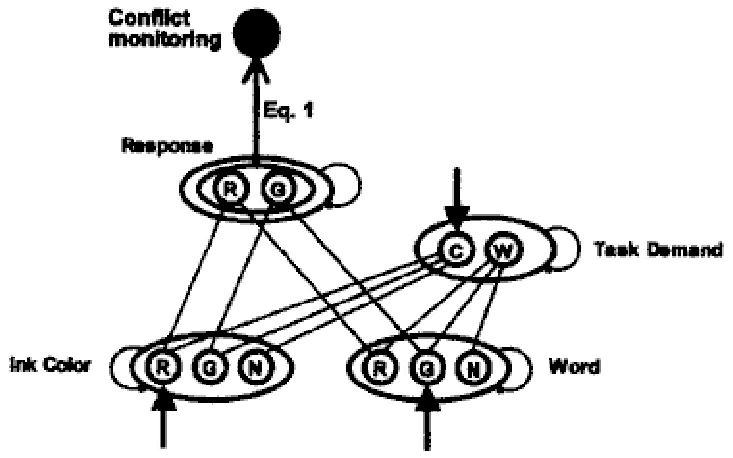


Fig 6. Structure of the Stroop task model R: red G: green N: neutral C: color naming W: word reading. The cells are circles, groups are ellipses, and the curves a the side of the ellipses -with a dot in the lower part- are lateral inhibition connections. Taken from: Botvinick et al., 2001, p. 631.

The network proposed by Botvinick et al. (2001) has a three-cell input for each task, a two-cell input to determine the task, a two cell output layer, no hidden layer, and a conflict-monitoring unit. The conflict is measured through the Hopfield energy of the response layer. This means in this case (RxG)+(GxR). Its value is zero if one or both cells are inactive, and maximum if both show maximum activity.

"The procedure used in simulating a trial is detailed in Cohen and Huston (1994). Briefly, one of the task units is activated during an initial priming interval, during which the output units are inhibited to prevent premature responses. The input pattern is then applied and the response-layer inhibition removed." (Botvinick et al, 2001 p. 631).

The procedure used to model human response times, is not described but simply quoted. In the Cohen and Huston (1994) model, the trials were not run in the usual way. Instead of computing only once the activation of the cells in a single pass trough the layers, the activation in the input layer was fed forward as a fraction, and the response units computed a running-average of their input. Gaussian-distributed noise was also added to the response units. After a number of iterations of this procedure, the difference in activation of the two response-layer units reaches a threshold, and a response is recorded. The number of iterations needed to reach the threshold is taken as equivalent to response time. The variability in the number of iterations is produced not by the network itself, but by adding noise from a pseudo-random number generator. This explains the somewhat cryptic caption under the results figure:

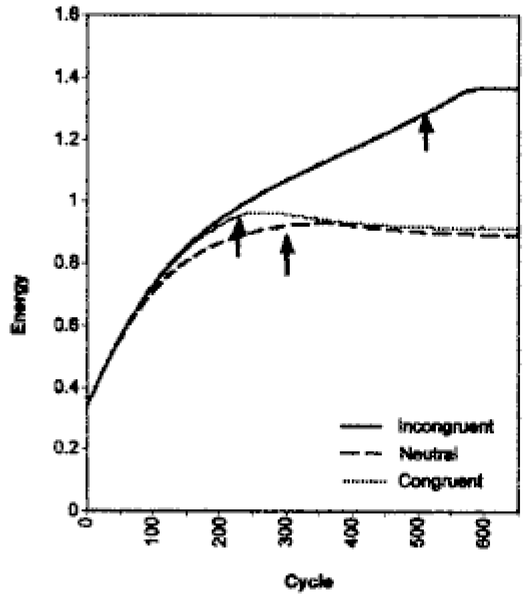


Fig. 7 *"Energy as measured in the response layer of the Cohen and Huston (1994) model during simulation of congruent, neutral, and incongruent trial in the Stroop task. Arrows indicate average response times. Energy was recorded for each cycle of processing, and the data shown are means based on 100 trials in each condition.".* Congruent is used as synonym of compatible. Taken from: Botvinick et al., 2001, p. 631.

The method used to simulate the "Gratton Effect" is very similar, with the important exception that the control monitoring unit is not a passive measurement device, but provides feedback to the network. Let us examine it:

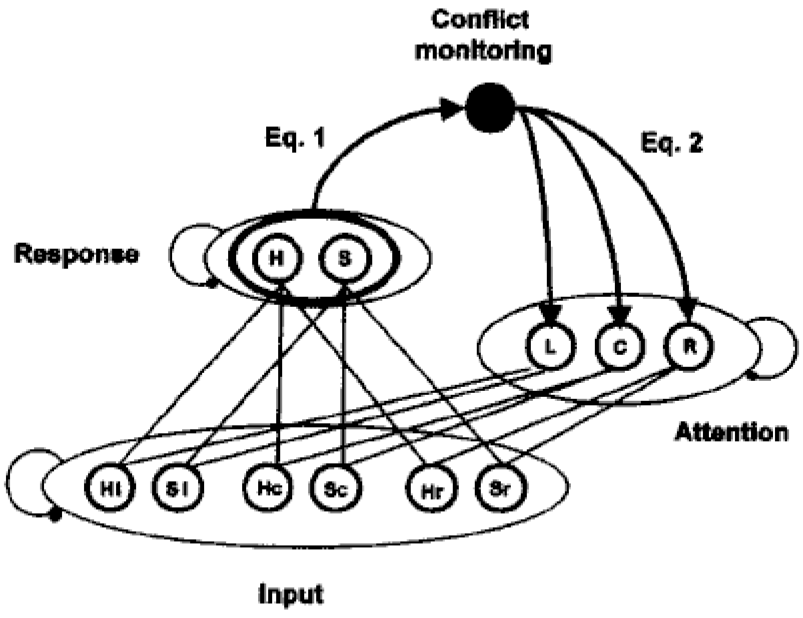


Figure 7. Structure of model used to simulate results of Gratton, Coles, and Donchin (1992). Eq. = equation; L = left; C = center; R = right; Hl = *H* left; Sl = *S* left; Hc = *H* center; Sc = *S* center; Hr = *H* right; Sr = *S* right.

Fig. 8 Taken from: Botvinick et al., 2001, p.640.

"... control is implemented as a spatial attention layer. Unlike the earlier simulation, the input to this layer is now adjusted from trial to trial on the basis of the output of conflict monitoring, so that high levels of conflict lead to a concentration of input to the center attention unit, and low levels of conflict lead to a more even distribution of input to the attention layer" (Botvinick et al., 2001, p.639)

The simulation produces results virtually identical to experiments with humans (Gratton et al 1992, experiment 1). The critical words here are "adjusted from trial to trial" and are easy to overlook. It means that since on a given trial units have no memory of the previous one, some device must transport this information. This is done by storing a fraction of the activation value of the conflict monitoring unit, and using it in the *next* trial as additional input for the attention units. Another crucial detail, is that the activity of the control monitoring unit is not simply given as input to the attention units, for this would produce no differentiated attention to the central as compared to the lateral letters. The simulation program computes each trial a different distribution of input for the attention units, depending on the measured conflict. The higher the conflict, the higher the input of the central attention unit, as compared to the left and right attention units. In other words, both the conflict and its effect on attention are managed by an *homunculus* in the program, not by any feature of the neural network itself.

It would have been interesting to run simulations without a conflict monitoring unit, and storing the activation value of the input and/or response units, but this was never tried (Botvinick, personal communication, 2007). No theoretical reason is given why only the conflict monitoring unit should have a short-term memory (an internal state) but not the others. If all cells had been treated equally, similar results might have been produced, and the conflict monitoring could be eliminated from the model. It seems that though the dependent variables to be fit are response times, the real interest of Botvinick et. al. is Anterior Cingulate Cortex activity.

Similarly, for the Stroop model, another effect could have been tested. In so-called negative priming, the stimulus feature that should be ignored in item n-1 becomes the target in item n. This sequence effect increases response times in humans, but the model could only account for it if the cells had a short-term memory.

2.8.1 Validity of BP-ANN models of cognitive conflict

The cognitive models described above dominate the field of cognitive control modelling, and influence research in other fields. Jonathan D. Cohen, one of the authors of the models above, writes in his web page:

"Theoretical work. Neural network models are developed as a way of articulating precise hypotheses [...]this work has led to novel hypotheses about system level function, such as the response of anterior cinglulate[sic] cortex to conflict in processing and its influence on adaptive changes in cognitive control. This work has also predicted, and led to the discovery of new anatomic relationships, such as projections from the anterior cingulate cortex to locus coeruleus." (Cohen, 2008)

It is important that the models are proposed to explain the data from human behavior experiments, where reaction times are the dependent variables. The "predictions" mentioned in the previous quote do not refer to as-yet undiscovered behavior effects, but rather to neuroanatomical or neurophysiological issues.

To make a preliminary criticism, a review of some points follows:

1) BP-ANNs lack any mechanism equivalent to short-time memory, no trial-sequence effects may be modelled with them without modification, and in this case the modifications were performed on the simulation program, not in the neural network. This modifications applied to a single cell, or in a single set of connections, instead of becoming a standard feature of all cells.

2) The models are ex-post-facto, they fit known data but do not actually predict.

3) Reaction time is basically constant. Reaction times simulated through fractional output, running averages and iterations reflect only the weights of the connections; they are an unnecessary complication.

4) Differences in reaction time do not reflect a chaotic or stochastic nature of the

networks, they are simply artificially added.

4) We have a model of the "Gratton Effect" that drives research over the anterior cingulate cortex, yet the effect may be an artifact and the simulations did not try simpler methods, like developing a short-time memory (more about this in the following).

More fundamentally, there is no model of cognitive conflict, but model**s**, in plural. Different connection structures, weights and special devices, e.g., response time measurement, are used depending on the task being simulated. Ideally, since BP-ANNs are capable of learning, a single network should be able to learn and perform all the tasks. It can be argued that it is possible to do so, but the model would have to be overly complex and it would be difficult to clearly understand how it works. On the other hand, lack of a general model makes incremental improvement of theory difficult or impossible.

To make a comparison with astronomical models, suppose that Kepler had written "my model predicts perfectly the position of Mars using an elliptic orbit, and the position of Venus using an eccentric circular orbit, but it is inaccurate for all other planets".

It must be noted, that although the many-models situation is unsatisfactory, modellers take great pains to make the differences as small as possible, e.g. adding a single cell to a previous model. Indeed, one of the difficulties trying to understand a given model, is that it has frequently been used in a chain of simulations for different purposes, and details of its implementation are buried in a quote from a quote from a quote.

A major problem is the presence of many free parameters. The first model on the chain of models examined here, is used by Roberts and Pashler (2000, p.358) as an example of this problem:

"1. Cohen, Dunbar and McCLelland (1990) proposed a parallel distributed processing model to explain the Stroop effect and related data, The model was meant to embody a "continuous" view of automaticity, in contrast to an "all-or-none" view (Cohen et al., 1990, p. 332). The model contained many adjustable parameters, including number of units per module, ratio of training frequency, learning rate, maximum response time, initial input weights, indirect pathway strengths, cascade rate, noise, magnitude of attentional influence (two parameters), and response-mechanism parameters (three). The model was fit to six data sets. Some parameters (e.g., number of units per module) were separately adjusted for each data set; other parameters were adjusted on the basis one data set and were held constant for the rest. The function relating cycle time (model) to average reaction time was always linear, but its slope and intercept varied from one data set to the next. [...] they did not try to fit a model based on the all-or-none view"

In the later models of Botvinick et al., (2001) the authors seem to be aware of this problem and claim: *"Leaving the models' original parameters intact and using the same simple computation to determine conflict across all three studies reduces the number of free parameters associated with our simulation to zero"* (Botvinick et al., 2001, p.630).

This is obviously not true. Not only has the original free parameters problem remained unsolved; the final models have even *more* free parameters, due to the addition of a procedure to measure conflict in the output units, the conflict monitoring unit plus its connections and weights, and the fraction of activation preserved for the next item (but not used for any other unit).

Some of these problems have been noticed in the research generated by the previously discussed models (Botvinick et al., (2001). One of the proposed solutions (Verguts and Notebaert, 2008) is running the learning algorithm during the trials, using the conflict measurement instead of a normal (training-pair generated) error signal. This proposal is equivalent to giving the cells a short time memory, implemented as small variations from trial to trial in the connections' weights. However, this also would extend the known biological implausibility of the back-propagation algorithm to the cells' internal states.

A more simple and plausible solution, would be to give every cell an additional variable, that stores a running average of the input. While Botvinick et al. (2001) used such a device to represent reaction time, instead of resetting the value to zero on every trial, the running average could be computed continuously, and represent instead the internal state of each cell. In any case, some modification of the neural network models is required to implement the changing input/output behavior implicit in the Gratton effect and negative priming. This implementation should be part of the network itself, not an external device under the control of the simulation program. Otherwise, it is questionable to claim that the model as a whole is a neural network.

Despite what may seem to be a hard criticism of these models, BP-ANNs represent an improvement in modelling, for various reasons. First, they are formal and their results can be compared with empirical data. While the comparison may expose an inaccurate or little useful model, the alternative are non-formal models, where it is difficult to make a comparison at all. Informal models tend to be difficult to disprove because of their very ambiguity, some are even *unfalsifiable* in the sense of Karl Popper. Second, including neural features increases the isomorphism, and makes the models more plausible. Third, the PDP paradigm is relatively new, maybe the models should be judged as work of pioneers. It is unfair to criticize the use of BP-ANNs without mentioning that there are still no better alternatives.

To resume, the problem is not that BP-ANNs are a step in the wrong direction, it is that they do not go far enough when trying to use neural features to model cognition. So to speak, the bridge between cognition and neurobiology has still to reach the river's banks.

2.9 The reverse-engineering method

Up to this point, we have discussed the concept of model, some attributes of formal models, the criteria used to evaluate a model's usefulness and how this all applies to examples of neural network models in the field of cognitive psychology. For the analysis of both the original and the models, the black box method has been used. In the following, a method that expands this approach will be presented.

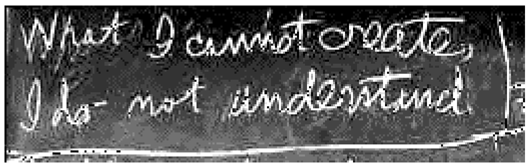


Fig. 9 "*What I cannot create, I do not understand*" Richard Feynman's last blackboard (undated). Taken form URL: http://duartes.org/gustavo/blog/post/2008/02/20/Richard-Feynman-Challenger-Disaster-Software-Engineering.aspx

The reverse engineering method is used in computer science and engineering in different ways, usually not to create simplified models but rather identical copies of the functions of complex systems, like an operating system or an aircraft.

The general idea, is that in order to understand a very complex system, one has to start by making a copy as accurate as possible of it. If the black-box method has produced boxes whose behavior is still too complex to reproduce in an efficient way, there is no choice but to apply the input-output analysis to smaller boxes, and to reproduce the internal workings of the subsystems. In the process of copying, insight might be gained. In the worst case, once the copy is complete -i.e., behavior of model and original are equal- one may try to understand the system by making changes in the components, and observing how its behavior is changed.

Reverse-engineering has been adopted by some leading psychologists (Pinker, 2008). Another example, closer related to this work, is the research project Blue Brain (Markram, 2006). Its stated goal is: "*The Blue Brain project is the first comprehensive attempt to reverse-engineer the mammalian brain, in order to understand brain function and dysfunction through detailed simulations*." (Blue Brain web page, URL: http://bluebrain.epfl.ch/) To this date, the project is capable of simulating a complete cortical column of a mouse, i.e., 10000 neurons including each local synapse, and the detailed neurophysiology of neurons down to the molecular level, including ion channels. Of course, such an ambitious goal can only be realized with the help of a large team of scientists, and using a parallel supercomputer donated by IBM.

In this work, the black-box method is combined with reverse engineering to reach a more detailed simulation than BP-ANNs do, in some cases even representing ions. Due to resource constrains, the size of the models is restricted to hypothetical systems of a maximum 100 cells. To give an idea of this constrain, the smallest multicellular model animal used in biology, the 1 mm. nematode worm Caenorhabditis elegans, has 302 neurons. Reverse engineering is used based on the assumption that neurons are too complex to be understood without the possibility of manipulating their subsystems one by one, and observing the effect on the whole system's input/output. An additional constrain dictated by reverse engineering is to avoid -whenever possible- abstract devices (e.g., BP-error measurement) without known counterpart in neurons.

3 METHOD

3.1 Description of the BCK neural simulation program

The version of the simulation software used as basis for the first series of simulations, was slightly modified from a program originally written as neural simulation construction set. This version is provided in the CD-ROM and described here, exclusively to allow running models developed later, and should not -by itself- be considered part of this work (after extensive modifications, the resulting program is called ION and was used for the next series of simulations). The program was designed to provide as much flexibility as possible to develop cognitive models, without having to learn computer programming. By itself, the program alone does not represent a specific model, it is a tool used to build models. Similar examples of the construction set design, are SNNS (Stuttgart Neural Network Simulator) -in the field of ANNs- and Mirek's Cellebration in the field of Cellular Automata.

Many of the basic input-output behavior of real neurons were implemented, as well as metabolic processes and synaptic weight modification. One of the main expected uses was to model learning at the cellular or synaptic level. Since learning is not a subject here, some parts of the software will not be described in the following text (see Appendix A).

The program was a modified Cellular Automaton. It would be classified now as a Spiking Neural Network.

"Networks of spiking neurons are more powerful than their non-spiking predecessors as they can encode temporal information in their signals, but therefore [they] do also need [a] different and biologically more plausible [learning] rule." (Vreeken, 2002)

The program has been for years in continuous development, the version described here is named BCK08.EXE , and reached this state on Apr 20, 2001. The program has been modified and recompiled to produce the graphics of this work, so the dates of the various files do not reflect the original ones.

3.1.1 Cells and connections

The BCK08 program simulates the behavior of a maximum of 100 neurons, each with a maximum of 40 synapses (connections). The main difference with most ANNs, is that each cell has a variable called "State", with 13 possible values. The output is binary, state 0 represents a spike, states 1 through 12 produce no output. This is the most important variable used to manage a cell's internal state. Besides producing spikes, State is used as an index to choose user-defined parameters that modify the input/output behavior of the cell.

The communication between cells is realized by a maximum of six simulated neurotransmitters (NT). More precisely, NTs represent the effect of neurotransmitters docking on neuroreceptors, and the resulting change in the postsynaptic concentration of ions. Each NT has a "power", a real number that represents how much it will influence the neural integration. Excitatory NTs have a positive-valued power, inhibitory ones a negative value. Independent of the intrinsic NT power, the amounts crossing the synapse are controlled by a synaptic weight, as it does in conventional neural networks. On a pre-synaptic spike, the NTs are added to storage variables, both locally for each individual synapse and globally for the cell as a whole. The NT stores represent the cell's electric/osmotic internal state and can be used to implement temporal summation (see metabolism parameter HL_PRI, 3.1.3). The individual synaptic stores allow to differentiate them according to their recent activity (e.g., for learning). The global stores represent the electric state of the axon's hillock, where the neural integration is performed.

Another important feature is the cell's energy management. One substance, called ATP (in analogy to adenosine triphosphate, the neuron's energy-transport molecule) is stored in the variable NT[0] and represents the energy available to the cell to produce spikes. The other substance is called LAC (in analogy to lactic acid, which causes fatigue in muscles) and stored in variable NT[1]. These variables do not represent neurotransmitters despite their name, while the others (NT[2] to NT[7]) do. Though probably confusing for the reader, this is done to simplify the internal data structures of the program, and improves the simulation speed. The parameters used to produce, destroy, and measure the effect on neural integration and other functions, are used in exactly the same way for all NT-variables.

To avoid confusion, it is important to note that **although the variable State is the most important variable to *manage* the internal state, it does not *represents* the internal state**. The mechanism that transports information over previous events to the present, is the storage and decay of NTs, which perform a role similar to a running average of the input. The variable State is the cell's clock. Only when other variables have different values for a given time of the clock (like varying cell's sensitivity to input during the refractory period), is the input/output modified.

Users of the program define their own model with the help of two text files. The file struc.txt defines the connection structure (topology) of the cells' network, i.e. the synapses. The file param.txt is used to set a series of parameters that represent the cells' metabolism and input/output behavior, these parameters may be changed also while the simulation runs, and stored whenever the changes seem promising.

The program also implements many other features of no relevance for the purposes of this work, and are only shortly described here. The neural network represents the brain of a "frog", who lives in a chessboard-like "world". The program implements sensor systems, like a retina and a "compass" sense. Depending on the position and orientation of the frog in its world, input is delivered to the sensor areas. Likewise, through the activity of the "motor areas" of the network, the frog is capable of three behaviors, turning left, turning right, and jumping to the square in its front. Also implemented, is automatic reinforcement or punishment, which are delivered as input to the "pleasure center" and the "pain center" of the network. Other features are implemented to allow users to develop a NT-based learning algorithm. The general idea, is that if the algorithm is successful, the frog will avoid punishment and receive reinforcements; if unsuccessful, the frog will simply perform what mathematicians call a random walk.

A tutorial and the program itself can be found in the attached CD-ROM. In order not to overburden the reader, a brief description of the relevant features follows, it should be enough to understand the specific models afterwards.

A single cell is defined by the following set of variables:

```
//cell's components
int      *Type,               //cell's default neurotransmitter
         *State,              //cell's "clock" (0==depolarization)
         *Axon,               //delay in transmission ??
         *LastTir,            //last wavelength
         *SynNum;             //number of synapses declared
float    *AuxInp,             //"electrode" for external stimulation
         (*NtInpP) [MNT],     //total store of primary NTs
         (*NtInpS) [MNT],     //idem secondary
         (*SynEq)  [MNT];     //equalizes number of syn. by NT
int      (*SynType)[MAX_SYN]; //types of declared synapses
int     *(*Syn)    [MAX_SYN]; //points to neighbor's State or Axon
float    (*SynWei) [MAX_SYN], //weights of synapses
         (*SyNtP)  [MAX_SYN], //amount of primary NT
         (*SyNtS)  [MAX_SYN]; //amount of secondary NT
```

(int means integer number, float is a real number.
Taken from the source-code file BCKMAIN.CPP)

`Type`: the specific NT send to the postsynaptic neighbors.

`State`: the most important variable of the cell. State 0 represents a spike, and is incremented by one afterwards on each following time step, until reaching a maximum value of 12. Whenever the integrated input reaches a threshold, the cell is reset to state 0. This variable is used as index in 1-dimensional arrays (vectors) and 2-dimensional arrays (tables) of parameters. This can be used to implement e.g., the refractory period by changing the effective threshold of the cell.

`Axon`: a simple counter variable to implement axonal delay depending on the cell's Type.

`LastTir`: whenever cells spike, the last state reached is stored in this variable. It so represents the wavelength between the last two spikes. It is used to color-code the graphic display, and plays an important role when evaluating synchronicity of cell groups.

`Synnum`: how many synapses the cell has, maximum is 40. These are input synapses, the maximum of output synapses is 100-1.

`AuxInp`: input patterns stimulate cells through this variable, it can be described as an electrode or input from sensor cells.

`NtInpP`: a one-dimensional array of values, stores the cell-wide amount of each NT. It is used to integrate all excitatory and inhibitory inputs, a biological analogy would be the axon's hillock.

`NtInpS`: a similar array, stores the "secondary NTs" which are no real NTs, but rather postsynaptic ions or secondary messenger molecules. Each time step, a given proportion of the normal, "primary" NT is transformed in secondary NT. This is used to represent a complex and only partly understood chain of chemical processes in the postsynaptic membrane. The purpose is to provide a chemical, longer-time indicator of the cells' previous activity history like it has been proposed, e.g., for calcium ions and secondary transmitters. It is irrelevant for this work, and will be not further mentioned.

`SynEq`: a normalization parameter. If specified in the structure file of a given model, the amount of each NT is divided by the number of synapses defined for that NT. This allows to change the number of synapses without having to adjust the threshold of the cell.

Each synapse is defined by the following variable arrays:

`SynType`: the presynaptic neurotransmitter type, usually. The user may define an arbitrary type if desired. This mechanism allows to declare all cells of the same type but still use more than one type of NT. It can be used, for example, to implement lateral inhibition. It can also represent the different effect of various neuroreceptor ion-channels that are triggered by the same NT.

`Syn`: this variable defines which cell is the presynaptic neighbor, and determines the connection structure of the network. It is implemented as a pointer to the neighbor's axon, and guarantees a constant addressing time (important if ported to parallel hardware).

`SynWei`: the effect of a synapse varies not only depending on the NT/neuroreceptor type, but also on its size and the density of NT-gated ion channels. Following a convention in neural networks, it is called the "synaptic weight".

`SyNtP`: the amount of received neurotransmitter/ions for an individual synapse.

`SyNtS`: the same for the "secondary" NT.

3.1.2 Connection structure

The default human-readable file used to define the connection structure is named "STRUC.TXT" and is found in each model's directory. Some of the commands have no theoretical relevance, including, e.g., which colors are used to represent neurotransmitters or the world. This file is loaded only once at the start of the simulation, the structure cannot be changed while the simulation runs. Explained bellow, are the most important commands only.

```
>Type---------   0  99  2 //excitatory 70
```

all cells are declared of having NT-type 2

who-begin who-end how-many type whom-begin whom-end
```
loc_nei------   0  99  2 -2 -1  1  2
```

A regular neighborhood, as a normal cellular automaton would use. In this case, all cells have four neighbors of type 2, the pairs immediately adjacent to the cell at both sides. This command binds the line of cells in a circle, so cell number 1 would have as neighbors 99, 0, 2 and 3.

```
>rnd_nei------   0  99  8  2    0 99
>rnd_nei------   0  99  8  3    0 99
>rnd_nei------   0  99  8  4    0 99
>rnd_nei------   0  99  8  5    0 99
>rnd_nei------   0  99  8  6    0 99
```

This commands produce a series of random neighborhoods. All cells choose 8 neighbors of the types 2, 3, 4, 5 and 6 each, among all cells. This is done to use different NTs with the same stochastic topology. By setting on run time the Power of one NT to a desired value and all others to 0, one may compare different NTs without having to load a new structure file.

```
>rnd_nei------   10 19  4  2    0  9
```
This command (only an example, not actually used) would produce a one-way connection, with cells 0 thru 9 each sending four outputs of type 2, to cells numbered 10 to 19. In this way layers may be defined if desired.

```
>World--------
....................
....................
..*-***-***-***-**..
..+++-+++-+++-+++-..
..-***-***-***-***..
..++-+++-+++-+++-+..
..*-***-***-***-**..
..+++-+++-+++-+++-..
..-***-***-***-***..
..++-+++-F++-+++-+..
..*-***-***-***-**..
..+++-+++-+++-+++-..
..-***-***-***-***..
..++-+++-+++-+++-+..
..*-***-***-***-**..
..+++-+++-+++-+++-..
..-***-***-***-***..
..++-+++-+++-+++-+..
....................
....................
```

The world, in which the "Frog" (marked as F) lives. The signs '.', '-', '+' and '*' represent sensor input: none, red, green and blue. Originally this was implemented to test learning in a maze, but it can be used to apply stimulation to the input zones of network.

3.1.3 Metabolism parameters

A parameters that define the "metabolism" are contained in a human-readable file whose default name is "PARAM.TXT", found like before in each simulation's directory. It is read at the simulation start, the parameters can be changed interactively, and stored in a new file if desired. Only the parameters relevant for the present work are explained bellow.

The program simulates a maximum of eight NeuroTransmitter (NT) substances which may have different properties. They are named NT[0] to NT[7]. There are two exceptions to this, the first is NT[0] which is not delivered through an axon when the presynaptic neighbor spikes. It represents instead the amount of energy produced by the cell.

```
>ATP----------
 0.0120
```
Each time step, the cell increases its amount of NT[0] according to the value of ATP.

```
>LAC----------
 0.1000
```
Every time a cell produces a spike, a part of the NT[0] is transformed in NT[1]. This is the second substance that is not delivered thru synapses, it can be used to represent "fatigue". The amount transformed is determined through the value of LAC.

```
>STI----------
 0.0000
```
The force of electrode-like stimulation applied to the input cells. If its value is 0, the networkis effectively isolated of the outside world.

```
>HL_PRI-------
  999    12    12    12    12    12    12    12
```
This is a very important variable array. It defines the half-life time of the energy substance, fatigue substance, and NTs 2 to 7. Real NTs increase fast their effect on the postsynaptic cell due to diffusion, and decrease slowly afterward as e.g., enzymes break them and the rests are recycled. To simulate this, a fraction is subtracted on each time step. The HL_PRI variables define how many time steps are needed to left half of the original amount, in a way similar to the half-life of radioactive elements.

```
>NT_TH--------
 0.0000  0.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000
```
Threshold to fire a spike, depending on the cell's type. Since NT[0] and NT[1] are reserved for energy management, no cells can be declared of this type, and need no threshold. Setting a value of 0.0 would make cells of a given type spike continuously, and signal an error in the structure.

```
>NT_PO--------
  1.0000  0.0000  0.4000  0.0000 -0.4000  0.0000  0.0000  0.5000
```
The power, or effect of each NT. Inhibitory NTs are assigned negative values, the same may apply to the fatigue substance NT[1].

```
>HL_AUX-------
    1
```
Half-life of the stimulation applied to the sensor input cells

```
>MASTER_DELAY-
    1
>DELAY--------
    1     1     1     1     1     1     1     1
```
The delay between the spike production and its reception by the post-synaptic neighbor is a minimum of one time-step. It can be declared for all cell types (master) or separately for each type.

```
>REF_PER_CELL-
 0.00 0.10 0.20 0.40 0.50 0.60 0.70 0.75 0.80 0.85 0.90 0.95 1.00
```
This array contains values indexed by the cell's state. The amount of each NT cell-wide store is first multiplied with their corresponding power, added, and finally multiplied

with the value of this array. In this case shown, the values increase in a S-shaped curve similar to the logistic function used in BP_ANNs. The middle part (0.40, 0.50, 0.60, 0.70) is linear, however. This illustrates one of the main advantages of using arrays of predefined values instead of a equation. Not only would it be very difficult to find a non-linear function with sigmoid ends and a flat middle part, computing it would be orders of magnitude slower that using a look-up array. This form of ref. per. was originally part of the simulation program itself, in newer versions is user-defined and is found in the PARAM.TXT file.

```
>REF_PER------
 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00
 1.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
 0.00 0.00 0.10 0.20 0.30 0.40 0.50 0.60 0.70 0.80 0.90 1.00 1.00
 1.00 1.00 0.90 0.80 0.70 0.60 0.50 0.40 0.30 0.20 0.10 0.00 0.00
 0.50 0.50 0.50 0.50 0.50 0.50 0.50 0.50 0.50 0.50 0.50 0.50 0.50
 1.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
 0.00 0.00 0.10 0.20 0.40 0.80 1.00 0.80 0.40 0.20 0.10 0.00 0.00
 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
```

This table shows some of the most important variables to explore the ref. per.. In the X-axis, it is indexed from 0 to 12 by the cell's own State variable. In the Y-axis, with 0 at the top and 7 at the bottom, the NT type is specified. The cross points define the fraction of a given NT, at a given State, that the cell will absorb from the input. In this example, the top line has a constant value of 1.0, so the energy production (NT[0]) is independent of the state. In the second line, fatigue substance (NT[1])is only produced during a spike. The third line, shows how the absorption of NT[2] increases during time. The fourth line shows decreasing absorption for NT[3]. The next line shows constant absorption. The sixth line, shows how NT[5] is only absorbed when both the cell and the presynaptic axon are spiking together. Notice the difference with the second line, fatigue substance. NT[0] is produced when the cell itself spikes, NT[5] is received when the presynaptic cell spikes.

3.1.4 Computing one time step

All calculations needed to compute the state of the cells for the next time step, are performed by the function "void compute(void)" in the source file COMPUTE.CPP. This is the hearth of the program, and uses most of the computer time while the program runs. It has been kept as compact as possible (112 lines) to improve performance.

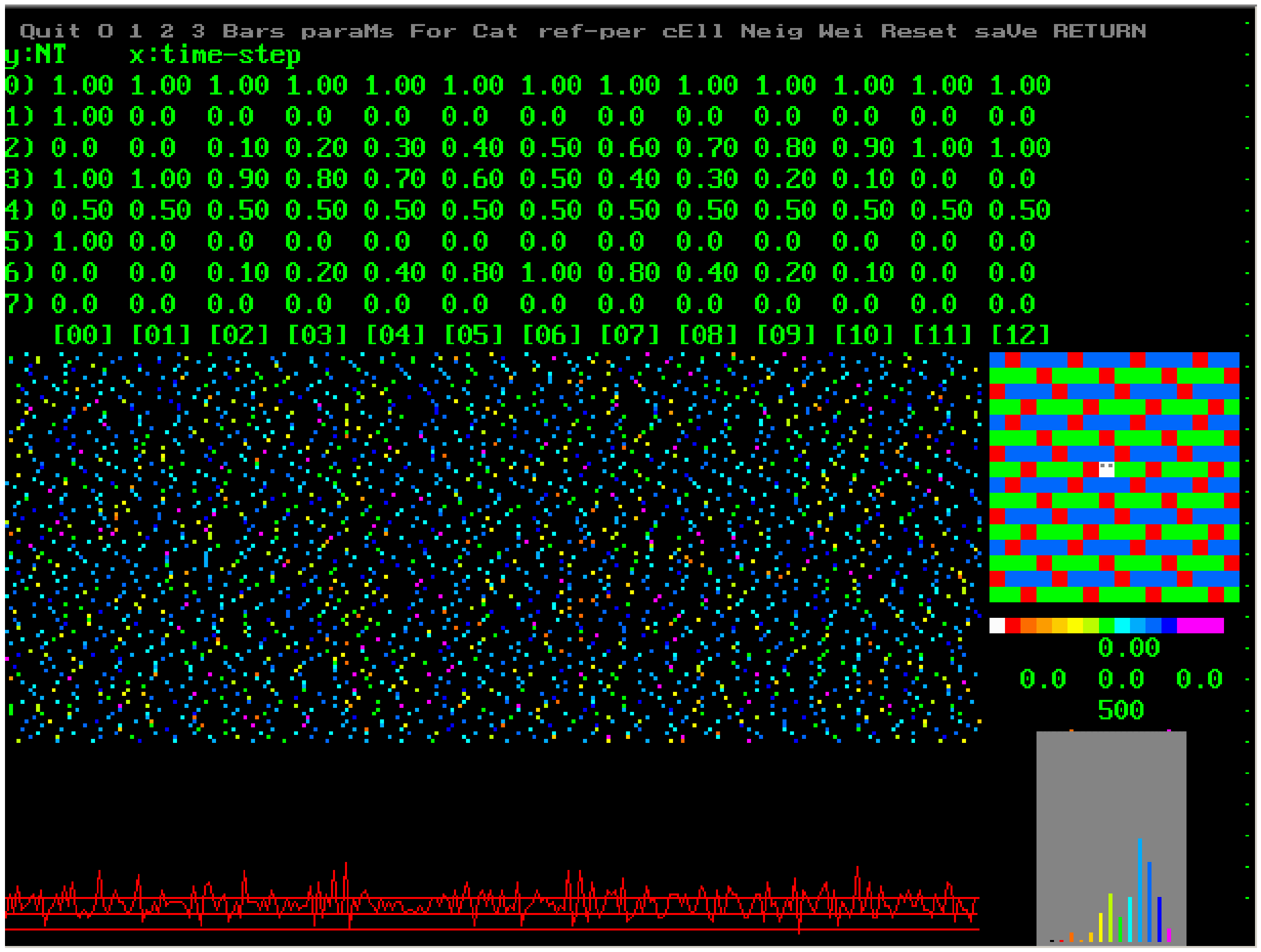
Another main reason for its small size, is debugging (finding and correcting programming errors) ease. When writing a program, it is recommended to run step-by-step every line of code, checking that all variables have the expected values. Normally, the expected values are easy to find "by hand" or with a pocket calculator, but with 100 cells and over 100 variables by cell, this is almost impossible. In theory, if a program is extremely short, it is possible to simply read the code and determine if there is a bug (error). Another alternative, is to include additional lines of code that check automatically for errors, e.g., a 0 value in a divisor. After debugging, the extra lines are eliminated, and the program runs faster. Both the "keep it simple" and the debugging code have been used in this case, as well as step-by-step tracing.

The computation (in pseudocode) is performed as follows:

```
For each cell:
   For each synapse in this cell:
      check if the presynaptic axon is spiking, if so,
      add to the synaptic store an amount of NT
      determined by the synaptic weight multiplied by
      the refractory-period-table parameter as indexed
      by the synapse's NT-type and the cell's state

      add the same amount to the corresponding NT
      store of the cell

      add to the synapse's secondary NT store the
      fraction of the primary NT determined by
      transformation parameter for that NT-type

   add to the cell's NT[0] store the amount determined
   by ATP multiplied by the refractory-period-table
   indexed by 0 and the LastTir (wavelength) of the
   cell

   substract from the cell's NT[0] store the amount
   determined by the refractory-period-table indexed
   by 1 and the cell's state

   add to the cell's secondary NT[0] the amount
   determined by the NT[0] multiplied by the
   transformation parameter[0]

   add to the cell's NT[1] store the amount determined
   by LAC multiplied by the refractory-period-table
   indexed by 1 and the state of the cell

   add to the cell's secondary NT[1] the amount
   determined by the NT[1] multiplied by the
   transformation parameter[1]

   the total input for the cell is calculated by
   adding all the cell's primary and secondary NTs
   (including NT[0] and NT[1]), each multiplied by
   its corresponding power parameter

   the total input is multiplied by the cell's
   refractory period parameter, indexed by the cell's
   state.
   If the result is bigger than the cell's
   NT-type threshold, and the amount of NT[0] is
   bigger than LAC, the cell's next state is 0,
   the LastTir is updated, and the next axon value
   is set to the cell's NT-type delay.
   If not, the next state is one more than the
   present state (maximum 12), and the axon is
   decremented by one (minimum -1)
```

```
(the state and axon are not updated at once, to
prevent the already computed cells to be one-step
before in time than the not yet computed cells.
The next part of the compute() function concerns
the learning algorithm, and is skipped here)

   the auxiliary input (electrode) is multiplied
   by its half-life parameter.

(all half-life parameters are smaller than 1.0,
this represents decay or catabolic processes)

   all the cell's primary and secondary NT stores
   are multiplied by their corresponding half-life
   parameters

(the learning algorithm is skipped here)

   for each synapse in this cell:
      multiply the synapse's primary and secondary
      NT stores by their corresponding half-life
      parameters

for all cells:
   update the cell's state
   update the axon's value
```

3.1.5 Sample screen



```
 Quit 0 1 2 3 Bars paraMs For Cat ref-per cEll Neig Wei Reset saVe RETURN
y:NT    x:time-step
0) 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00
1) 1.00 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
2) 0.0  0.0  0.10 0.20 0.30 0.40 0.50 0.60 0.70 0.80 0.90 1.00 1.00
3) 1.00 1.00 0.90 0.80 0.70 0.60 0.50 0.40 0.30 0.20 0.10 0.0  0.0
4) 0.50 0.50 0.50 0.50 0.50 0.50 0.50 0.50 0.50 0.50 0.50 0.50 0.50
5) 1.00 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
6) 0.0  0.0  0.10 0.20 0.40 0.80 1.00 0.80 0.40 0.20 0.10 0.0  0.0
7) 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
   [00] [01] [02] [03] [04] [05] [06] [07] [08] [09] [10] [11] [12]

                                                          0.00
                                                     0.0  0.0  0.0
                                                          500
```

Fig. 10 a sample full screen of the simulation program, in graphic-interactive mode

This is how the simulation looks like when run interactively by the user, when changes in parameters are desired. The top line is the menu, in this case the NT-ref. per. table is active (in green). There are many other menus, some of them not documented (the last ones mainly for program debugging). The most important part is the spike display, the part in the middle full of colored points. The time runs from left to right, and each vertical column shows the spiking cells, cell 0 at the top and cell 99 at the bottom (many variables start with 0 instead of 1, this is a convention used in language C). The cells are only shown when spiking (State 0), and are color-coded depending on their last wavelength. At the right, the red, green and blue-checkered square shows the "world", where the "frog" (little white square with eyes close to the middle) lives. The color-coding of the spikes' wavelength is the bar directly under the frog's world, 0 is at left (white) and 12 or more is at right (magenta). The red lines at the bottom, show an EEG-like track that simply displays how many cells are spiking at a given time-step. Finally, in the lower-right corner, an histogram of the actual wavelengths of all spikes is shown. The color coding is important when it comes to judge synchronization. In this case, the EEG line may suggest some synchronization, and the histogram is unclear, but the spike display shows no discernible pattern at all. Human vision is subjective, but frequently better able to perceive patterns and computers are.

3.2 Parallel computers

A fundamental design philosophy of the simulation program, is that is was written almost from the beginning to be easy to port to parallel computer hardware. This is partly due to the parallel nature of the cellular automata in which the program is based, and partly due to the expected growth on computer resources if the simulation should use a genetic algorithm.

Most PCs have a single processor using the serial Von Neumann architecture. This is the reason for the race to provide more megahertz or gigahertz clock speeds in typical computers. Only recently have multiple-processor ("cores") appeared in the market. However, since commercial software is written still only for serial computers, efforts to provide transparent parallelization at the hardware level result mostly in a waste of resources an little improvement in execution time (the same applies to 64-bit hardware, because the software is rarely re-compiled for such systems).

The theoretical limit for acceleration of execution time by using multiple processors, is given by Amdahl's law. Suppose that one has a program where 80% of the code is parallelizable (able to run independently in various processors) and the other 20% can only be executed in a serial way. If two processors are used, the run-time is 60% of the single-processor time. With four processors is 40%, with eight is 30%, with 16 is 25%, with 32 is 22.5% and so on. No matter how many processors are used, the serial part of the program will always take the same 20% of the original time, limits the maximum speed and makes adding processors wasteful.

Another problem of parallel software, is how much communication is needed between the different processors. Each unit running in its own processor, eventually needs input from other units, and needs to send its own output to other units. The communication between processing units must be synchronized, so the program usually cannot run faster than the slowest unit on each cycle. To complicate matters, the amount of data that has to be transferred usually grows much faster than the number of processors, to the point where a program spends most of the time transporting data and synchronizing units, and not actually computing.

These problems allow the use of parallel hardware only in problems that can be broken in many similar units, that take a similar time to process, and whose units need to communicate at best only with a few neighbors close to them. Examples of such problems are meteorological forecasts, aerodynamics/hydrodynamics, and some atomic particles' simulations.

From the viewpoint of cellular automata, these problems are not a fundamental constrain. Each cell can be run in its own processor due to the inherent parallel nature of the system. All cells take a similar time to complete a time step, and this time is identical if they have the same number of neighbors. Finally, communication is reduced to a single integer number by cell, the state (or, to implement delay, the axon's state). If the states are copied to a memory area that can be simultaneously accessed by many processes at the same time (some parallel computers have such special hardware), and there are as many processors as cells, the communication time does not grows at all with the number of cells.

Admittedly, these advantages were originally not foreseen or intended. Using cellular automata as a basis for neural simulation was almost an accident, and at the time (1980s) parallel computers were little known outside computer science. However, the simulation has been written to optimize the runtime of the procedure that computes the cells' next states. This procedure ("void compute(void)") is less than 200 lines long, can be fully parallelized, and uses well over 90% of the runtime. It uses an even greater part when graphics or the genetic algorithm are not used. The communication needs are extremely low: one byte by cell by iteration. All variables that define the cells, an the parameters used to define a model, are "global", i.e., accessible program-wide. Programming textbooks advice to avoid global variables whenever possible, specially in big programs. The simulation has been kept very compact over the years, so breaking this rule allows it to run faster, and giving each processor a copy of the parameters would greatly diminish the communication needs.

Hopefully, once a genetic algorithm is perfected and delivers promising results on a PC, the program can be ported to a parallel computer. Experience shows that using a new compiler and new hardware is not a trivial task, and it is difficult to gain access to parallel computers due to the heavy demand from other users. Whatever the problems may be found, they will probable not require changes of the essential data structures and algorithms.

4 SIMULATIONS I

4.1 The laboratory method

To explore the effects of various variables on the simulation's behavior, many combinations of parameters were run in the simulation program. Typical procedure was to change online the value of one of the parameters, reset the simulation and let it run again. When many different values of a parameter were to be systematically explored, sets of parameter files would be generated and tested in succession. The results were evaluated simply by looking at the spike display and the neurotransmitter patterns. This is an unsatisfactory, subjective way of searching for, and evaluating results. From the methodological point of view, it is subject to the free-parameters problem (2.3). In this preliminary phase there was no simple way of defining in advance the expected results, so the intention was simply to find "interesting" results. But, how should one try to develop an objective way of measuring "interesting" when one of its characteristics is that it is unexpected?

Each combination of parameters represents a different model. Defining thousands of them in this more or less random way and running them, delivered a preliminary result. Unsurprisingly, the simulations are chaotic systems. Big changes in one particular parameter frequently produce undistinguishable results, in a different configuration a change of the same parameter of less than one percent, produced extreme differences. This is a characteristic of chaotic systems, like the three-body-problem in physics. There is no general way to know in advance what will happen.

Eventually, some interesting and surprising behaviors were found, but due to the many free parameters used, the usefulness of so developed models is methodologically highly questionable.

A standard experimental technique used in psychology laboratories may be useful in this case. A popular, skeptical view of experimental analysis of behavior, is that "people are not rats, and the world is no Skinner box". This is of course true, but epistemologically irrelevant. The conditions in a laboratory are artificial and not found in real life, their purpose is to keep all possible variables constant, except one. If only the independent variable is manipulated, and this changes the dependent variable (behavior), a causality relationship has been found. The actual methodological criticism, is if this causality is also found in the real world, i.e., the generalization problem. If various individuals and animal species are tested, and finally a similar experiment with children in a kindergarten is performed, one may conclude that, e.g., aversive conditioning is a poor pedagogic practice, even if this was first found out giving rats electroshocks in a Skinner box.

In this context, the approach of systematically manipulating a single independent variable, and measuring the effects in the dependent variable(s), will be called here the "laboratory method". It is used widely in the natural sciences, and it is an integral part of the scientific method and the black box method.

Laboratory experiments are usually done before field research, because it is much easier to keep all variables constant than it is in the real world, but even then no perfect control can be achieved. Not two rats show identical behavior, since it is impossible to use genetically identical rats that have grown-up in identical environments an have the same history. This does not apply to computer simulations, in which the researcher has absolute control of every variable. With laboratory rats, experiments are performed only once with each animal, because they must be experimentally naïve. In a computer simulation, the variables can be easily reset to the same starting values (if these are random, the pseudo-random number generator is also reset). The simulation is then run again with another value of the independent variable, under guaranteed identical conditions for everything else.

Precisely this possibility of resetting the system is what makes an apparently trivial, standard scientific practice more important than in other contexts. The black box method can be impractical when the box has many different states, but if the system analysed can be reset to the initial state, the laboratory method can be applied. If the sequence of inputs affects the internal state and modifies the input/output, testing all possible input sequences after a reset can still provide a useful description of how the system functions. This combination of the black box and the laboratory methods, is simply another way to describe what a computer programmer frequently does during reverse engineering or debugging.

Another methodological aspect, is that the laboratory method is useful to address the problem of free parameters. Instead of changing many parameters from model to model, the results of changing a single parameter are examined within a single model. It may be the case that the other parameters, the ones kept constant, represent a completely unrealistic model. The point is that this does not prevent a better understanding of the parameter being studied. Combining the laboratory method with reverse engineering, has another advantage in this respect. Many of the model's features are not free parameters, but rather dictated by empirical evidence. An ANN that represents output as average firing rate, is freely choosing a parameter, an arbitrary non-linear function. If spikes are used in the model instead, there is a theoretical ground for it, namely that this is what neurons actually do.

The laboratory, black-box and reverse engineering methods will be used in the following to show how they can be used to evaluate the relevance of features in a spiking neural network. Once the method has been chosen, the next step is to find a theoretical framework that suggests which behavior should be investigated.

4.2 Animal model

To attain the goal of identifying which are the relevant features that neural-inspired cognition models should represent (see 1.3), the features of the most simple forms of nervous systems will be examined. Even if such systems are incapable of any relevant form of cognition, the neural features *sine qua non* must be already present, and it is plausible to expect that they are also important for cognitive processes. It may be the case that the most primitive features play no direct role in cognition, but even in this case they will not break the isomorphism between model and original.

When considering the most simple form of neural systems, it can be asked how and why neural systems evolved in the first place. Examining the fossil record delivers some tentative answers to this question. The geologists' consensus over the age of the planet earth is around 4.5 billion years, and the earliest fossil record of microbial life appears 3.8 billion years ago, maybe even as early as 4 billion years ago. In other words, life developed quite fast, and has been a feature of this planet for most of its existence. However, the first multicellular life forms appear in the fossil record much later, around 700 million years ago in the Vendian period. Most of the oldest preserved fossils belong to the so-called Ediacara fauna, but paleontologists have not yet agreed whether such organisms are animals, plants or something else. Ediacara organisms are extinct since the Cambrium period, so the question whether they had a nervous system may never be decided.

The only other group of multicellular organisms that predated the Cambrian period are the Cnidaria (Cartwright et al. 2007), a group that today includes coral polyps, jellyfishes and sweet-water hydras. This can be interpreted as meaning that the development of multicellular forms may be much more difficult than the development of life itself, and it is not known which are the critical factors that make such a development possible.

One of the requisites for the survival of life as we know it, is the ability of cells to maintain a constant internal equilibrium in the concentration of seawater ions, such as calcium, chloride, magnesium and others. Organisms across all branches of the phylogenetic tree, from archaea to humans, share a very similar equilibrium of seawater ions, at a concentration much lower than today's oceans. Indeed, this is one of the reasons proposed by biologists to argue that all life forms developed originally in seawater, at a time billions of years ago when the concentration of salts was different and lower than today. To maintain the intracellular ionic equilibrium despite a different and changing concentration in the environment, cells in all life kingdoms use so-called ion channels, specialized molecules in the membrane that control the exchange of ions with the exterior. Cells also use energy-driven metabolic mechanisms to overcome osmotic pressure, one of the best known is the sodium-potassium pump, which is known to have a decisive role in neurons' resting equilibrium.

The first working hypothesis is that at some point in evolution, organisms began to use their ion-equilibrium devices to communicate with each other cells, allowing the development of multicellular organisms. This is true even in the case of today's sponges, which lack a nervous system.

Communication through diffusion of atoms or molecules is very fast at short range, as shown by neurotransmitters in the synaptic cleft. However, beyond a given distance diffusion is too slow to coordinate widely separated parts of the organism. In humans this is illustrated by the relatively slow endocrine system when compared to the fast-communicating nervous system, despite the acceleration of hormone distribution due to our circulatory system. The second working hypothesis is, that in order to coordinate muscles far away from each other, making locomotion possible in big animals, the ion-communication devices were further developed to produce neurons.

The animal model used as target for the simulation is the jellyfish. It was chosen for two main reasons. The preservation of soft-bodied organisms as fossils is rare, yet probable jellyfish impressions date from the pre-cambrian, making them probably the oldest multicellular, non-extinct organisms. Recently, jellyfish fossils have been confirmed as dating as early as the Cambrian at least (Cartwright et al, 2007), and comb-jellies are probably the oldest animal form, instead of sponges (Dunn et al 2008) . The second reason, is that jellyfishes have the most primitive nervous system known, the so-called neural-net. The neurons in such systems are undifferentiated, and have no distinction between axons and dendrites. They resemble physically the spiny stellate neurons in mammalian cortex, showing undifferentiated processes extending in all directions, and relaying impulses in all directions. The neurons are spread over the whole body, with no concentrations in the form of ganglia or brain. So jellyfishes are not only probably the oldest animals with a nervous system, they also have the most simple one. There are classes like cubozoa that show more advanced features, like eyes and brain-like organs, but it is not surprising that some of them are somewhat more complex having had more time to evolve than most animals.

The coordination of far-away muscles is essential in order to let the whole animal contract its body at the same time -or in a wave-like fashion beginning at the center- otherwise it would not produce the water jet necessary for propulsion. The third working hypothesis for this simulation, is that coordination is the main task of the neural net, and a major candidate for the reason why a nervous system constitutes an evolutionary advantage.

None of the working hypotheses mentioned can be tested directly, this would require being witness of the transformation of single-cell organisms into multicellular animals, a process that took place millions of years ago. On one hand, this makes computer simulation the ideal tool to explore the problem. On the other hand, the purpose of the following simulations is not to propose a plausible evolutionary hypothesis, which is better left for biologists. Neither is the purpose to produce a cognitive simulation. Simulating jellyfish locomotion is simply a case example to address the methodological goals and hypotheses as stated in 1.2. Specifically, the proposed methods should help to identify relevant neural features that should be implemented in models of cognition, and provide insight in the way they function.

The chosen feature for this test, is the refractory period, and how does it influence an extremely simple nervous system. The production of spikes consumes energy, and sets a limit for the frequency of neurons. The question here, is what happens when the cell's own ref. per. influences not only the response to a given input, but also modulates the magnitude of the effective input itself. Biologically, this is equivalent to ask if the ref. per. could have had another form than today's. Alternatively, it is conceivable that nature could have evolved non-correlated ref. per. and input sensibility. In both cases, today's configuration and effects of the ref. per. could be an evolutionary accident, and possibly irrelevant for cognitive processes.

4.3 Method

The simulation program -version BCK08- was used, the models were defined using the

STRUC.TXT and PARAM.TXT in the same directory.

The connection structure for the cells used in simulations 4.5.1 to 4.5.4 was simply random, because this avoids making assumptions about the structure. Each cell has eight input synapses of NT[2] type, eight of NT[2], eight of NT[3], and eight of NT[4]. Only one type of synapse was active in each simulation, this is decided on run-time by setting the NT_PO of a single type to 0.1, and the rest to 0.0. This is done to compare the different types without starting the program each time with different model files.

A reasonable assumption about the structure would be that cells should have the highest probability of receiving input from their closest neighbors. This assumption was tested and delivered the same results. Since the structure file is much more complex, it will be ignored in the following.

The independent variable tested in the simulations, was the effect of various forms of the function that determines the absorption of NT-gated ions during the ref. per., the two-dimensional variable array REF_PER.

Before this could be done, values had to be found for all other parameters that would be kept constant in the simulations. The half-life of all NTs was set to 12 time-steps, which is probably a little longer than in neurons, but increases the temporal summation. The only exception to this was the half-life of the energy, NT[0], which was set to infinite half-life.

The NT power of NT[0], that represents the available energy, was set to 0.5. Because NT[0] is produced constantly, its amount is reduced only during spikes, and is unaffected by the ref. per., it constantly accumulates. Giving it a positive power eventually triggers spikes. This is needed because otherwise no input-less cell would ever fire. NT[0] simply guarantees in this way an absolute minimum of activity, but the other NTs are the ones that determine the actual activity modulation and the interaction between cells. In each simulation, only one of the other NTs influences cells, with an excitatory power of 0.1. If values for NT powers are bigger than the one of NT[0], the network's activity becomes very unstable, and a sensitive equilibrium using negative-powered NT is needed. The simultaneous development of two different neurotransmitters in jellyfishes, specially if they need a critical equilibrium, seems implausible.

The production of NT[0], determined by the ATP variable, was set to 0.012. The LAC variable, that determines of much energy is consumed by a spike, was set to 0.1. LAC must be greater than ATP, otherwise the amount of NT[0] will grow without limit and result in continuous tetanic spiking (and variable overflow).

The parameter values were chosen to produce a moderate amount of overall activity with the control value of the independent variable (4.5.1). All other parameters were set to default values and are irrelevant for the simulations.

4.5 Results

4.5.1 Constant refractory period

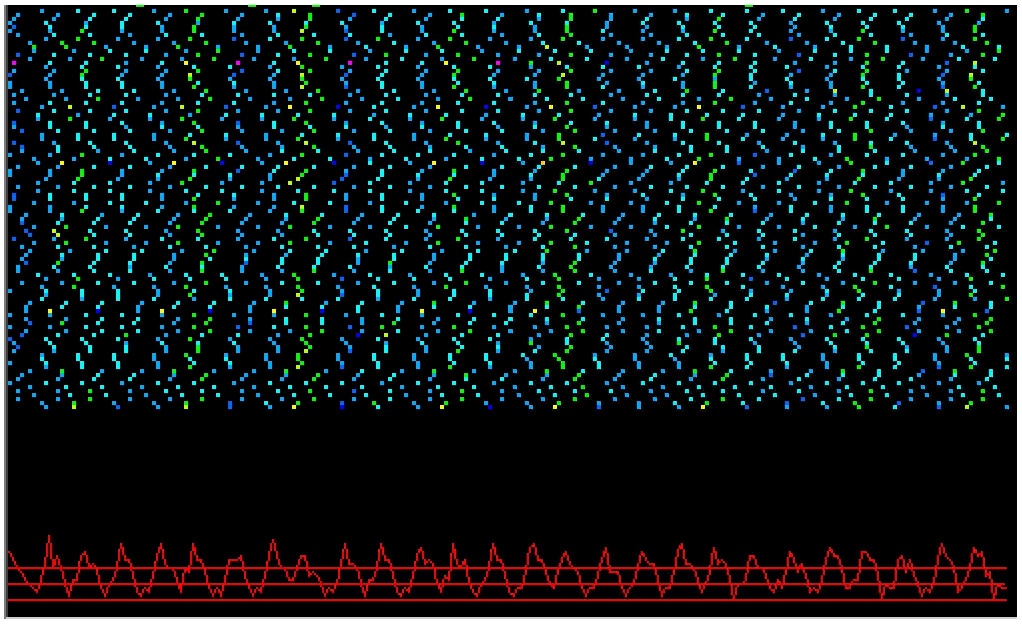


Fig. 11 Refractory period-independent NT influence (constant value)
`REF_PER[4]: 0.50 0.50 0.50 0.50 0.50 0.50 0.50 0.50 0.50 0.50 0.50 0.50 0.50`

This model show the results of constant (state-independent) modulation of the input from presynaptic cells through the ref. per. This is a control value of the parameter being examined, equivalent to no ref. per. at all. In other words, altough the cell has a maximum spike frequency limited by the need to regenerate its energy (one of the assumptions of most neural networks, as shown in 2.7.1), this is not a feature that causes state-dependant input/output.

It must be remembered that even though this is true for the NTs, the ref. per. still applies for the cell as a whole, for the activation function used has an S-shape. Looking at the EEG track alone, it appears to be a binding among the cells, though in the spike display the waves show a less clear pattern. In fact, some of the apparent synchronization is due to an artifact. Since the cells start with all NT values at 0, it takes them about the same time to start spiking. When the initial stores of NTs are started with random values, and this is repeated with different starting values of the random number generator, the synchronization is barely perceptible.

4.5.2 Decreasing refractory period

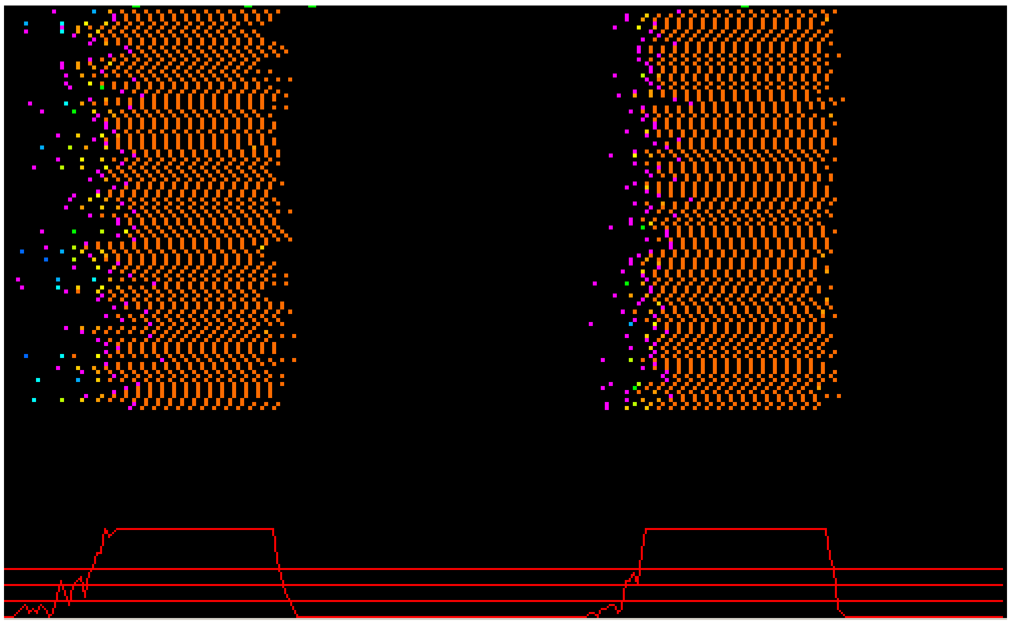


Fig. 12 Effect of decreasing values of NT ref. per.

```
REF_PER[3]: 1.00 1.00 0.90 0.80 0.70 0.60 0.50 0.40 0.30 0.20 0.10 0.00 0.00
```

In this simulation, the ref. per. modulation of presynaptic input produces a maximum receptivity for input during a spike and inmediatly afterwards, (i.e., in the absolute refractory period). In the following time, the cells become decreasingly receptive, and at the end the cell receive effectively no input at all. Following the "laboratory method" principle of changing only the independent variable, no other changes have been made. The cell's ref. per. still has the normal form, it is only its effect input that has been reversed.

When the influence of the NT is greatest just after a spike, and lowest at the end of the ref. per., a positive feedback ensues. Once the first cells fire, still inactive cells receive input, and they spike earlier than if no input were present. This is normal and can also be observed in heart cell in vitro. However, the decreasing effect of NT ref.per. results in a runaway speeding of the spikes, and the cells fire as soon as the cell-wide rep. per. allows. Eventually, the cells spend much more energy than they can produce on a given time, and some of them stop spiking. The decreased input of their neighbors triggers a another positive feedback, this time in the opposite direction. This behavior can be seen as a loose analogy to out-of-control fast spiking in tetanus. In this particular case, the spiking is so extreme that the EEG track has reached the top of the scale and looks flat, but even if it did hat not, the track would show no waves but noise.

4.5.3 Coincidence detector

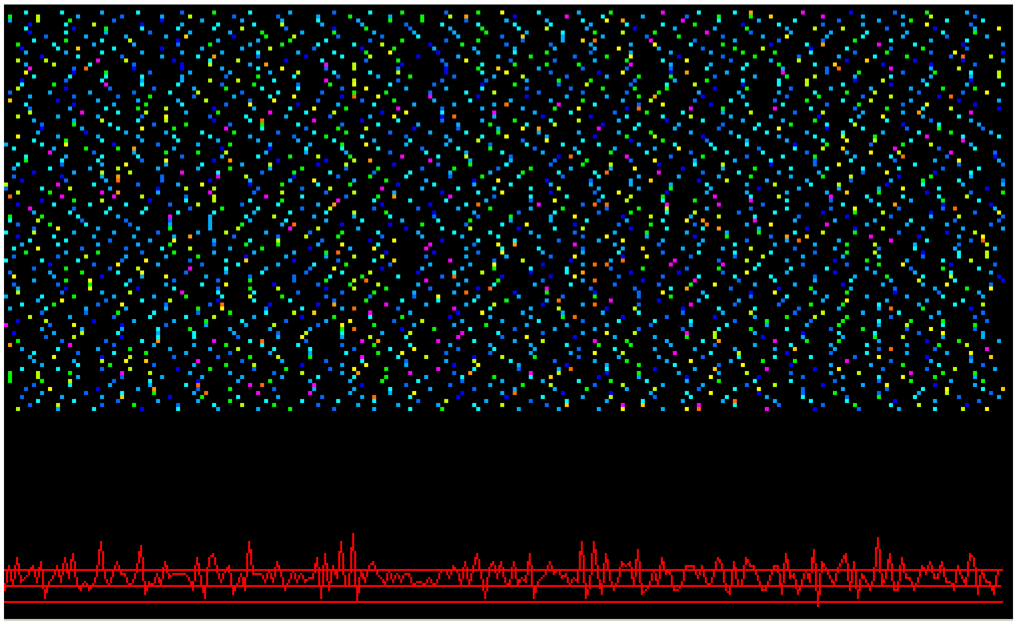


Fig. 13 (a part of Fig. 10) Effect of spike-only influence of NT
`REF_PER[5]: 1.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00`

This models shows the case when cells are allowed to receive presynaptic input only while spiking themselves.

Neuroscientists are very interested in finding the so-called "coincidence detector", a biochemical mechanism which realizes Hebb's rule. Such a mechanism should detect an immediately preceding or simultaneous spike in the presynaptic and postsynaptic cells. This is implanted in a straightforward way, by making sensitivity to presynaptic spikes only during the receiving cell's spike. The color-coded wavelength of the cells shows more variation then in 4.5.1, but no synchrony. A possible reason for this is discussed in 4.5.7.

As usual, no other parameters have been changed. Plotting in two dimensions the ref.per. input modulation, can be compared to a function with discrete values on the X-axis. This is an extreme case of the many that were tested, a "L-formed" function with binary results and a threshold, i.e., full reception of input during spikes and none afterwards. The effect of the refractory period on input needs not to be linear functions. Also tested were hundreds of other forms, most of them non-linear and non-monotonic functions, discussing all of them would be beyond of the scope of this work.

4.5.4 Increasing refractory period

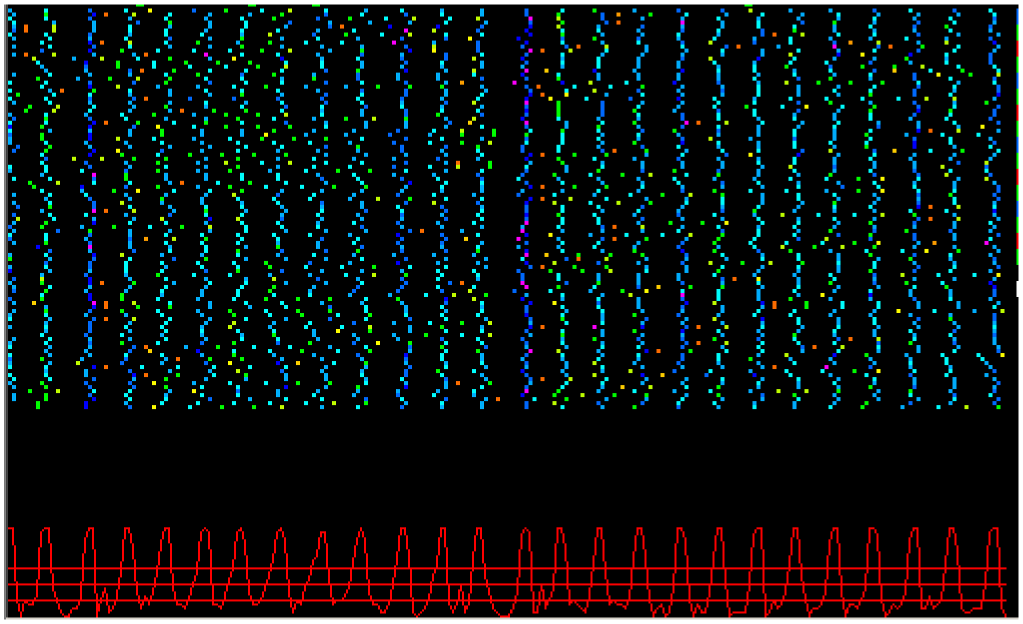


Fig. 14 Effect of increase of NT influence that parallels the cells ref. per.

```
REF_PER[2]: 0.00 0.00 0.10 0.20 0.30 0.40 0.50 0.60 0.70 0.80 0.90 1.00 1.00
```

In this model, the cell's ref. per. corresponds exactly to its influence on the reception of presynaptic input, both are 0 during a spike and the next time step, and increase linearly reaching a maximum eleven time steps after a spike.

This is the most interesting result of the four models. It uses a S-shaped increasing absorption of the NT. Here, the highest degree of synchronization can be observed, both in the EEG track and the spike display. The remarkable feature is not the synchronization of the network as a whole, but rather the fact that the individual cells are constantly changing their wavelength, as seen by their heterogeneous color coding. How comes that the network achieves this stable behavior as a whole with such "noisy" cells? The answer is that the increasing effect of the NT results in a negative feedback. Assume, for example, that the mean wavelength is 6. Cells that spike one time step before the wave's peak, would receive a less effective NT store, since they would be in the absolute refractory period when most other cells spike; in consequence, the will tend to spike later than the average next time. For cells spiking later than average, the effect is the opposite. They will have a more effective NT store because of NT -effect increases with time, and will tend to spike earlier next time.

4.5.5 Comb-Jellyfish connections

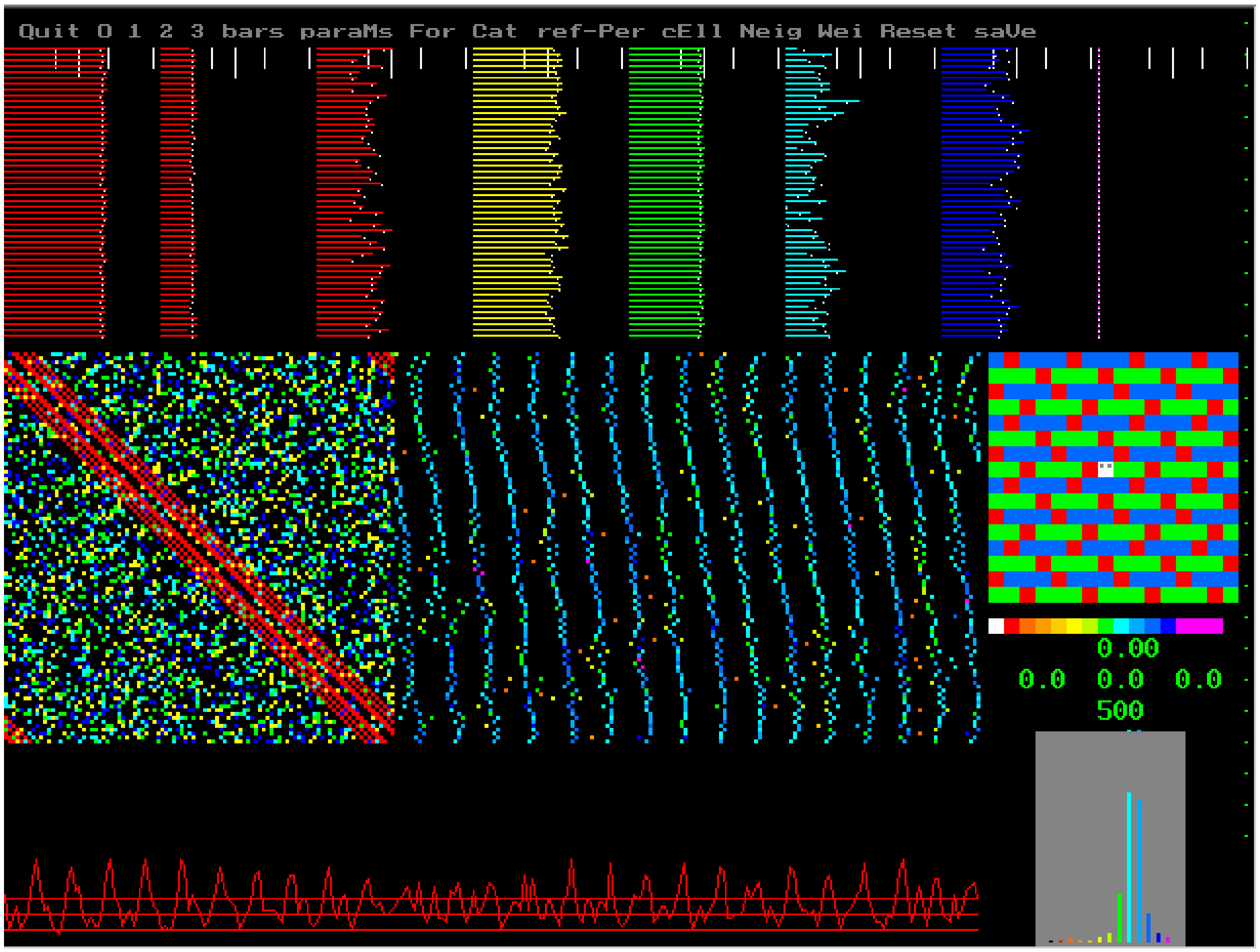

Quit 0 1 2 3 bars paraMs For Cat ref-Per cEll Neig Wei Reset saVe
0.00
0.0 0.0 0.0
500

Fig. 15 Increasing ref. per. with local neighborhood. The upper side of the screen (not shown in previous figures) shows the amounts of cell-wide NTs (the only active NT is the third column from the left). On the center, at left, the connection structure used is displayed. In the middle a short part of the spike display is visible.

Instead of a random connection structure used to model bell-shaped jellyfish, the connection structure used in this simulation simulates a comb jellyfish, which use lines of cilia for propulsion. Comb jellies are known to be the oldest multicellular life forms, even older than sponges (Cartwright et al, 2007).

For a cell numbered n, the presynaptic connections come from cells n-7, n-5, n-3, n-2, n+2, n+3, n+5 and n+7. The one-dimensional array of cells is bound to form a circle, so all cells have the same connection structure irrespective of position. The only active NT is colored red in the structure display at the center-left of the figure.

The metabolism parameters are the same as in 4.5.4, only the connection structure has been changed.
In this case, instead of a simultaneous contraction of the whole jellyfish, the simulation shows a spreading wave. Notice that although the structure is perfectly symmetrical, the

wave runs only in one -random- direction.

4.5.6 Wavelength adjustment

The overall activity of the models is determined by the ATP variable, which increases the store of NT[0], representing energy. There is a relationship between energy production (ATP), excitatory power of the energy substance (NT_PO[0]), and energy consumption (LAC), and the mean spike frequency. This can be examined by setting the power of all neurotransmitters to 0. In models 4.5.4 and 4.5.5 (assuming that the ATP production is not dependent on the refractory period, and NT[0] is not self-destroyed, i.e., NT_HL[0] = infinite), this provides a simple mechanism to regulate the speed of jellyfish locomotion. Slightly decreasing the value of ATP, the frequency of waves diminishes. If ATP is too small, only sporadic, unsynchronized spikes are produced. Conversely, if ATP is bigger the frequency of waves increases, if too big each wave is mixed with the last one, much like in Raffone & Wolters' (2001) paper. Therefore, a jellyfish can move slow, fast or stop, by simply changing the overall metabolism of muscles and neurons.

4.5.7 Axonal delay

One of the interesting results is that when NT is allowed to influence the postsynaptic cell only when both are spiking (Fig. 13), one would expect that synchronization is facilitated, since this is a way to implement the Hebb's rule. Instead, the pattern is more random even than with constant input. This result was puzzling, so the simulation was run step-by-step, examining the individual cells' values to understand how this happened.

The first probable explanation was that the input should be received shortly before, and not *while* both pre- and postsynaptic cells spike. But this would be very difficult to implement. How should a neuron know when the postsynaptic neuron is about to fire? even if it does know, the presynaptic neuron could be in absolute ref. per. when the postsynaptic one spikes, and would receive not input. If it were in the partial ref. per., it would receive less input than the other, and it's improbable that neurons can effectively synchronize when some of them are receiving more input than others. It could be that some stochastic negative feedback mechanism is at play here, like in Fig 14.

After pondering this problem, a second explanation suddenly becomes obvious. If two neurons fire at the same time, the spike has an inevitable delay due to the time it takes to spread along the axon. Short-axon neurons are not myelinated, long-axon ones are (in mammals), so the difference between local and far connections does not depend on axon length, or at least not completely. Still, delay is unavoidable, and it's longer than the duration of a spike.

Suppose, for example, that two neurons spike at the same time. A minimum delay of one ms would make them receive no input at all, since both would find themselves in the absolute ref. per. Suppose now that their wavelength is six ms, and their axonal delay is five ms. Both would receive the same amount of input, and synchrony would be maintained. A simple diagram illustrates this mechanism:

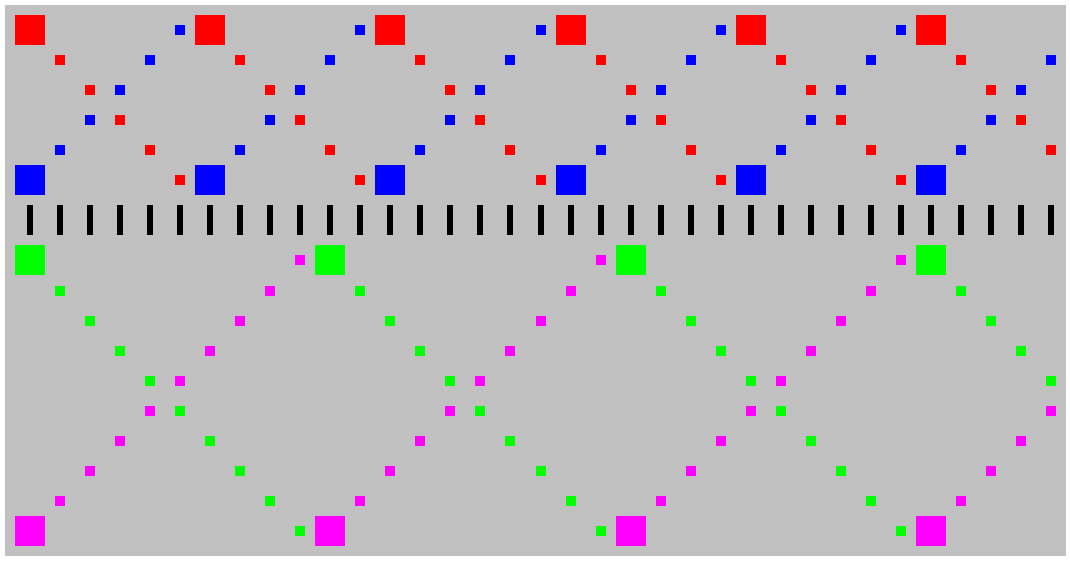


Fig. 16 Synchronization due to axon delay. The black lines mark time in ms, cells are the colored blocks, and the dots represent the spike traveling along the axon. In the upper side, cells have wavelength of 6ms and an axon delay of 5ms. In the lower part, wavelength is 10 and delay is 9.

If this analysis is correct, axonal delay is not simply an upper limit for synchronization, it may also play a more active role. Simulations were run with the model 4.5.3, using all values of delay, but no synchronization resulted. This may be due to the short NT sensitivity period of a single time-step.

It is difficult to say which feedback mechanisms could be at work if this were the case.

4.6 Discussion of simulations I

The model fulfills/confirms many of the goals and hypothesis as stated in 1.3.

The third hypothesis claims that it is possible to develop a biologically detailed model of a neural process, that needs modest computer resources, by using techniques typical of cellular automata. This has been implemented.

The second hypothesis claims that -one of the most basic characteristics of both automata and neurons- state-dependant input/output can be implemented using only simple neural features, i.e., spikes, ref. per., temporal summation, and others. This has been realized, following the reverse engineering method.

With respect to the goal of identifying relevant neural features for models, the simulations show how the influence of the refractory period modulates the dependant variable, synchronization in the jellyfish. They also show how the various factors interact with each other (ATP production determines the rate of locomotion), and helps to understand the system as a whole. It is of course questionable if input-modulation through the refractory period should be included in models of cognitive conflict, simply because it is a plausible mechanism for jellyfishes. However, the simulations could have shown that it is not relevant for jellyfishes, and that would make much less probable that it is relevant for cognitive models.

Synchronized cells as representations of cognitive contents was not a confirmed hypothesis, if only because it was the dependant and not the independent variable. I. e., synchronization shown to be the consequence of the a given form of the ref. per., and not the cause of a cognitive effect. Obviously, muscle coordination in a jellyfish cannot be seriously considered a cognitive process. Its importance lies in the fact that understanding synchronization mechanisms is necessary if they are to be used eventually to represent reaction times and their variability in human subjects. Strictly speaking, this is not a hypothesis of *this* work, but one originally proposed by von der Malsburg (1983) and confirmed in neurobiology (2.5.2). Unexpectedly, a stochastic synchronization (4.5.4) and its mechanism were found, apparently not previously described in neurobiological literature, and certainly unknown to the author. As it will be discussed later, this may be due to the statistics normally used to measure synchronization.

Evaluating if the free parameters make the models questionable, is not simple. Building a model, a researcher has the choice of using various nonlinear output functions, this would be a free parameter. Using actual neural features like refractory-period-dependant output is not “free”. However, the specific values used in the REF_PER table *are* free parameters. What the simulations show, is that nature had little choice when it comes to the *type* of function used (ref. per. Sensitivity increasing, 4.5.4). Many more forms of this function were tested without results, and the reader is invited to use the program with his/her own parameters. However, the sheer amount of possibilities makes an exhaustive test almost impossible. A conservative judgment would be that the methods used in these simulations have reduced the amount of free parameters as compared with a conventional BP-ANN. What is needed, is a method that would severely constrain or eliminate arbitrary choices by the researcher (like the specific values in REF_PER). This is what will be attempted in the following.

Last, and most important question given the emphasis of this work on methodology, is if the black-box, reverse engineering and laboratory method have proven to be useful. This is better left for the general discussion.

5 SIMULATIONS II

5.1 Genetic algorithms and free parameters

As shown in the previous simulations, using features that are actually present in neurons, instead of abstract programming devices, helps to diminish the danger of theory-less parameters that make a simulation questionable. One problem when using neurobiological facts as basis, is that the literature about them tends to focus on qualitative issues. There are exceptions to this, like the Goldman-Hodgkin-Katz equation or the Nerst equation, which give a precise numerical description of the electric and concentration behavior of ions in neurons. When it comes to a question like, e.g., duration and precise voltages of the ref. per., answers are mostly found only for specific neuron types, instead of a general description. Neurobiologists tend to concentrate on problems like the molecular structure of a neuroreceptor (qualitative), not on the average number of them (quantitative) in synapses. This is of course legitimate, if the goal is to develop a drug that influences such neuroreceptors, but is not very helpful in model making.

This lack of numerical data makes the selection of parameters for models a difficult, sometimes trial-and-error procedure. What is needed, is a way to avoid having the need to arbitrarily choose the specific values of those parameters.

The solution proposed here, is to use a genetic algorithm that chooses the parameter values by itself, without human intervention. This method simulates biological evolution, which is the way neural systems were developed. Using recombination (or sexual selection) and mutation, it evolves "genomes" (a set of parameter values that represent a model) and subjects them to selection. This process will be explained in detail later. At this point, it is important to know that the algorithm starts with many genomes, their parameter values are chosen at random, recombination and mutation are random, and the whole process may be run several times with different random values at start.

In their critic of free parameters in cognitive models, Roberts and Pashler (2000) write: *"A prediction is a statement of what a theory does and does not allow. When a theory has adjustable parameters, a particular fit is only one example of what it allows. To know what a theory predicts for a particular measurement, one needs to know all of what it allows (what else can be fit) and all of what it does not allow (what it cannot fit)."* (2000, p. 359). In the case of cognitive models (2.8), the free parameters are the different numbers of cells used in the models, their connections, the number of layers, etc. Basically, the argument is that if such parameters can be chosen in different ways for different models, all possibilities should be tested and most of them would probably not fit any empirical data. This means that there is no *one* model of cognitive conflict, but *many* models whose parameters are arbitrarily chosen ex-post-facto to fit a set of data. Such models make no predictions.

Genetic algorithms are ideally suited to solve this problem, because they use all -plausible- values for each parameter, and the selection procedure eliminates those that do not conform to the expected results. It both shows the parameter range that fits the theory, and those values that do not. This can be established by running the algorithm a number of times with different random values at the beginning. Each set of values selected for a given parameter, has a range, average and variability. The "theory" of a genetic algorithm is found in the selection procedure, specifically in the "fitness" computed for each particular set of parameters values, more about it in 5.11.

Genetic algorithms can deal with large numbers of parameters, yet tend to be very computationally intensive. This may seem to be contrary to this work's hypothesis that detailed models do not need to place a heavy burden on computer resources. The models themselves should demand modest resources, but this does not apply to the *search* for a good model.

The specific goal of the second series of simulations is to develop a prototype program, that using a genetic algorithm produces a model of neural synchronization. The prototype should help to evaluate the feasibility of following the genetic algorithm approach, using much more powerful parallel hardware. The feasibility can be tested in a normal desktop computer by reducing the number of parameters to be found to a subset of the total.

In order to attain this goal, a number of preliminary tasks needs be accomplished:
1) Modify the simulation program to run as fast as possible.
2) Finding a "positive case", i.e., a model that comes close to the expected results, i.e., synchronization similar or better than model 4.5.4. Its performance should be matched or exceeded by the genetic algorithm.
3) Selecting a number of parameters and a range of values of these. A portion of the positive case parameters' will remain constant, for the rest a range should be chosen, including the values of the positive case.
4) A selection procedure must be chosen.
5) A fitness measure must be chosen, probably the most critical task.

5.2 From neurotransmitters to ions

Beginnig on 2005, and using as starting point the BCK08 program used for the previous simulations, the program was ported to a more powerful compiler, renamed ION and an extensive series of modifications were performed. Though the general design philosophy based on celular automata and many of the data structures were not changed, the simulation program is quite different today, and is included as part of this work

The most important single modification of the main algorithm is to simulate ions instead of neurotransmitters. The main problem with neurotransmitters is that there are too many of them, and new ones keep being discovered. Even worse, the transmitter molecule docks in a postsynaptic neuroreceptor molecule, and it is the receptor that decides if the effect is excitatory or inhibitory, and how long it will last. For example, dopamine has five types of receptors and variants of these. One family (D1, D5) is excitatory and the second family (D2, D3, D4) is inhibitory. For the D4 receptor alone, Wikipedia quotes 18 variants (http://en.wikipedia.org/wiki/Dopamine_receptor, 2008). Different neuron types and anatomic locations have different dopamine receptors and variants, each with potentially different temporal characteristics.

Reducing the level of analysis, from ion channels to the ions themselves, the problem is enormously simplified. Neurotransmitters do not cross the cellular membranes nor influence the electrical state of neurons, ions do. Even better, there are only four neurotransmission-relevant ions: sodium, potassium, chloride and calcium (maybe magnesium should be added to the list, though neurology textbooks usually ignore it). The ions have a clear, either excitatory or inhibitory effect due to their electrical charge, and this is expressed in integer numbers: Na+, K+, Cl-, Ca++. How a neuron's resting potential, action potentials (spikes), temporal integration and many other basic input-output mechanisms work, is well understood as the effect of ions crossing the membrane. It can be calculated using standard formulas like the Goldman and the Nernst equation. There are only three variables accounting for neural electrical state: how many ion channels (neuroreceptors) are open for every ion type at a given time, the electrical equilibrium, and the ion concentration (osmotic pressure).

By following this reductionist approach, some components of cognitive systems are modelled as atoms. It seems implausible that going further to subatomic particles will be useful, so it's not possible to further reduce the problem. There is of course no guarantee that taking the black-box and reverse engineering to such a detailed level will produce useful models, but higher levels of analysis have not fared much better.

In the program BCK08 used in the first series of simulations, the neurotransmission process is not perfectly represented. In an excitatory postsynaptic potential ("EPSP"), after a presynaptic spike, the neurotransmitter diffuses through the synaptic cleft, docks on neuroreceptors and opens the channels for ions. The neurotransmitter is then broken by enzymes in the cleft. Despite the inactivation of the neurotransmitter, the additional Na+ ions that have crossed the membrane would remain there, if it were not for the Na/K pump that forces them out at a more or less constant rate. The net result is that Na concentration increases fast for around one millisecond, then returns slower to the resting level over some milliseconds.

The program simulates an incoming spike as a pulse that lasts for a single time-step. The synaptic store undergoes an exponential decay. A more accurate representation would require a counter for the maximum duration of NTs in the cleft, ion channels opening and closing in a non-monotonic, nonlinear fashion, and a Na/Ka pump that reduces linearly the ions while avoiding negative values. Despite their name, as used, NT stores actually represent the *effect* of neurotransmitters in the form of ions.

5.3 Program modifications

Given the relative success that such a simple neurotransmission implementation delivered in the previous simulations, it was kept relatively intact. To make explicit that not neurotransmitters are being simulated but rather their effects, the "NT" prefix in most variables was changed to IO (Input/Output substance or IOns).

The secondary NTs were originally included to simulate learning. These, along all parts of the main algorithm in COMPUTE.C that were related to learning, were eliminated.

Besides learning, secondary messengers were also used to simulate other long-term effects of spikes (like post-tetanic potentiation or LPT). However, it is still possible to define additional synapses for the same presynaptic cell, and define IOs with other time characteristics to represent C-AMP, G-proteins or calcium ions, for example. The normal IOs and the additional ones are linked by the fact that they are received during the spikes of the same presynaptic cell. Even if they are not used for the present simulations, the program has not lost this functionality in principle.

Another type of input, "virtual synapses" were added. These can be used to define synapses that receive their input (point to) not to other cell's axons, but to the cell's own State variable, or a constant value of zero (an always firing spike). Using virtual synapses, the program needs no special lines of code to account for energy or fatigue processes. Instead of the dedicated variables ATP, LAC, NT[0] and NT[1], used to represent energy management, virtual synapses are used. The IO type is chosen by the user, it can be any allowed type (0 to 7), but for consistency in the following the IO[0] represents enerby (IO[1] represents fatigue, but is not used in the simulations and its power is set to 0).

Each cell has one virtual synapse of IO type 0, pointing to 0. Its effect on the cell's input integration is controlled by its synaptic weight, half-life (HL_IO[0] set to 150 time steps), ref. per. (REF_PER[0], inactive) and power (IO_PO[0], set to 1.0). Given that the ref. per. is set to 0, energy should have no effect. This is not the case, because a new parameter table has been added to implement constant decay of IOs. It is read from the PARAM.TXT file, and in the case of IO[0] it has the values:
 -0.10 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01
These values are added to the IO store of the appropriate type, indexed by the cell's State variable, as usual. The first value, corresponding to State 0 is negative, so IO[0] will be decremented by 0.1 on each spike. The other states produce an increment of 0.01. This means that IO[0] will increase if the cell has a wavelength bigger than 10, and decrease if smaller than 9.

Virtual synapses are also used to substitute the auxiliary input, which connects the input areas of the networkwith the "world". The program includes, as before, a retina which in turn receives its input from the world. The virtual synapses are connected to the retina, whose values are set to 0 (spike) when input is received. The use of virtual synapses means that all input to the cell is received exclusively from synapses, and all of them are processed in exactly the same way. This makes the program's code more simple and uniform, and gives the user control over parts of the simulation that otherwise would have to be rewritten and compiled. The price of this, is that now all such features must be provided by the user trough the structure and parameter files.

The ref. per. for the cell as a whole, previously part of the source code, is now read as a parameter array from the PARAM.TXT file, under the name >REF_PER_CELL. The reason for this, is that neurotransmitters are electrically neutral, their diffusion and docking on neureceptors is not affected by the electrical state of the postsynaptic membrane. Moreover, the spike itself and the refractory period -both absolute and relative- are processes that run always in the same fashion, unaffected by NT sitting outside the membrane. With the possible exception of Calcium, ions -while each on a different way- always do the same. This means that instead of painstakingly testing thousands of combinations for the individual IOs in the REF_PER table, all that may be needed is the right form for the REF_PER_CELL array. One may compute with neurobiological simulation software the values for the effect of the ref. per. over individual ions if needed, but the important feature is the response of the cell as a whole.

5.3.1 Cell's structure and main algorithm

The cell's data structure has changed little as a consequence of the modifications:

```
extern int    *Type,             //cell's default neurotransmiter
              *State,            //cell's "clock" (0 == depolarization)
              *Axon,             //delay in transmission of spikes
              *WaveL,            //last wavelength
              *WaveL1,           //last but one wavelength
              *SynNum,           //number of synapses declared
              *Posflag;          //positive flag, catch negative ion amounts
extern float  *TotalIo;          //total input*equa.*power*cell ref.per.
extern float  (*Cell_Io)[MAXIO],  //total cell's store of neurotrans/ions
              (*SynEq)  [MAXIO];  //equalizes number of syn. by NT/ion
extern int    (*SynType)[MAX_SYN],//types of declared synapses
              (*SynPos) [MAX_SYN];//neighbor' number, negative:free_syn
extern int   *(*Syn)    [MAX_SYN];//points to neighbor's Axon
extern float  (*SynWei) [MAX_SYN],//weights of synapses
              (*SyIo)   [MAX_SYN];//Synaptic Input/Output, Neurotrans & Ions
```

Secondary NTs and AusInp (the last one is now received through virtual synapses) have been eliminated. A few new variables have been added. Sustituting LastTir, WaveL and WaveL1, (last-but-one) record the wavelength. Posflag signals negative amounts of IOs. Cell_Io[] substitutes NtInpP[], and TotalIo is now used instead of a local COMPUTE.C variable, to store the cell-wide input.

A description in pseudocode of the streamlined main algorithm:

```
For each cell:
     set all cell's IO stores to 0
     set total input to 0
     For each synapse in this cell:
          multiply each synaptic store by its corresponding half-life
          add/substract the linear decay

          check if the presynaptic axon is spiking, if so,
          add to the synaptic store an amount of IO
          determined by the synaptic weight multiplied by
          the refractory-period-table parameter as indexed
          by the synapse's IO-type and the cell's state

     add the synapse's IO store to the corresponding IO
     store of the cell

     add to the cell's total input each cell's IO store multiplied
     by its corresponding power and equalization parameter.

     if the total input multiplied by the cell's refractory period
     is bigger than the cell's threshold do:
          set the next state to 0
          set the next axon to 0
          set last-but-one wavelength to the last wavelength’s value
          set the last wavelength to the state's value
     if not, do:
          increment by one the state
          increment by the axon

For each cell:
     update the cell's state
     if state is bigger than 12 do:
          set state to 12
          if axon bigger than 24 do:
                set last-but-one wavelength to 24
          set wavelength to 12
          if total input is not negative do:
               set state and next axon to 0 with a probability of 1%
     update the cell's axon
```

The only problem of this algorithm is, paradoxically, that inactive cells (State>12) take a little longer to compute than active ones, which must be accounted for in a parallel computer.

A connection structure was defined, similar to the one used by Botvinick et al (2001) for the Stroop model (2.8). The purpose of using this connection structure, is not to produce a model of the Stroop task (although it would make it easier in the future). Its purpose is to have a connection structure where positive feedback stimulation within groups, lateral inhibition, and synchronization of cells' groups spreading to the next layer can be observed and analysed.

The main differences are: instead of single units, groups of 8 cells are used. There are 6 input groups:

```
cells   0 to  7 color red
cells   8 to 15 color green
cells  16 to 23 task: name color
cells  24 to 31 task: read word
cells  32 to 39 word "red"
cells  40 to 47 word "green"
```

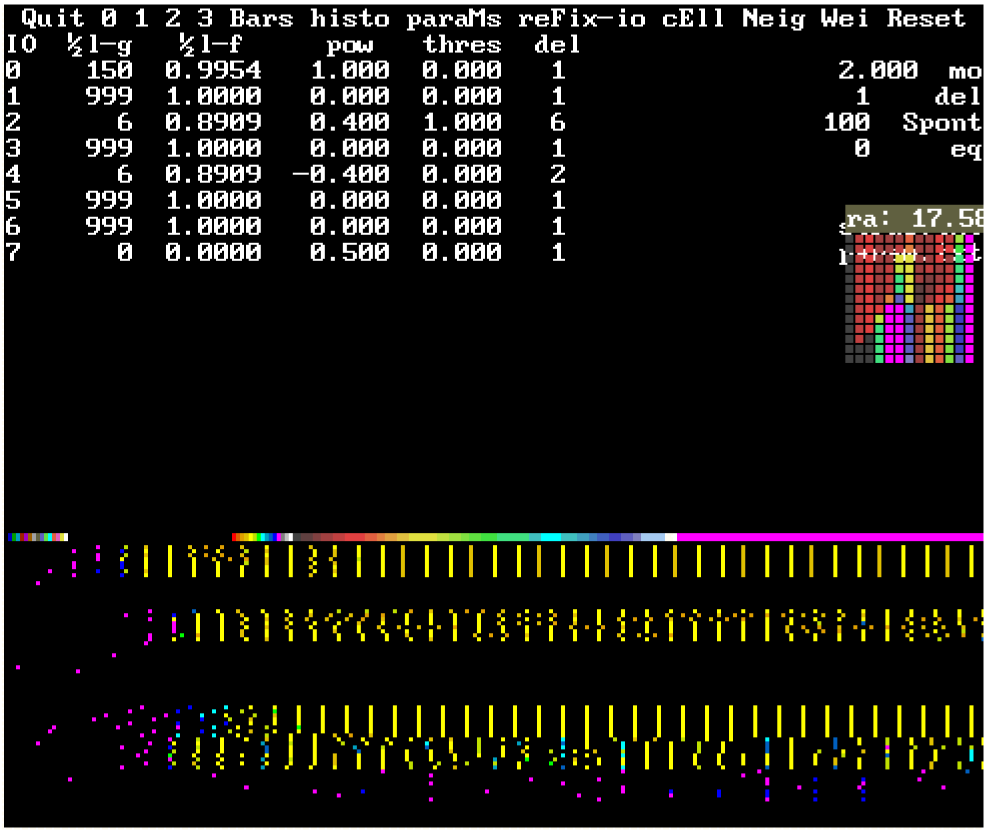
Each group has four excitatory (type 2) synapses with other cells of the same group, and four inhibitory (type 4) with the opposite stimulus/task (this is known as lateral inhibition). The stimulation is received through virtual synapses to the retina, in turn the retina receives input from the world. The pattern in the world allows to use all combinations of stimulus (including neutral ones) and tasks.

A middle layer receives input from the input cells. It is divided in 4 groups corresponding to "name color red", "name color green", "read word red" and "read word green". Each group's cell has two excitatory synapses with the appropriate stimulus, two excitatory with the task, two inhibitory ones with the opposite stimulus, and finally two inhibitory ones with the opposite task.

Finally, there are two motor (i.e., response) groups of cells, using the same structure of the previous layer and receiving input only from that layer. The connections in detail can be found in the file STRUC.TXT. All synapses have a weight of 1.0, and all have a total of four excitatory inputs and four inhibitory ones.

5.3.2 Reference model

After testing only around 50 parameter values combinations, a positive case model was found, clearly showing synchronization:



```
IO  ½l-g   ½l-f      pow    thres   del
0    150  0.9954   1.000   0.000    1                     2.000   mo
1    999  1.0000   0.000   0.000    1                       1      del
2      6  0.8909   0.400   1.000    6                     100   Spont
3    999  1.0000   0.000   0.000    1                       0       eq
4      6  0.8909  -0.400   0.000    2
5    999  1.0000   0.000   0.000    1
6    999  1.0000   0.000   0.000    1                        ra:  17.58
7      0  0.0000   0.500   0.000    1
```

Fig. 17 Reference model to test a genetic algorithm

-The model shows fast synchronization of cell groups.
-The synchronization is not totally stable for the individual cells.
-The groups maintain a wave for a while, but within some hundreds of time steps de-synchronize.
-The wave pattern spreads from input to hidden layer to output layer, though the last one is barely perceptible.
-The synchronization is remarkable because the ref. per. used is constant (4.5.1), unlike the one that produced stable synchronization in the first series of simulations
-The synchronization is, in a sense, too stable. If the input is changed, the pattern shown does not change. Only if the simulation is reset, and the input is changed, other groups of cells synchronize.

Though the model is far from perfect, it can be a basis on which to test a genetic algorithm, this is the "positive case" which the algorithm should find, as long as the way the fitness measurement can grasp its essential properties.

At this point, it is necessary to provide the theory behind genetic algorithms and a precise description how they work.

5.4 Combinatorial Explosions

The simulation program was originally written to implement a construction kit, it offers maximum flexibility when it comes to choose which are the relevant variables and the values of those. This means that there are many more variables than necessary, because it was not possible in advance to know what the users may consider relevant. The possible values for most variables are many, in the case of real-number variables (like half-life values for the effect of ions) they are infinite in theory. Obviously, some way must be found to limit the number of combinations that have to be tried. The first task is to determine the range, a minimum and maximum value for every variable. Second, the resolution of the variable, i.e. how many different values are to be allowed. One may choose for a real variable, for example, values starting at the minimum and increasing in 10% steps until they reach the maximum; or one may choose increases of 1% which delivers ten times as many values to be tested. Since combinations of all variables are being tested, the problem grows extremely fast when variables are added or their resolution increases. Suppose for example, that there are only six variables, each with 15 possible values. The number of combinations is 15 to the sixth power, or 11390625 different combinations. Testing all at a rate of one per second, means more than 131 days of computer time. Such extremely fast-growing problems are common in computer science, a fact known as "combinatorial explosion". The number of combinations in this particular problem is actually so big that it is not possible to calculate it in a pocket calculator, and rapidly overflows the maximum size of numbers normally used in computers. An actual calculation of the time required for an exhaustive search is futile, since it almost surely exceeds the age of the universe.

5.5 Search space and local maxima

The number of different values allowed for each variable is also crucial because choosing too few may prevent finding an optimum solution. To use an analogy, suppose one were required to find the tallest building in a big city measuring five by five kilometers. The problem has only two variables, latitude and longitude. The city is explored defining a grid whose squares measure 200 meters on the side, so there are 26 possible latitudes and as many longitudes. One is allowed to measure the elevation only at the crossing points on the grid, so changing one’s measurement position means making leaps of 200 meters north, south, east or west. There are 676 such crossing points, at which one will find street-level elevations, the top of buildings or even traffic lights. Using a grid of 200 meter squares, one may find parks, living quarters and probably the financial district, but instead of the tallest building one may end up with the position of a neighborhood church. Halving the size of the squares to 100 meters, the search space grows to 2601 positions, and one may still miss the tallest building if the building has a narrow top floor and one is not lucky. Using a grid of 10 meter squares guarantees that one will find the tallest building, but one can still miss the TV antenna on the top, and the number of positions to search has grown to more than a quarter of a million.

Many algorithms get trapped in these so-called local maxima, points in which even searching in arbitrarily small distances around the last point will only deliver worse solutions that the last one. To follow our analogy, even if one reduces the size of the leaps from one's position, it will not help if one is on top of a factory in an industrial area. At best, one will find a bigger factory, but not the Empire State Building. The chances of avoiding local maxima depend to a great degree on the characteristics of the search space, and are better if the rate of change of the solution is slow as one searches around one's last position. In our analogy, it means it is easier to find the tallest mountain in a mountain range than finding the tallest building in a city. In the last case, the elevation may vary from street-level to the highest point within one meter of horizontal distance.

What kind of "landscape" can be expected in the search for neural synchronization? since the simulation is chaotic, there may be great flatlands, soft hills, bottomless pits and the occasional Eiffel Tower or Matterhorn. All that can be realistically expected is to find at least some mountain range, without knowing if it's the Alps or the Himalayas.

5.6 Heuristics

When confronted with a search space to big or too complex, programmers may abandon conventional algorithms that guarantee to find the best solution. They develop instead what is called a heuristic, the computer version of a –sophisticated- rule of thumb. Heuristics do not guarantee an optimum solution, but rather a solution "good enough". This is the strategy of choice in so called NP-hard (non-deterministic polynomial-time hard) problems. These include problems sometimes apparently trivial, like choosing the order of a number of cities to be visited (traveling salesman problem), arranging objects in a container using the least space (knapsack problem) or even playing Tetris.

Proving that a given problem is actually NP-hard is very difficult, and can be the subject for a mathematician's doctoral work. Let us simply assume at this point that the subject of this work has such a big solution search space, that it is justified to use a heuristic to find it.

5.7 Genetic algorithms

One family of heuristics that has been used for NP-hard problems is called the "genetic algorithm" class. These algorithms try to imitate the way nature uses to develop life forms adapted to their environments as described by Darwin and modern biology. Many of the terms used in genetic algorithms are taken from biology, but should be understood in this context as a simplistic analogy. Hopefully the reader will manage not to confuse real biological terms with those of artificial neural networks and genetic algorithms.

The basic idea is to choose the values ("genes") for a fixed set variables ("chromosomes" or "genomes") at random. Each genome represents an individual within a large population. Then, each genome is tested against some criteria to find out how good it at solving the problem in question ("fitness"). Then, some genomes are chosen ("selection") and some of their genes are changed in a random way ("mutation"), or various genomes exchange some of the values ("crossover" or "recombination", the equivalent in higher life forms is sexual reproduction). Then the whole cycle repeats ("generation", not to be confused with a cellular automaton's generation), until the best solutions stop improving or a given number of cycles is reached.

A step-by-step description:

1. Choose the genes of each genome in the initial population at random.
2. Evaluate the fitness of each genome in the population
3. Repeat the following steps:
   A) Select best-ranking genomes to reproduce
   B) Breed new generation through crossover and/or mutation and give birth to offspring.
   C) Evaluate the individual fitness of the offspring.
   D) Replace the worst ranked part of population with offspring
   E) If the criteria for termination is reached, end. If not, go to A

5.8 An hypothetical example

A practical example of a problem where genetic algorithms are used, is the design of camera lenses. In this case, the optical engineer would first set the number of genes i.e., individual lenses to be used. Each lens may be defined by five genes: the curvatures of the forward and backward surfaces, their positions in the optical axis, and the refractive index of the glass. In this specific case, constrains are chosen like maximum curvature (say +4 to -4 diopters), available types of glass, the position of the back surface must be behind the frontal one, and so on.

If the gene with the maximum number of values is curvature, say hundred (not necessarily at regular intervals), The engineer should choose a population number that makes probable that all possible values are represented several times, say thousand genomes. If the population is too small, the best value for one curvature may land together with poor values for the other genes, in a genome with low fitness. The selection process would wipe it out from the genetic pool before it has a chance of reproducing.

Then the algorithm judges how good each genome is by calculating its spherical aberration, pincushion distortion, focal length, etc. and combining all in a single number. Once each fitness is calculated and the genomes are sorted by fitness, selection begins. The most simple way is truncation selection. From the fittest two thirds of the genomes, pairs are chosen at random and the offspring receives at random a gene from one of the "parents" (constrains may greatly complicate the selection, our engineer must find a way to prevent the third surface being forward of the second one, for example). The offspring then substitutes the un-fittest third of the population, their fitness is calculated, the genomes are sorted again and the next round of selection begins.

Values for the genes are limited by the engineer's imagination and available computer resources, but the advantage of such algorithms is that genes that the engineer would have never combined, have a chance because the selection process has no prejudice, it's simply random. The competence of our engineer is not the inspiration for a new design, but rather in the judicious way of choosing gene values, fitness evaluation procedure, and selection method. The results are often enough innovative and counterintuitive camera lenses, but this process is rarely mentioned in the press.

5.9 Fitness proportionate selection

The truncation selection described above is the most simple one, but in some cases it delivers poor results. The reason is that the fittest genomes have no higher chance of reproducing than the ones just above the lower third cut-off. Another reason, is that if the lower third of the population is eliminated in each generation, many of the "good genes" risk being eliminated from the genetic pool if they are part of overall poor genomes.

A more complex method, is the "fitness proportionate selection". It is also known as roulette-wheel selection, and this analogy is easy to illustrate. As in truncation, the process begins by sorting the genomes according to their fitness. Now, instead of producing a whole new generation of genomes that substitutes the lower third of the population, only one new genome is created every time. The fitness of all sorted genomes is added, this represents the total of pockets in the roulette. Instead of giving all genomes equal chances, each is assigned a number of pockets proportionate to its fitness. Even the un-fittest genomes stand a chance of getting selected, their chances are simply better (more pockets) if their fitness is higher. The ball (a pseudorandom number) rolls and falls in a pocket, the "mother" has been selected. Then the ball rolls again, and falls in the "father's" pocket. Of course, the same genome is not allowed to be mother and father for the same "baby", if that happens the ball rolls again until it chooses another father. Once both parents have been chosen, the baby is created by taking at random for every gene the corresponding one from either the mother or the father. Once the "baby" genome is tested for fitness, it is compared with the existing genomes beginning at the bottom of the ranking. If the baby is worse than the lowest-fitness genome, it dies and a new baby from newly chosen parents is produced like before. Otherwise, the baby is sorted in the proper place, all lower-fitness genomes are shifted one place downwards, and the lowest genome dies since there's no place for it in a constant population.

The roulette selection improves the efficiency of the sorting algorithm, but the more important advantage is that good genes whose genome has a low fitness have a better chance of reproduction.

5.10 Measuring synchronization

It is difficult to describe with precision which kind of behavior the models should produce. There is detailed information available over synchronization of pairs of neurons, less data over small groups, and none over whole nervous systems. It is of course impossible to place electrodes on more that a very small proportion of all neurons of an organism. There is some evidence of a natural neural tendency to synchronization. For example, neurons in the primary visual cortex of humans show a big-amplitude alpha wave (8-12 Hz) with closed eyes. In the case of a grand mal epileptic seizure, most neurons in the cortex spike together at around 3 Hz. It seems plausible that brains are chaotic systems that must perform a delicate balance between too little and too much synchronization, and this balance is poorly understood. For the purposes of the following simulations, it is only expected that active cell groups will synchronize at high frequency, while inactive groups will either not synchronize or do it at low frequency. At any rate, a synchrony measurement is needed for the simulation.

A genetic algorithm requires a fast, automatic measurement of the expected performance of the model. One would expected that standard statistic methods used by the scientific community would be easy to find. Yet no conventional statistic measure promises practical results when used with a genetic algorithm.

This problem is far from obvious. After all, synchronization between pairs of neurons within a group of around hundred, has been measured using a single electrode by Ahissar et al. (1992) using cross-correlation. This statistic is frequently used by engineers to perform signal analysis, usually with the help of special hardware due to the heavy computational resources it requires in real time. Instead of implementing it, more simple statistics were tried.

It seems simple enough to calculate a Pearson correlation coefficient between the firing time of two cells, even if the spike themselves have the discrete values "silent" and "firing". A simulation has the advantage of making accessible more information that is available to neurobiologists. Instead of correlating spikes, one may take the State variables of the cells, which should be the same or close, even when they are not firing, and delivers values from 0 to 12. Such a correlation is fast to compute, though it increases the memory requirements because a record of the states must be kept over a minimum of two or more wavelengths. The situation is more complex when groups of cells have to be accounted. With a group of cells, a general synchronization would require (n-1)*n Pearson calculations, each using over 24 value pairs. With 100 cells it means 9900 coefficients. The time it takes to calculate the synchronization may even be longer than running the simulation itself, partly because of heavy floating-point arithmetic.

This should not have been be a serious problem, but during the debugging of the statistic a more fundamental problem was noticed. Some of the cell pairs were showing strong correlations, yet they belonged to different sensor areas and were out of phase. They should not show any correlation, because they had no synapses among them. After an unsuccessful search for a program error, none was found. Whenever two cells had the same firing rate, the phase would produce a statistic artifact. Out-of-phase by half a wavelength produces 0 correlation, the closest they are in phase delay the higher the value, reaching 1 if they fire together. The statistic would be influenced by the artifact of unrelated cells firing together on the same frequency, after all, that is what synchronization looks like. Then, another problem was found. The maximum State is limited to 12, so completely inactive cells also show perfect correlation. It was tried adding a variable not tied to State of WaveL, that will simply record how many time-steps had gone since the last spike, even at the cost of eventually overflowing the variable and getting negative values. But the problem persisted, since inactive cells are evaluated by Pearson as a perfectly linear correlation, with absolute values differing only in the constant no matter how big the values are. Pearson correlation is designed to detect two linear equations of the form ax+b and factor out a and b, it ignores synchronization altogether.

Using both the State and WaveL variables of the cells was also considered, but Pearson and non-parametric correlation measures are designed for two variables, not two *pairs* of variables. Other alternatives like factor analysis can handle many variables, but factor analysis relies on a Pearson correlation table anyway, would need a heavy programming overhead, and would run orders of magnitude slower than an already computer-intensive simulation.

Cross correlation would have a heavier computing burden to carry, and would fail partly on the same grounds. There is another important factor to consider. As shown in 4.5.4, in the case of stochastic synchronization it is possible to have a highly stable wave produced by unstable cells. Since the wavelength of individual cells is constantly changing, they would show a low cross correlation with the network as a whole. Worse, correlation between two cells would be closer to 0, despite the fact that both are part of the same wave. So cross correlation is in this case much more affected by noise than it would be in other fields, and bias the fitness measurement against stochastic synchronization. This may even be the reason why such phenomena are not included in basic neural simulation literature.

An interesting piece of anecdotal evidence for this, is that despite decades of neural research using electrodes, Singer's team discovery of neural synchronization (Gray et al.,1989) was not due to statistical analysis, it was simply *heard*: *"One day the experiment did not work very welly, because some electrodes had no signal. Searching for an error, we changed the filters, and because the amplifiers were connected to a loudspeaker, a sound was heard, something like a cat purring. [...] We had an indicator for a very synchronized signal"* (Singer, 2000). It seems that human senses are better than statistics when it comes to detecting a signal embedded in noise.

Genetic algorithms do not require, however, a commonly accepted way of defining the fitness of an individual, they simply require a fitness function that produces the expected results. The fitness function is of no relevance for the model itself, it is simply a programming tool. Without a background of standard methods for genetic algorithms in cognitive modeling, there is no choice but to develop synchronization-fitness measurement from scratch.

5.11 Fitness measurement as theory embodiment

Before examining how the fitness can be measured, it is important to describe how the theory represented by a model is related to the fitness measure. Similarly to how models should fit data, genomes "fit" the fitness measure. This measure is the formal description of what is expected, and is not in principle excluded from the free parameters criticism. Suppose, for example, that a genetic algorithm would be used to develop a conventional BP-ANN, and the measurement were how well the genomes (cells, layers, connections, factor for the running average, etc.) agree with the empirical data when run. It is possible that the algorithm would deliver, without human intervention, a model similar to the ones developed by Botvinick et al. (2001). It would still be a questionable model.

Just like any scientific method can be abused, genetic algorithms can be used the wrong way. The defense against this, is not asking the algorithm to fit the data directly, but to deliver genomes that fit a theory (e.g., conflict detection, attentional adjustment, etc.) and *afterwards* run the resulting model with inputs similar to the humans experiments, and see if the data is fitted. In our case, the theory is that cognitive contents are represented in brains as groups of synchronizing neurons.

It is too early to know how this methodological strategy will be received by the scientific community, but at least it seems more conservative than choosing parameters by hand, and the grounds for the parameter selection are made explicit in a formal way.

5.12 The accumulated synchronicity table

The fitness measurement was implemented using an "accumulated synchronicity table", or, shortened, accu-synch table. It is designed to require very little computer time, and -in its most simple form- its main data structures are only two, 2-dimensional arrays (tables) 13 variables wide on each dimension . After every iteration of the compute function, the data table increments one of its variables for each cell, indexed by the cell's State and WaveL variables.

For example, suppose that a given cell is in State 3 and its last wavelength was 5. The data table will increment its content by 1, in the row 3, column 5. This will be done with every cell and every iteration, accumulating in the table this information until a fitness needs to be calculated (afterwards the table is erased). When the fitness is evaluated, every position in the data table is multiplied by its respective position on the second, "fitness" table. The fitness criteria are defined by writing multiplication values on them.

Cells having the same State and the same WaveL spiked the last time together, and at the last-but-one time too. The result is, that without synchronization, some positions of the table will accumulate bigger numbers than others, but they would do this at random on a pattern that can predicted from their mean frequency. If synchronized, some positions in the table will grow much faster, forming a column with the top being close to their mean wavelength. The column is formed due to the states changing on each time-step, and the top is the cut-off point because cells do not reach a higher State than their wavelength. In case the synchronization is stochastic (4.5.4), the columns pattern will be fuzzy, but is still different than without synchronization.

Suppose, for example, the table is as follows:

```
const double FIT2[R_P_SPAN][R_P_SPAN] = {
//0  1  2  3  4  5  6  7  8  9 10 11 12 Wavelength
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 0 state
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 1
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 2
{ 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 3
{ 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0}, // 4
{ 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0}, // 5
{ 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0}, // 6
{ 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0}, // 7
{ 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0}, // 8
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0}, // 9
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0}, //10
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, //11
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}};//12
```

If the values of the last State before a Spike and last wavelength are identical for a given cells, they will be stored in the data table in the diagonal running from top left to top down. The fitness table shown here, is multiplied by the data and accumulated for all positions to calculate the fitness measure.

5.13 Simulation procedure

The reference for the range of values of a given parameter to be selected, are taken from 5.3.2. The parameters subjected to the genetic algorithm, were the following:

HL_IO[0]: 15 values, range 12-120
HL_IO[2]: 15 values, range 2-20
HL_IO[4]: 15 values, range 2-20
IO_PO[0]: 15 values, range 0.513158118 to 1.948717100
IO_PO[2]: 15 values, range 0.263331254 to 1.0
IO_PO[4]: 15 values, range -0.263331254 to -1.0
DELAY[2]: 1 to 12
DELAY[4] 1 to current value of DELAY[2]

The ranges were chosen by placing the reference in the middle of the range, and adding seven values bigger and seven smaller. In the case of delay, the reference values were 6 for excitatory synapses, and 2 for inhibitory ones. The excitatory DELAY[2] was chosen at random between 1 and 12. For the inhibitory DELAY[4] the values were constrained to be no bigger than the excitatory delay.

Each genome is tested by running it two screens (a screen: 300 time-steps), the fitness is measured at the end of each screen, and the values added to give an overall fitness. The first screen has input in the cell groups for red and color, the second screen in the groups for green and word-reading (5.3.1).

There is no termination criteria for the program, it simply runs until the user notices little or no improvement in the fitness of new genomes. The program was stopped after fitness stops improving, usually after running a couple of hours. During each run, a text file was written with the genomes produced, in order to track the evolution of the population. Additionally, the best genome was stored (named PARAM00.TXT) in the standard format used by the program.

The rest of the procedure, i.e., which selection method and how the fitness was measured, is not the same for each program run. Each run is equivalent to an experiment with unpredictable results, and the changes made are equivalent to independent variables. This is described separately in the various runs of the algorithm. Of the many runs performed, only four will be describes in detail for reasons of space.

5.14 Results

5.14.1 Preliminary runs

The first version of the genetic algorithm used a truncation selection (5.8), a fitness evaluation table with the same values (e.g., 2.0) on all positions of the diagonal, and zero for the rest of the table. The algorithm converged very rapidly producing a population of identical genomes. The resulting behavior of the model shows very short wavelength for all cells, without synchronization i.e., uncontrolled tetanic spiking.

Two problems were found. The first one, is that truncation selection eliminates fast many genes, namely those that find themselves at the lower third of the fitness ranking. To make a biological analogy, the variability in the genetic pool is rapidly reduced, and the end population consists exclusively of clones. Of course, recombination of two clones produces only another clone, since all genes are identical. In this way, the algorithm stops producing better genomes, no matter how long it runs.

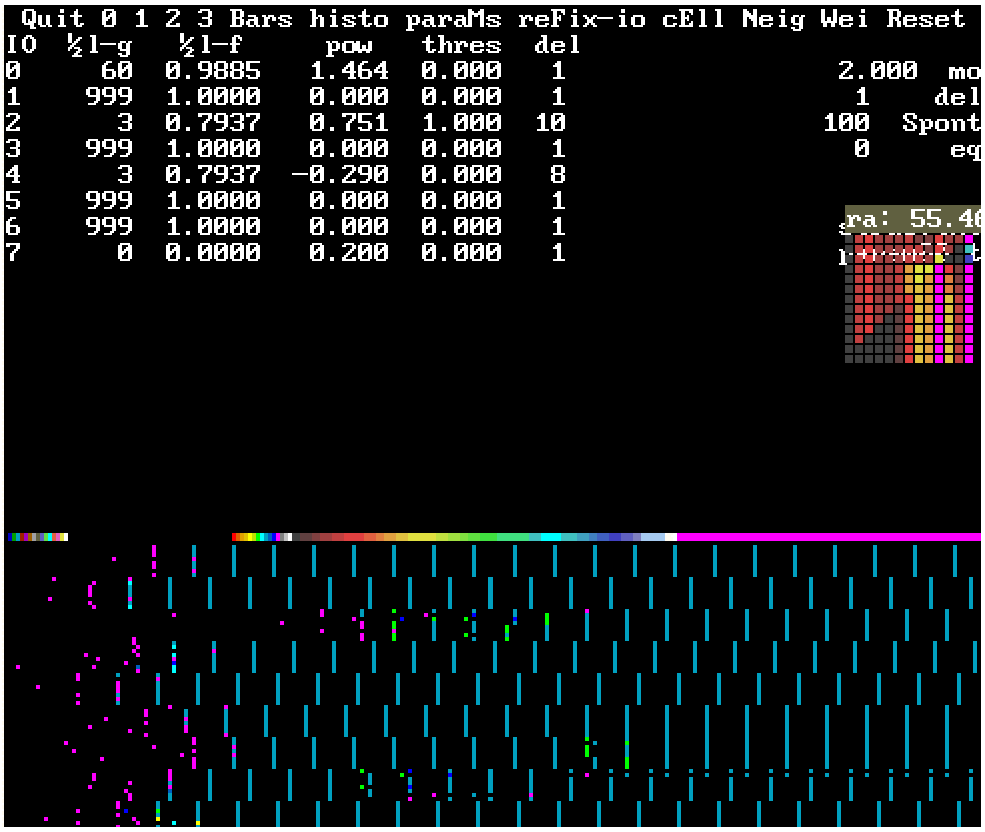
This problem was compounded by the use of the same values in the fitness evaluation table. Suppose that half of the cells is in perfect synchrony at wavelength three, and the other half is synchronized at wavelength nine. The first group's state reaches the diagonal one third of the time-steps, while the second one does so only one ninth of the time steps. When the data table is multiplied by the fitness table, it gives a value to the fast group three times as big as the one of the slow group. This should be no problem because the evaluation is designed to give advantage to short-wavelength synchronization. Suppose now, that none of the cells is actually synchronized, but simply have very similar wavelengths. The probability that fast cells *appear* synchronized is three times as high for cells of wavelength three than for cells of wavelength nine. This is an artifact that gives an undue high fitness to rapid-spiking cells. This artifact is more troublesome when many cells are part of the group being evaluated, and the first simulations computed the fitness value for all 100 cells.

5.14.2 run 0124

To solve the problems mentioned above, the selection procedure was changed to fitness proportional selection (5.9). The second change, is to use values in the diagonal that are identical to the State/wavelength index between four and nine, and smaller or zero at both ends. The fitness table used:

```
const double FIT2[R_P_SPAN][R_P_SPAN] = {
{0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0}, // 0 state
{0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0}, // 1
{0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0}, // 2
{0.0,0.0,0.0,1.5,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0}, // 3
{0.0,0.0,0.0,0.0,4.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0}, // 4
{0.0,0.0,0.0,0.0,0.0,5.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0}, // 5
{0.0,0.0,0.0,0.0,0.0,0.0,6.0,0.0,0.0,0.0,0.0,0.0,0.0}, // 6
{0.0,0.0,0.0,0.0,0.0,0.0,0.0,7.0,0.0,0.0,0.0,0.0,0.0}, // 7
{0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,8.0,0.0,0.0,0.0,0.0}, // 8
{0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,9.0,0.0,0.0,0.0}, // 9
{0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,9.0,0.0,0.0}, //10
{0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,9.0,0.0}, //11
{0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0}};//12
```

This was expected to decrease (or eliminate) the fitness advantage for too fast- and too slow-spiking genomes, and give equal chances to the middle range. The resulting best genome’s behavior:



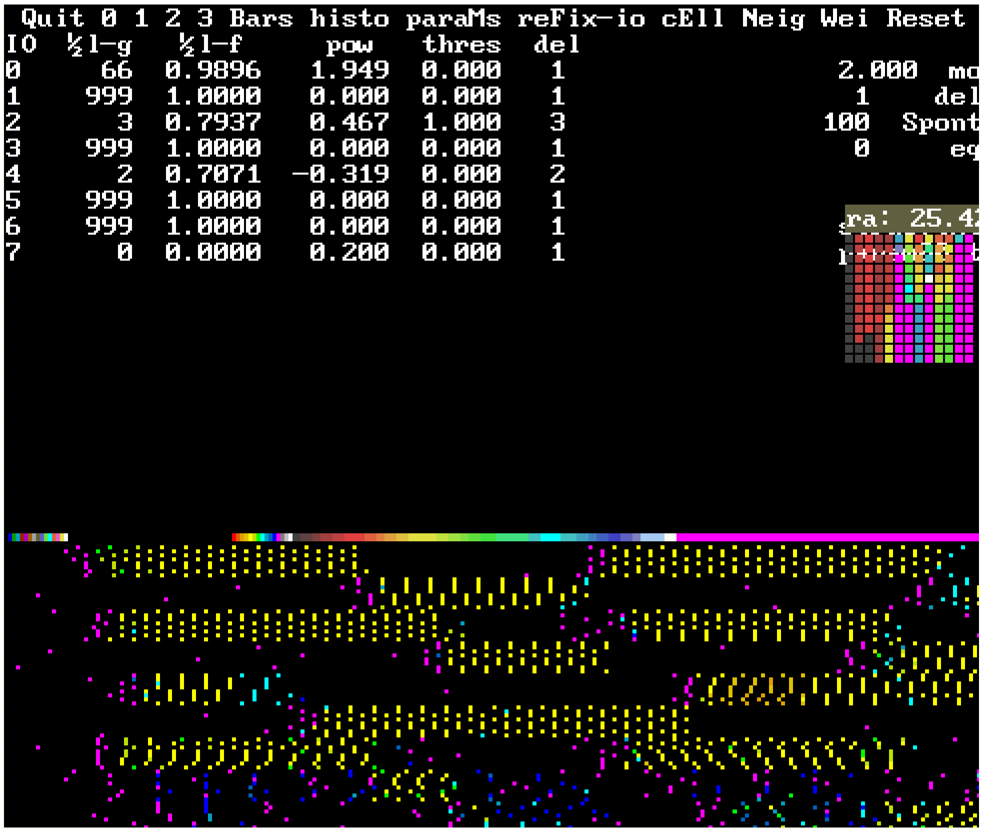
```
 Quit 0 1 2 3 Bars histo paraMs reFix-io cEll Neig Wei Reset
IO  ½l-g   ½l-f      pow    thres  del
0      60  0.9885   1.464   0.000   1                     2.000  mo
1     999  1.0000   0.000   0.000   1                       1     del
2       3  0.7937   0.751   1.000  10                     100   Spont
3     999  1.0000   0.000   0.000   1                       0      eq
4       3  0.7937  -0.290   0.000   8
5     999  1.0000   0.000   0.000   1
6     999  1.0000   0.000   0.000   1                     ra: 55.46
7       0  0.0000   0.200   0.000   1
```

Fig. 18 run 0124

This run shows almost perfect synchronization for every group of cells, but has two deficiencies. First, the wavelength is too close to the maximum (10 time-steps). Second, it is insensitive to changes in stimulation. Once synchronized, it remains essentially unchanged. The model has landed in a position in the solution space that is simply too stable. Examining the genomes themselves, after 3165 new genomes the population consists exclusively of clones, so there is no improvement to be expected by running longer.

5.14.3 run 0128

It seems that most of the improvement in the previous run is due to the fitness evaluation, so this was kept unchanged. The selection procedure was inadequate, so for this run one selection constrain was added. Every time a pair of parent genomes are selected, they are compared. If found to be identical, a new pair is chosen until the constrain is fulfilled. It follows that clones are allowed, but they cannot spread fast in the population.



```
 Quit 0 1 2 3 Bars histo paraMs reFix-io cEll Neig Wei Reset
IO  ½l-g   ½l-f      pow   thres  del
0     66  0.9896   1.949  0.000   1                     2.000  mo
1    999  1.0000   0.000  0.000   1                      1     del
2      3  0.7937   0.467  1.000   3                    100  Spont
3    999  1.0000   0.000  0.000   1                      0      eq
4      2  0.7071  -0.319  0.000   2
5    999  1.0000   0.000  0.000   1
6    999  1.0000   0.000  0.000   1                      ra: 25.4
7      0  0.0000   0.200  0.000   1
```

Fig 19 run 0128

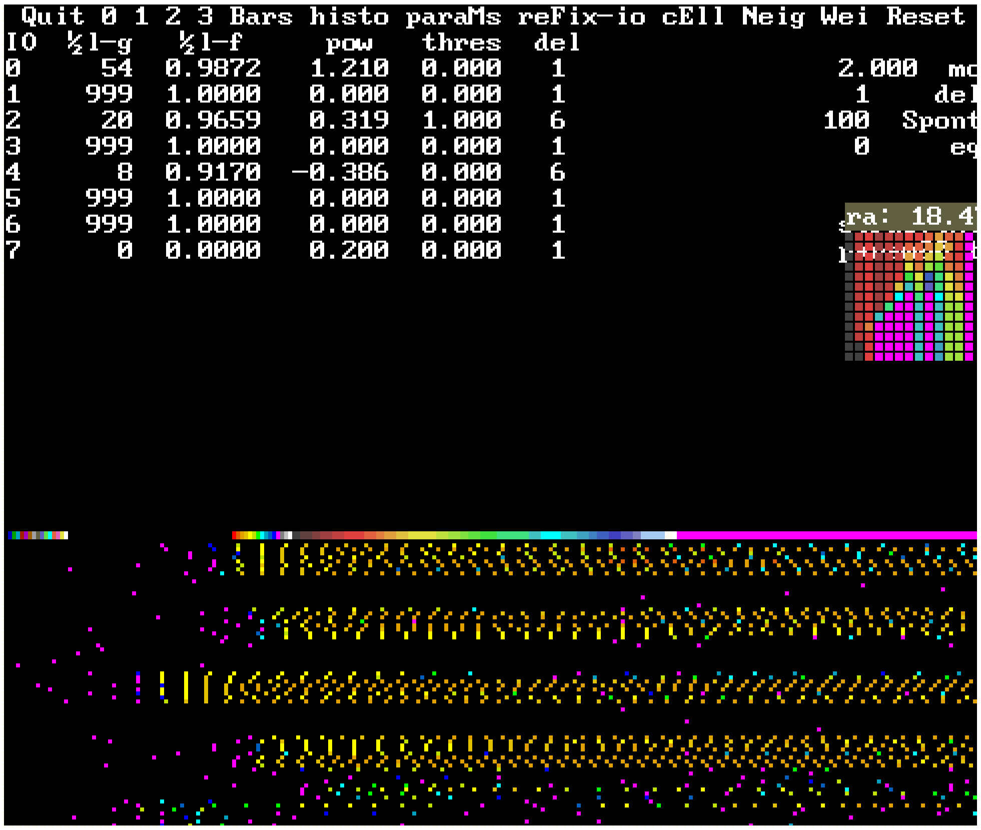
This result is very close to the reference model (5.3.2), though not identical. The main difference, is that though each group is synchronized, half of its cells do it in one phase, and the other half does it shifted by a half-wavelength. More important, the synchronization spreads from one layer to the next more clearly than in the reference model, so the genetic algorithm can be declared a qualified success.

It is difficult to assess if this model is better than the reference, and if so, by how much. The reason is that there is no independent way to measure the performance, given that the fitness measurement itself is causing the difference.

The genomes are not all identical, but various groups of identical genomes are present. The selection procedure still offers room for improvement.

5.14.4 run 0130

One of the deficiencies of the reference model, is that the synchronization is not sensitive to input. Cells spike together for a while, then synchronization is broken and other cell groups take their place. When reset, the input plays a role in deciding which groups start to synchronize, but afterwards the semi-stability of the model takes over. Up to this point, the fitness evaluation has been applied to the whole system, irrespective of input. To explore if the genetic algorithm may improve this situation, The fitness function was modified to produce two values. The cells that according to input should be synchronized -including the task and response groups that receive no input except from previous layers- are added and multiplied by two, the rest is added to the total. This gives the "right" cells double weight in the fitness measurement.



```
 Quit 0 1 2 3 Bars histo paraMs reFix-io cEll Neig Wei Reset
IO  ½l-g   ½l-f      pow    thres   del
0      54  0.9872    1.210  0.000    1                      2.000  mo
1     999  1.0000    0.000  0.000    1                         1    del
2      20  0.9659    0.319  1.000    6                       100  Spont
3     999  1.0000    0.000  0.000    1                         0      eq
4       8  0.9170   -0.386  0.000    6
5     999  1.0000    0.000  0.000    1
6     999  1.0000    0.000  0.000    1                        ra: 18.4
7       0  0.0000    0.200  0.000    1
```

Fig 20 run 0130

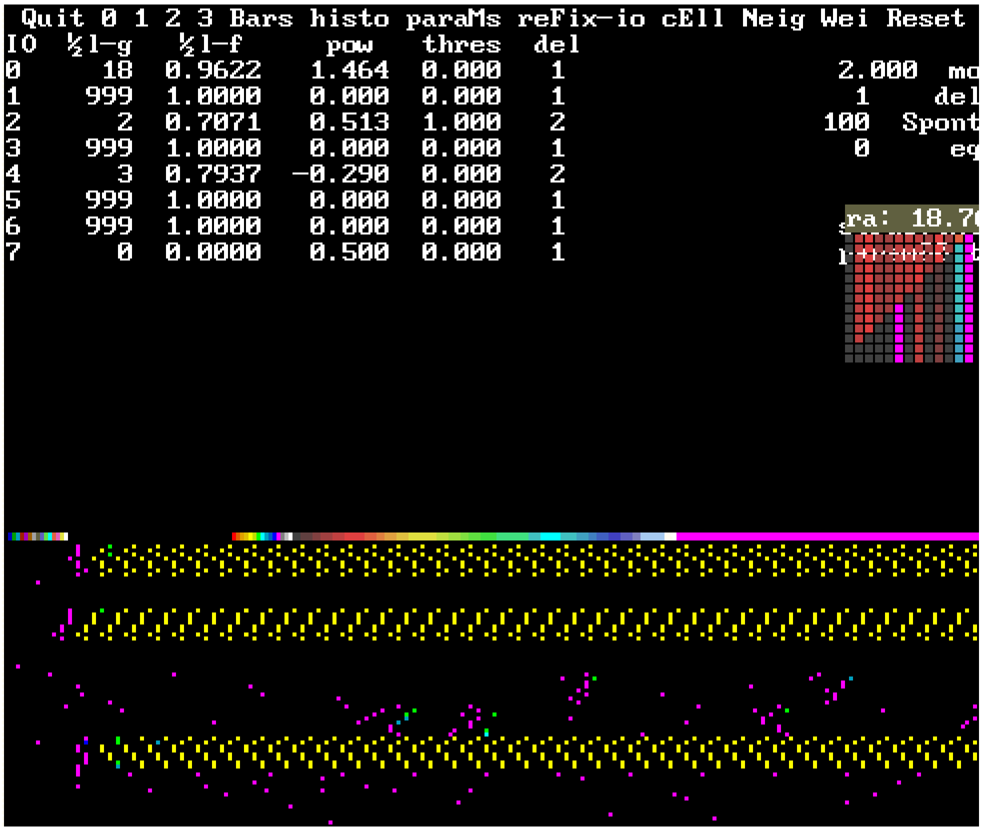
The model seems promising, but only at the beginning. If the simulation is run longer, the appropriate groups' activity does not last. It may be simply the case that in order to generate stable, input-dependant synchronization, the reference model used is inadequate. It is also possible that the genetic algorithm has not been allowed to affect variables that are indispensable for this task, e.g. HL_IO[7] and IO_PO[7], which control the effect of the IO used to deliver input from the world.

5.14.5 run 0218

As stated in 5.11, the way fitness is determined can be viewed as the formalization of a theory. The fitness table allows some flexibility in this respect, which will be illustrated here (only for the sake of a fictional example). When cells get "tired of spiking", their wavelength becomes longer, in the form of a multiple of their previous frequency. When this happens, the cells that are still synchronized produce an harmonic resonance that eventually lets them fire in synchrony with their group. This can be represented by non-zero values in the fitness table beyond the diagonal. Only *bigger* state positions can be used, if not also normal-spiking cells would be affected:

```
const double FIT2[R_P_SPAN][R_P_SPAN] = {
//0  1  2  3  4  5  6  7  8  9 10 11 12 Wavelength
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 0 state
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 1
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 2
{ 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 3
{ 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0}, // 4
{ 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0}, // 5
{ 0, 0, 0, 3, 0, 0, 6, 0, 0, 0, 0, 0, 0}, // 6
{ 0, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0}, // 7
{ 0, 0, 0, 0, 4, 0, 0, 0, 8, 0, 0, 0, 0}, // 8
{ 0, 0, 0, 3, 0, 0, 0, 0, 0, 9, 0, 0, 0}, // 9
{ 0, 0, 0, 0, 0, 5, 0, 0, 0, 0,10, 0, 0}, //10
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,10, 0}, //11
{ 0, 0, 0, 3, 4, 0, 6, 0, 0, 0, 0, 0, 0}};//12
```

```
Another change has been made, in order to eliminate identical genomes.
Each time that a new genome is selected, it is compared with all
existing genomes. If duplicated, one of its genes is chosen at random
and changed at random, i.e., mutated.
```



```
 Quit 0 1 2 3 Bars histo paraMs reFix-io cEll Neig Wei Reset
IO  ½l-g   ½l-f      pow    thres   del
0      18 0.9622   1.464  0.000    1                   2.000   mo
1     999 1.0000   0.000  0.000    1                     1     del
2       2 0.7071   0.513  1.000    2                   100  Spont
3     999 1.0000   0.000  0.000    1                     0      eq
4       3 0.7937  -0.290  0.000    2
5     999 1.0000   0.000  0.000    1
6     999 1.0000   0.000  0.000    1                    ra:  18.7
7       0 0.0000   0.500  0.000    1
```

Fig. 21 run 0218

The model shows no improvement compared with previous ones, except that the genetic variability is greater. By including mutations there is no way to know if running the simulation longer will produce better results. It may be the case that more than one mutation for each individual is needed to measurably increase the fitness.

5.15 Discussion of simulations II

The results of the simulations show that a genetic algorithm can indeed be developed to find parameter values of the kind used in the first series of simulations. Finding *all of them* would require porting the program to parallel hardware, yet a standard desktop computer is already capable of finding parameters in a much bigger solution space than it could be systematically explored by a human (not to speak of the methodological problems this would carry).

Several tasks have to be accomplished before such a program can be run:

1) The groups of cells show good synchronization internally, but frequently at a different phase. This is inevitable, because many groups have no connections between them. One simple way to change this, is to define first synapses at random between all groups. This would be a simple implementation of Hommel's (2008) principle of feature integration.

2) The fitness evaluations table should be further analysed. One of the main advantages of using parallel hardware would be the possibility of testing many different versions of the table. Choosing a table that delivers interesting results is equivalent to choosing among theories, and not between free parameters. The table could be removed from the program, and read from a file in a similar way as the structure and parameters are. In this way, it would even be possible to generate by software many of them, and let the final judgment to a human.

One simple idea that may be worth to test, is a synchronization data table that is incremented only during spikes, and has as indexes the last wavelength and the last-but-one. This would extend the synchronization measurement to three spikes instead of two.

3) All parameters should be available to the genetic algorithm, though this is only possible using much computer resources. The REF_PER table, e.g., was set to constants mainly because it has so many values. Using look-up tables makes the simulation fast, but choosing the table's values would require additional constrains that may prove difficult to realize. In the case of the REF_PER table, one such constrain could be that the values from one point of the re. per. to the next can only change by a given amount, so simply mixing values from different point of the ref. per. would not be possible. One possible solution for this, would be to use pre-written series of values, and letting the selection procedure choose among them.

4) Criteria to automatically terminate a given run are necessary, a task relatively easy to implement.

5) Finally, if the genetic algorithm begins to generate too many unfit new genomes, or if these are too similar to the already existent, the mutation rate can be increased. There is always the possibility that the algorithm becomes trapped in a local maximum, the equivalent of an evolutionary dead-end.

6 SIMULATIONS III

After the development of a genetic algorithm, it is possible to conceive a cycle of development that includes as steps the various methods. As done in Simulations I, possibly relevant features can be identified. The models used the reverse engineering method to reduce the number of free parameters, and the laboratory method to identify the effect of the ref. per. as a possibly relevant feature. Still, many free parameters remained. In simulations II, the genetic algorithm was used to eliminate free parameters. Now, one of the interesting models of the first series (4.5.4) can be used as "positive case" and run through the genetic algorithm. The interesting features will -hopefully- be found again, this time with the rest of the parameters not being free ones.

It may be the case that the previously free parameters are found again, and the genetic variability is low. This would mean that by a extraordinary coincidence, all the right values of parameters chosen by hand were close to optimal, and the theory is restricted to a very specific and narrow range of parameters. Much more probable, is the case of other parameters are found, and/or the genetic variability is high. Statistics can be performed to identify the most common values of a given parameter, or factor analysis may be used to find specific combinations that tend to appear together. Finally, specific models that either unusual (that is, unlike the majority of the population) or typical representatives of a group of similar models, may be subjected to the laboratory method. Of special interest are variables that are parameter values that are very frequent across the population, these may be good candidates for relevant features.

The original "interesting" feature (in our case the form of the ref. per.) should be tested again by the laboratory method, to add (or subtract) credibility to the claim of relevance. The same should be done with other features, as suggested by the statistics. Once this is done, one cycle of development had been completed. Hopefully, new interesting features have been identified, and probably irrelevant ones may be discarded making place for other biological features to be included, and the next iteration may begin.

The next series of simulations should try to come close to completing one cycle of development, by using the genetic algorithm with the model in 4.5.4 as positive case

The first step, is translating the old STRUC.TXT and PARAM.TXT data files to the new format required by the many changes in the simulation program. The files are named STRSTOSY.TXT AND PARSTOSY.TXT. The simulation produced the following result:

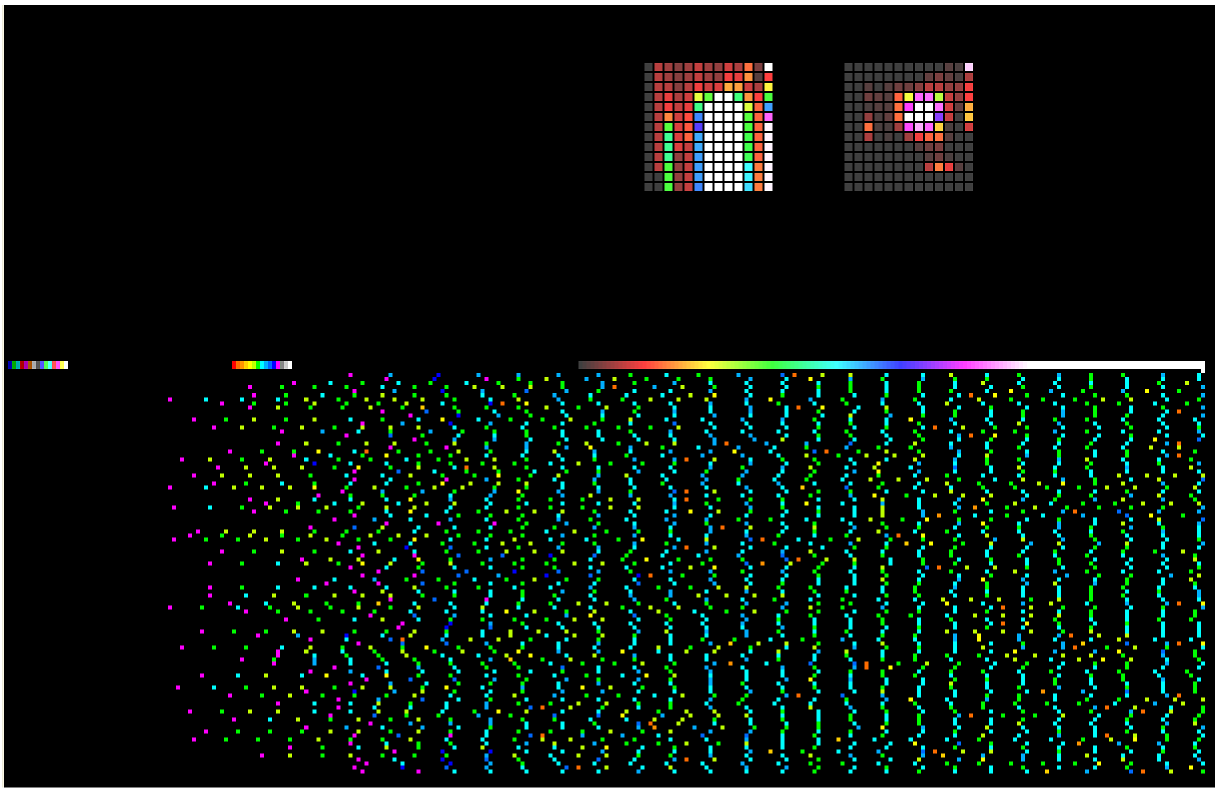


Fig. 22. model 4.5.4 adapted to run under the ION simulation program. The accumulated synchronicity table (5.12) is shown approximately in the middle the screen above the spike display. To the right of it is the new synchronicity table, showing only spiking cells indexed by the present wavelength and the previous wavelength, as proposed 5.15 (point 2, second paragraph).

The model behaves in a virtually identical way to the model presented in 4.5.4. The individual spikes are different mainly because the compiler used for this simulation is a different one, and are not due to the many changes in the structure of the simulation program itself (technically: a different pseudo-random number generator was used, as well as floating-point arithmetic of higher precision). The model is still deterministic, resetting the simulation and running again produces an identical pattern of spikes.

6.1 changes on the genetic algorithm

Beside some minor changes in the graphics, all the changes were performed in the genetic algorithm on source file GENE.C. In model 4.5.4 there are no distinct areas of cell, but rather homogeneous random connections. The fitness measurement was adapted to this change as compared with model 2.18. Two parameters have been added to the solution space, the ref. per. (the independent variable in Simulations I) and cell-wide ref. per. In this way, both the output for a given input, and the influence of the ref. per. in the receptivity to input itself, can interact with each other. This eliminates the cell's ref. per. as free parameter. The form of the function used for the cell's ref. per. has as constrain that the values used have average 0.5, and the function is linear, the only difference being the rate at which the values increase or decrease. If the rate is too high (or low), non-linear flat sections appear at both extremes. A very similar arrangement was used the ref. per. input modulation. See appendix A.7, constant tables `REF_PER_CELL_GEN` and `REF_PER_GEN` for details.

6.2 run 0909

The fitness evaluation was simply adapted to results of accumulated synchronicity of the model 4.5.4 (as shown in the previous figure):

```
>RATE_SYNCH---
  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  9  0  0  0  0  0  0
  0  0  0  0  0  9  9  9  0  0  0  0  0
  0  0  0  0  0  0  9  9  9  0  0  0  0
  0  0  0  0  0  0  0  9  9  9  0  0  0
  0  0  0  0  0  0  0  0  9  9  9  0  0
  0  0  0  0  0  0  0  0  0  9  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0
  -1 0  0  0  0  0  0  0  0  0  0  0  0
```

A small difference with previous forms of fitness evaluation is the use of a negative value in the lower right corner of the fitness table. The intention is to diminish the measured fitness of models whose cells fire continuously.

A final, relatively important change was to limit the value of the synchronicity data to a maximum of 100. The reason for this, is to force the models to distribute their spikes over the whole expected area (the digits '9' on the fitness table). If more than 100 spikes are concentrated in given position, the excess would be wasted for the fitness measurement.

Running the program produced the following results:

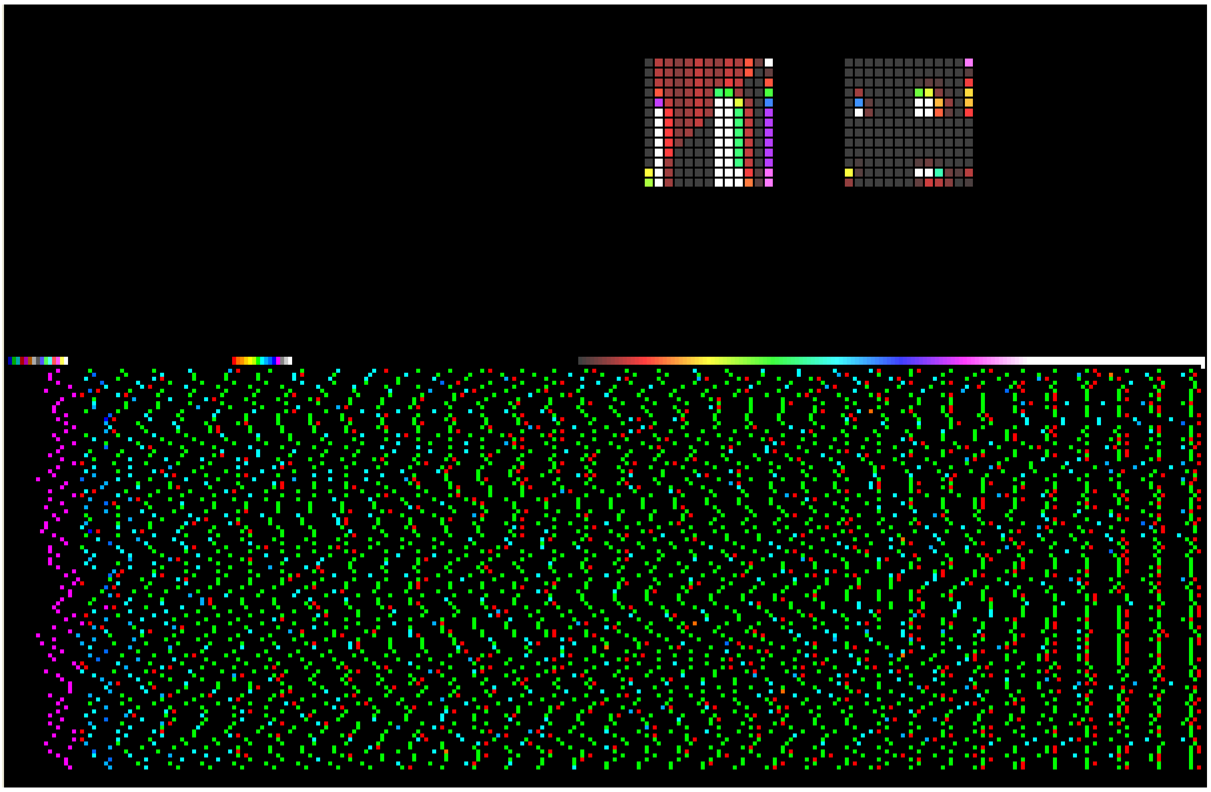


Fig. 23 Run 0909

This result is surprising for two reasons. First, stochastic synchronization is achieved (and remains stable in the form shown at the right of the screen) but on each wave the cells spike *twice*, producing the green/red columns shown. If it were not for this short-long-short-long wavelength pattern, the model would have shown better stochastic synchronization than the positive case used.

The second surprise was found while searching for an error in the program that could have produced this result. The limit of a maximum 100 spikes on a position of the table had been wrongly implemented. Instead of converting values greater than 100 to 100, the operation used was modulo 100 (taking as result the rest of an integer division). So values of e. g., 85, 185, 285 ... 1085, etc were all converted to 85. Despite this defective implementation, the genetic algorithm achieved the expected stochastic synchronization. This shows that despite the difficulties that this method carries, it is actually able to find solutions in parameters combinations that a researcher (the author in this case) could not have thought about.

6.3 run 0923

Finally, the error was corrected, and the limit was taken out of the program and read as a model parameter from the data file. A new fitness evaluation was tested:

```
>RATE_SYNCH---
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1  6  6  7  8  9 -1 -1 -1
 -1 -1 -1 -1 -1  6 -1 -1 -1  8 -1 -1 -1
 -1 -1 -1 -1 -1  5 -1 -1 -1  7 -1 -1 -1
 -1 -1 -1 -1 -1  5 -1 -1 -1  7 -1 -1 -1
 -1 -1 -1 -1 -1  5  5  5  6  6 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

>RATESYNCHCAP-
 370
```

The theory expressed in this fitness evaluation, is that all activity outside the expected one should decrease the fitness (-1 values). The wanted wavelengths
should oscillate between 5 and 9. The values used are adjusted in inverse proportion to the expected spikes at a given frequency (5 in diagonal 5, 9 in diagonal 9). Finally, the cells should oscillate between wavelengths 5 and 9 but do so without having two consecutive equal wavelengths (the rest of the positive values forming a square. The limit for the spikes was set rather arbitrarily to 270, and being a single value it cannot adjust for expected spikes at a given position of the table (it should be in future versions). The results:

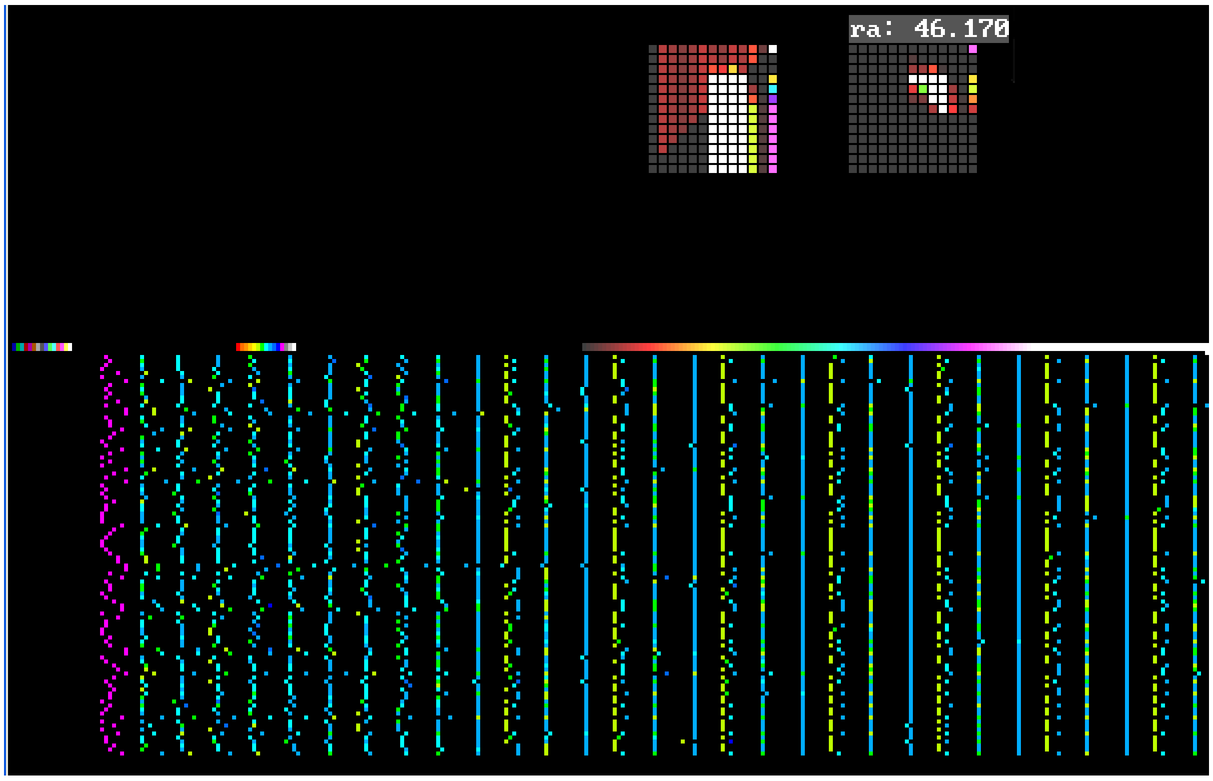


Fig. 24 run 0923

This model synchronizes fast, the individual cells have no stable frequency, and the wave is much less spread than 4.5.4., it can be considered an improvement over the positive case model used as reference.

7 GENERAL DISCUSSION

The use of formal models of cognition is certainly an improvement over non-formal models. The main reason is that the theory implicit in the model, is completely unambiguous and predictions are precise. As a consequence, the results of models are quantitative, which makes it possible to compare them with empirical data.

The use of neural networks as formal models, improves the plausibility because it uses features known to be present in neural systems, like cells and connections with variable weight. However, in the field of cognitive conflict, BP-ANNs are frequently developed to model a single effect, and the models differ from each other (free parameters). A comprehensive theory of cognitive conflict -and the control mechanisms than counteract it- should be able to model many of these effects at the same time.

This work proposes four hypotheses (1.1) concerning this problem:
1) ANN cells should have a internal state to account for sequence and timing effect.
2) The state-dependant input/output can be implemented using known neural features.
3) It is possible to develop such models efficiently using principles from cellular automata.
4) Cognitive contents can be represented as synchronized cells.

This work also proposes four methods to develop better models:
A) The black box method
B) The reverse engineering method
C) Genetic algorithms
D) The laboratory method

That hypothesis 1 is probably true, is shown -without simulations- by using the black box method in 2.1. In short form, the argument claims that sequence and timing must be represented in models, because they are crucial in classical and operant conditioning, as well as in the Gratton effect and negative priming. To put it in everyday words: if a neural network does not recall what happened before, neither the order of events nor their timing can affect the network's behavior. It follows that additionally to long term memory (connections weights) cells need a shorter-term memory, i. e., an internal state that influences their input/output.

Using reverse engineering, the internal state (hypothesis 2) is implemented in Simulations I in the form of synaptic-level storage and decay of input. State-dependant input/output is implemented by using the same device and adding refractory-period dependant input. The laboratory method is used to suggest that the ref. per. is probably one of the neural features relevant for models.

The development of a simulation fast enough to be used in a genetic algorithm (Simulations II and III) suggests that cellular automata (hypothesis 3) are a promising concept for neural network models. Genetic algorithms, together with reverse engineering, are also useful to reduce or eliminate the number of free parameters.

Hypothesis 4 (synchronization) was not proven. There is however extensive neurobiological evidence that suggests it is correct, this makes it possible to treat it more like an assumption. Simulations I (4.5.4) suggest a feedback mechanism that produces stochastic synchronization, and 5.10 why this type of mechanism is difficult to detect with standard statistical methods. Whether previously not described, not present in nature, or simply unknown to the author, the discovery of this synchronization suggest that the methods presented here may be useful in neural modeling.

Finally, the attempt to show how the proposed methods can reinforce each other was performed in Simulations III. Although only partially successful, the interaction of such methods with more traditional ones may be a promising scientific strategy for the future.

Altogether, this work is an attempt to go beyond deploring the unsatisfactory state of cognitive modeling. It tries to pinpoint specific problems of models, in which directions to search for solutions, and non-traditional or overlooked methods to find those solutions. To what measure this attempt has been fruitful, is maybe better left to the ultimate scientific tool (to paraphrase Oscar Wilde), the method that dares not speak its name: intuition.

8 Bibliography

Ahissar, E., Vaadia, E., Ahissar, M., Bergman, H., Arieli, A., & Abeles, M. (1992). Dependence of Cortical Plasticity on Correlated Activity of Single Neurons and on Behavioral Context. *Science*, 257, pp. 1412-1415.

Alkon, D. L. (1983). Learning in a Marine Snail. *Scientific American,* 249, 70-84.

Atkinson, R. C. (1961) The use of models in experimental psychology. In: H. Freudenthal (Ed.), *The concept and the role of the model in mathematics and natural and social sciences*. Dordrecht, Holland,: Reidel, as quoted in: Deppe, W. (1977) *Formale Modelle in der Psychologie*. p. 160 Stuttgart: Kohlhammer.

Barinaga, M.(1990). The Mind Revealed? *Science*, 249, pp. 856-858.

Barnes, D. M. (1986). Brain Architecture: Beyond Genes. *Science, 233,* pp. 155-156.

Beardsley, T. (1997) Debunking the Digital Brain. URL: http://www.sciam.com/explorations/020397brain/020397explorations.htm

Botvinick, M. M., Braver T.S., Barch D.M., Carter C.S., & Cohen J. D. (2001). Conflict monitoring and cognitive control. *Psychological Review*, 108(3) pp. 624-652.

Botvinick, M. (2007) personal communication. At the *Final Meeting of the Priority Program on Executive Functions*. Leiden, Holland.

Braitenberg, V. (1993): Vehikel. *Experimente mit kybernetischen Wesen*. Rowohlt Taschenbuch, Deutschland.

Braitenberg, V., & Shüz, A. (1989). Cortex: hohe Ordnung oder größtmögliches Durcheinander? *Spektrum der Wissenschaft*, May 1989, pp. 74-86.

Braithwaite, R. B. (1963): Models in the Empirical Sciences. In Nagel, A. Suppes, P., & Tarski, A. (Eds.), *Logic, Methodology and the Philosophy of Science.* pp. 224-341. Stanford:Stanford University Press.

Cartwright P, Halgedahl S.L., Hendricks J.R., Jarrard R.D., Marques A.C., Collins, A.G., & Lieberman, B.L. (2007) Exceptionally Preserved Jellyfishes from the Middle Cambrian. *PLoS One* 2(10): e1121. doi:10.1371/journal.pone.0001121

Cohen, J. D. (2008) personal web page. URL: http://www.csbmb.princeton.edu/ncc/jdc.html

Cohen, J.D., Dunbar, K., & McClelland, J.L. (1990). On the Control of automatic processes: A parallel distributed processing account of the Stroop effect. *Psychological Review*, 97, pp. 332-361.

Cohen, J. D., & Huston, T. A. (1994). Progress in the use of interactive models for understanding attention and performance. In Umilta, C., & Moscovitch, M. (Eds.), *Attention and performance XV* pp. 453-476. Cambridge, MA: MIT press.

Dörner, Dietrich (1994) Heuristic der Theorienbildung. In Herrmann, T., & Tack, W. H. (Eds.) *Enzyklopädie der Psychologie*, Themenbereich B Serie I Band 1

Dunn, C. W., Hejnol, A., Matus, D. Q., Pang, K., Browne, W. E., Smith, S. A., Seaver, E., Rouse, G.W., Obst, M., Edgecombe, G. D., Sørensen, M. V., Haddock, S. H. D., Schmidt-Rhaesa, A., Okusu, A., Møbjerg Kristensen, R., Wheeler, W. C., Martindale, W. Q., & Giribet G. (2008). Broad phylogenomic sampling improves resolution of the animal tree of life. *Nature* 452, pp. 745-749.

Engel, A. K., Debener, S., & Kranczioch, C. (2006) Coming to Attention. How the brain decides what to focus conscious attention on. *Scientific American Mind* - August, 2006.

Eriksen, B. A., & Eriksen, C. W. (1974). Effects of noise letters upon the identification of a target letter in a nonsearch task. *Perception and Psychophysics*, 16, pp. 143–149.

Feynman, R. (undated) URL: http://duartes.org/gustavo/blog/post/2008/02/20/Richard-Feynman-Challenger-Disaster-Software-Engineering.aspx

Galileo Galilei (1632), *Dialogue Concerning the Two Chief World Systems*. Florence, Italy: Unknown publisher.

Gratton, G., Coles, M. G. H., & Donchin, E. (1992). Optimizing the use of information: The strategic control of the activation of responses. *Journal of Experimental Psychology: General*, 121, pp. 480-506.

Gray, C. M. Konig, P. Engel, A. K., & Singer, W. (1989). Oscillatory responses in cat visual cortex exhibit inter-columnar synchronization which reflects global stimulus properties. *Nature,* 338, p. 334.

Goodman, C. S., & Bastiani, M. J. (1984). How Embryonic Nerve Cells Recognize One Another. *Scientific American*, 251, pp. 58-66.

Hayes, B. (1984). Computer Recreations. *Scientific American*, 250, 3, pp. 10-16. (auch zu finden im: *Spektrum der Wissenschaft*, June 1984 pp. 6-16).

Hebb, D.O. (1949). *The organization of behavior*. New York: Wiley.

Herzog, W. (1984). *Modell and Theorie in der Psychologie*. Göttingen: Hogrefe.

Hinton, G. E., & Sejnowski, T. J.(1983) Analyzing Cooperative Computation. In: *Proceedings of the 5th Annual Congress of the Cognitive Science Society,* Rochester, NY, May 1983.

Hommel, B., Proctor, R. W., & Vu, K. P. L. (2004) A feature-integration account of sequential effects in the Simon task, *Psychological Research,* 68, pp. 1–17.

Kandel, E.R. Schwartz, J.H. Jessell, T.M., Eds. (2000). *Principles of Neural Science, 4th ed.* New York: McGraw-Hill.

Koch, C. (1997) Computation and the single neuron. *Nature,* 385, pp. 207-210.

Kuhn,T. (1962) *The Structure of Scientific Revolutions*. Chicago: Univertity of Chicago Press.

Luna-Rodriguez, A., Hübner, M., Peters, A., & Kluwe, R. (2003). A Cellular Automata Model of the Origins of Neural Synchronization. In: Detje, F., Dörner, D., & Schaub, H. (Eds.) *Proceedings of the Fifth International Conference on Cognitive Modeling* pp. 159-164. Bamberg: Üniversitäts-Verlag Bamberg.

Markram, H. (2006) The Blue Brain Project. *Nature Neuroscience Review*, 7, pp. 153-160.

McCulloch, W., & Walter Pitts, W. (1943) A Logical Calculus of Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics* 5, pp. 115-133.

Minsky M. L., & Papert S. A.(1969) *Perceptrons.* Cambridge, Mass.: MIT Press.

O'Reilly, R. C., & Munakata, Y. (2000) *Computational explorations in cognitive neuroscience : understanding the mind by simulating the brain*. Cambridge, Mass. : MIT Press.

Pinker, S. (2008) Reverse-Engineering the Psyche. URL: http://www.wired.com/science/discoveries/news/1998/03/10964

Plunkett, K., & Marchman, V. (1993) From rote learning to system building: acquiring verb morphology in children and connectionist nets. *Cognition*, 48, pp. 21-69.

Raffone, A., & Wolters, G. (2001) A Cortical Mechanism for Binding in Visual Working Memory. *Journal of Cognitive Neurocience*, 13, pp. 766 – 785.

Roberts, S., & Pashler, H. (2000) THEORETICAL NOTES How Persuasive Is a Good Fit? A Comment on Theory Testing. *Psychological Review*. 107(2), pp. 358-367

Rosenblatt, F. (1958), The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain, Cornell Aeronautical Laboratory, *Psychological Review*, 65(6), pp. 386-408.

Rumelhart, D.E., McClelland, J. L., & the PDP Research Group (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Volume 1: Foundations, Cambridge, MA: MIT Press

Searle, J. (1983). *Intentionality: An Essay in the Philosophy of Mind*. New York, Cambridge University Press.

Seidenberg, M., & McClelland, J. (1989) Adistributed model of word recognition and naming, *Psychological Review*, 96, 523-568.

Simon, H. A., & Newell, A. (1963): The uses and limitations of models. In: M. H. Marx (Ed.), *Theories in contemporary psychology*. New York, MacMillan.

Singer, W. (2000) interview for the Nordwest Funk. URL: http://www.wz.nrw.de/magazin/artikel.asp?nr=259&ausgabe=2000/1&titel=Entdeckungen%5Esind%5Enicht%5Eplanbar&magname=

Skinner, B. F. (1947) 'Superstition' in the Pigeon. *Journal of Experimental Psychology*, 38, 168-172.

Stachowiak, H. (1973). *Allegemeine Modelltheorie*. pp. 129-133. Wien: Springer-Verlag.

Stroop, J. R. (1935) Studies of interference in serial verbal reactions. *Journal of Experimental Psychology*, 18, pp. 643-662.

Ueckert, Hans. (1983) Kapitel 5: Computer-Simulation p. 587. In: Brädenkamp, J., & Feger, H. (Eds.) *Enzyklopädie der Psychologie*, Themenbereich B Methodologie und Methoden, Serie I Forschungsmethoden der Psychologie, Band 5 Hypothesenprüfung. Göttingen: Hogrefe.

Verguts, T., & Notebaert, W. (2008) Hebbian learning of cognitive control: dealing with specific and nonspecific adaptation. *Psychological Review*, 115, pp. 518-525.

von der Malsburg, C. (1983) How are nervous structures organized? In: *Synergetics of the Brain, Proceeding of the International Symposium on Synergetics*. Basar, E., Flohr, H., Haken, H., & Mandell A. J. (Eds.), Springer, pp. 238-259.

Von Neumann, J. (1948). *The Computer and the Brain*. New Haven and London: Yale University Press.

Vreeken, J. (2002) Spiking neural networks, an introduction, *Technical Report UU-CS-2003-008,* Institute for Information and Computing Sciences, Universiteit Utrecht. URL:
http://www.cs.uu.nl/groups/ADA/people/vreeken/index.php

Walter, G. (1950). An imitation of life. *Scientific American*. 182(5), pp. 42-45.

Wendt, M., Kluwe, R. H., & Peters, A. (2006). Sequential modulations of interference evoked by processing task-irrelevant stimulus features. *Journal of Experimental Psychology: Human Perception and Performance*, 32, 644 - 667.

Zimbardo, P. G., & Gerrig, R. J. (2008). *Psychologie.* München: Pearson Studium.

```
// ion.h first line

#include <stdio.h> //FILE
/*
const int
   //general caracteristics of cellular automata
   N_C      = 100, //total Number of Cells
   MAX_SYN  =  40, //MAXimum number of SYNapses by cell, must be < N_C
   MIO      =   8, //Maximum number of neurotransmiter/Ions types
   R_P_SPAN =  13, //"normal" span of Refractory Period
   //other values used in compile time
   WO_SZ_X  =  20, //size of world, including two-row margins (in squares)
   WO_SZ_Y  =  20,
   WS       =   8, //side-size in pixels of World's squares
   WO_PO_X  = 504, //position of world map, upper-left corner (in pixels)
   WO_PO_Y  = 176, //..also vertical margin for plot-depo()
   WAVES    =   4, //how many "EEG" of groups of cells are to be ploted
   RNDSZ    =   7; //rnd vector's size

due to a bug in gpp, previously declared constants cannot be used
to define the size of arrays, have to use #define
*/

#define N_C      100
#define MAX_SYN   40
#define MIO        8
#define R_P_SPAN  13
#define WO_SZ_X   20
#define WO_SZ_Y   20
#define WS         8
#define SHOW_GEN 250
#define WAVES      4
#define AREAS     12
#define RNDSZ      7

#define MAXRBOWCOLORS 16

//#define NDEBUG
#include <assert.h>

extern const int FX,FY;//font size in pixels, hires.c

typedef unsigned char byte;

//vars of main()
extern const int ON; //used for virtual_syn with constant input
                     //presynaptic axon is firing if == 0

extern int    *Type,              //cell's default neurotransmiter
              *State,             //cell's "clock" (0 == depolarization)
              *Axon,              //delay in transmission of spikes
              *WaveL,             //last wavelength
              *SynNum,            //number of synapses declared
              *Posflag;           //positive flag, catch negative ion amounts
extern float  *TotalIo;           //total input*equa.*power*cell ref.per.
extern float  (*Cell_Io)[MIO],    //total cell's store of neurotrans/ions
              (*SynEq)  [MIO];    //equalizes number of syn. by NT/ion
extern int    (*SynType)[MAX_SYN],//types of declared synapses
              (*SynPos) [MAX_SYN];//neighbor' number, negative:free_syn
extern int   *(*Syn)    [MAX_SYN];//points to neighbor's Axon
extern float  (*SynWei) [MAX_SYN],//weights of synapses
              (*SyIo)   [MAX_SYN];//Synaptic Input/Output, Neurotrans & Ions

//global parameter pseudo-constants user-defined
extern int
```

```
   EegB[WAVES],   //"electroencephalograph" channels
   EegE[WAVES],
   AREA_B[AREAS], //area's begin (first cell)
   AREA_E[AREAS]; //area's end   (last cell)
extern char AREA_NAME[AREAS][20];

extern int RANGE[4][MIO];//x-axis nt/io type
                         //line[0][io-type] IO top of range on noisy reset(1)
                         //line[1][io-type]    bottom
                         //line[2][io-type] WEIGHT top
                         //line[3][io-type]        bottom

extern int
   HL_IOG [MIO],  //Half-Life  of neurotransmitter/ion (generations)
   MASTER_DELAY,  //default delay, value 0 means fire in _next_ generation
   DELAY  [MIO],  //cell's delay by type
   Spontaneous,   //to make cells fire without input
   EQUA;//??

extern double
   HL_IO [MIO],   //Half-Life of NTs & ions (factor)
   NT_TH [MIO],   //NT ThresHold;
   NT_PO [MIO];   //NT POwer

extern double
   MOTOR_TH,                 //MOTOR ThresHold
   FIXDECAY[MIO][R_P_SPAN], //add xor substract to IO stores
                            //depending on type&ref.per.
   REF_PER [MIO][R_P_SPAN], //REFractory PERiod (ion absorption)
   REF_PER_CELL[R_P_SPAN];  //REFractory PERiod (CELL sensitivity)

extern char SEED[81],       //random number generator's seed
            STRUCFIL[81],   //structure file's name
            PARAMFIL[81];   //parameter file's name

extern int  RBOW[MAXRBOWCOLORS], //rainbow, defined in start_graphics()
            NtCol[MIO],          //colors of NT/ions
            WO_COL[8];           //world colors

//program's internal global variables
extern int  RndVec[RNDSZ], //random number generator
            RndIdx;        //gerator's index

extern byte NeiSta[N_C][N_C];        //Neighbor's Status
extern int World[WO_SZ_Y][WO_SZ_X]; //defined in set-World-up
extern int WO_PO_X,WO_PO_Y;

extern double Left,  //runnig average of activity in motor zone
              Right, //idem
              Move;  //idem

//sensor input uses virtual syns to these vars 0:firing else:inactive
extern int North,
           East,
           South,
           West;
                       //frog's retina
extern int Pix[9][4];  //pix[0][ ] is dummy, always black
                       //pix[ ][0] world values
                       //ACHTUNG: 0:firing   else:inactive
                       // first dimension map (looking up/north):
                       //                        1 2 3
                       //                        4 5 6
                       //                        7(f)8
                       //2nd dimension: R,G,B.
```

```
extern long   Gen;       //counter of automata's generations

extern int    Frog_po_x, //default frog's position (X axis)
              Frog_po_y, //idem Y axis
              TimePos,   //X-axis position for depo & EEG
              Interactive_mode,  //1:interactive 0:batch
              SEE;       //show hires on batch

extern char   Direc,     //direction: 'n'orth, 'e'ast, 's'outh, 'w'est
              DispUp;    //display up (paraMs Bars For...)

//************** global to gene.c, used by hires.c ******************

extern long His2D[R_P_SPAN][R_P_SPAN];

extern long His3D[AREAS][R_P_SPAN][R_P_SPAN];

extern long Accumulated;

// *** functions ****************************************************

// *** main.c ***

void alarm(char *msg,char *str_fil, unsigned uLine) ;

int rnd(int range) ;

void reset_generator(void) ;

double half_life(int generations) ;

void hl_factors(void) ;

void setEqu(void) ;

void retina(void) ;

void compass(void) ;

char motor_navigation(char reset) ;

int cold_start(void) ;

//void batch(int see) ; moved to gene.c

void main_command_line_params(int argc,char *argv[]) ;

// *** compute.c ***
int check_cells(void) ;

void compute(void) ;

// *** hires.c ***

int start_graphics(void) ;

void plot_n(int x, int y, int size, int color) ;

void plot_4(int x, int y, int color) ;

void plot_depo(void) ;

void plot_waves(void) ;
void plot_nei(int what) ;

void plot_wei(void) ;

void plot_frog(void) ;

void plot_World(void) ;

void show_synchro_table(void) ; //uses His2D from gene.c

void synchro_stats(void) ; //shows instantaneous, not accumulated

void histogram(int full) ;

void show_LMRG(void) ;

void bars(void) ;

void restore_text_mode(void) ;

// *** files.c ***

void set_World_up(FILE *dta_fil, int load_fil) ;

void reset(int noise) ;

void read_nei(int reads,int readed,int syntype) ;

int wrap(int first, int last, int pos) ;

void loc_nei(int first, int last, int syn_typ,
             int n0, int n1, int n2, int n3) ;

void rnd_nei(int z_b, int z_e, int s_q, int s_t, int n_b, int n_e) ;

void virtual_syn(int first, int last, int syn_typ, float weight, char *target) ;

int load_struc_file(char *struc_file_name) ;

int load_param_file(char *params_file_name) ;

int save_params(int interactive, char *genename) ;

// *** menus.c ***

void clean_line(int first, int last) ;

void menu_line_tx(void) ;

void menu_line_gr(void) ;

double ask_double(char message[],double min,double max,double old) ;

int ask_int(char message[],int min,int max,int old) ;

int change_cell() ;

void show_syn(int cell, int begin, int tempi) ;

void show_cell(int cell) ;

void def_menu_gr(void) ;

void show_table(int x,int y) ;
void show_table_gr(int x,int y) ;

void display_up(void) ;

void change_params(void) ;

void change_ref_per(void) ;

void warm_reset(int noise) ;

long menu_gr(void) ;

// gene.c

char * sex(int woman, int man) ;

void roulette_selection(long count) ;

void bubble_sort(void) ;

void accu_synch(int mode) ;

double rate_accu_synch(int zonesrow) ;

double two_screens(void) ;

void set_params(int which) ;

void seed_orga(int which,
               int hl_iog0, int hl_iog2, int hl_iog4,
               int nt_po0, int nt_po2, int nt_po4,
               int delay2, int delay4,
               double fitness) ;

void seed_popu(void) ;

//void dump_organism(struct organism which_orga, int orga_num) ;

void batch(void) ;


//****************** global constants & macros ****************

#define TEST

#define F __FILE__

#define L __LINE__

#ifdef TEST

   #define ASSE(f,str)                \
      if (f)                          \
         NULL;                        \
      else                            \
         alarm(str,__FILE__,__LINE__)
   #else

      #define ASSE(f,str) NULL

#endif

//good-old EGA colors
#define BLACK         0
#define BLUE          1
```

```
#define GREEN         2
#define CYAN          3
#define RED           4
#define MAGENTA       5
#define BROWN         6
#define LIGHTGRAY     7
#define DARKGRAY      8
#define LIGHTBLUE     9
#define LIGHTGREEN   10
#define LIGHTCYAN    11
#define LIGHTRED     12
#define LIGHTMAGENTA 13
#define YELLOW       14
#define WHITE        15

//ion.h last line
```

```
//compute.c first line

#include <math.h>
#include "ion.h"

int check_cells(void) {
   int c,c2;
   for(c=0; c<N_C; c++) {
      for(c2=0; c2<MIO; c2++)
         if( (Cell_Io[c][c2] < -0.0001) || (Cell_Io[c][c2] > 50.0) )
            return(c);
      for(c2=0; c2<SynNum[c]; c2++)
         if( (SyIo[c][c2] < -0.0001 ) || (SyIo[c][c2] > 50.0) )
            return(c);
      }
   return(-1);
}//check_cells

void compute(void) {
   double tot_inp = 0.0, //total NTs input (formula)
          nt_act[MIO];
   int c,c2,
       cell_typ,         //local cell type
       ref_per,          //local Refractory Period (to use States > R_P_SPAN)
       syn_typ,          //local Synaptic Type
       next_state[N_C],  // defined in formula, synchronize generations
       next_axon[N_C];   // idem

   for(c=0; c<N_C; c++) { //for all cells
      tot_inp = 0.0;
      for(c2=0; c2<MIO; c2++) {
         nt_act[c2] = 0.0;
         Cell_Io[c][c2] *= HL_IO[c2];            //cell's total stores decay
         }
      cell_typ = Type[c];
      ref_per = State[c];
      Posflag[c] = 1;
      TotalIo[c] = 0.0;
      for(c2=0; c2<SynNum[c]; c2++) {
         syn_typ = SynType[c][c2];
         SyIo[c][c2] *= HL_IO[syn_typ];        //syn store decays  ???
         SyIo[c][c2] += FIXDECAY[syn_typ][ref_per]; ///new and improved??
         if(SyIo[c][c2] < 0.0) {
            Posflag[c] = 0;
            Cell_Io[c][syn_typ] -= SyIo[c][c2]; ///???
            SyIo[c][c2] = 0.0;
            }
         Cell_Io[c][syn_typ] += FIXDECAY[syn_typ][ref_per];
         if(*Syn[c][c2] - DELAY[syn_typ] == 0) { //neighbor's axon firing?
            SyIo [c][c2]      += SynWei[c][c2] * REF_PER[syn_typ][ref_per];
            Cell_Io[c][syn_typ] += SynWei[c][c2] * REF_PER[syn_typ][ref_per];
            }
         nt_act[syn_typ] += SyIo[c][c2];
         }
      for(c2=0; c2<MIO; c2++)
         tot_inp += (nt_act[c2] * SynEq[c][c2] * NT_PO[c2]); //pri
      //FORMULA
      TotalIo[c] = tot_inp;
      tot_inp *= REF_PER_CELL[ref_per];
      //cell can't fire if some NT/ion store is negative
      if(tot_inp > NT_TH[cell_typ]) { //&& (Posflag[c]) ) {
         next_state[c] = 0;             //FIRE!
         next_axon[c] = 0; //put spike on axon-delay
         WaveL[c] = State[c];  //update wavelength
         }
      else {
         next_state[c] = State[c] + 1;
         next_axon[c] = Axon[c] + 1;  //?? overflows
         }
      }//for all cells

   for(c=0; c<N_C;  c++) {                          //bug somewhere here???
      State[c] = next_state[c]; //update cell's State
      if(State[c] >= R_P_SPAN) {
         State[c] = R_P_SPAN-1;
         WaveL[c] = R_P_SPAN-1;
//         if((Posflag[c])&&(!rnd(Spontaneous))) { //random firing??
         if((TotalIo[c] > -0.01)&&(!rnd(Spontaneous))) { //random firing??
            State[c] = 0;
            next_axon[c] = 0;
            }
         }
      Axon[c] = next_axon[c];    //idem axon ??
      }
//   if(check_cells() >= 0) {
//      show_cell(check_cells()); ???
//      return(1);
//      }
//   return(0);
}//compute
//compute.c last line
```

```
//gene.c first line

#include "ion.h"
#include <bios.h> //bioskey()
//population, number of organisms
#define POPU 100

FILE *gene_output;

struct organism {
          int hl_iog0,
              hl_iog2,
              hl_iog4,
              nt_po0,
              nt_po2,
              nt_po4,
              delay2,
              delay4;
          double fitness;
          char name[20];
          };

struct organism Orga[POPU+1]; //organisms, Orga[POPU] or...
struct organism Baby;         //...Baby is the newborn

long His2D[R_P_SPAN][R_P_SPAN]; //used by hires.c

long His3D[AREAS][R_P_SPAN][R_P_SPAN]; //this too

long Accumulated = 0;

//int SEE = 0; //to avoid printf in hires

const int HL_IOG0[15] = {12,18,24,30,36,42,48,54,60,66,72,84,96,108,120},
          HL_IOG2[15] = { 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,14,16,18,20},
          HL_IOG4[15] = { 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,14,16,18,20};

// 0.163507991, 0.179858790, 0.197844669, 0.217629136, 0.239392049,
// 0.263331254, 0.289664380, 0.318630818, 0.350493899, 0.385543289,
// 0.424097618, 0.466507380, 0.513158118, 0.564473930, 0.620921323,
// 0.683013455, 0.751314801, 0.826446281, 0.909090909, 1.000000000,
// 1.100000000, 1.210000000, 1.331000000, 1.464100000, 1.610510000,
// 1.771561000, 1.948717100, 2.143588810, 2.357947691, 2.593742460,
// 2.853116706, 3.138428377, 3.452271214, 3.797498336, 4.177248169,
// 4.594972986, 5.054470285, 5.559917313, 6.115909045,    *= 1.1

const double NT_PO0[15] = {0.513158118,
                           0.564473930,
                           0.620921323,
                           0.683013455,
                           0.751314801,
                           0.826446281,
                           0.909090909,
                           1.000000000,
                           1.100000000,
                           1.210000000,
                           1.331000000,
                           1.464100000,
                           1.610510000,
                           1.771561000,
                           1.948717100},

             NT_PO2[15] = {0.263331254,
                           0.289664380,
                           0.318630818,
                           0.350493899,
                           0.385543289,
                           0.424097618,
                           0.466507380,
                           0.513158118,
                           0.564473930,
                           0.620921323,
                           0.683013455,
                           0.751314801,
                           0.826446281,
                           0.909090909,
                           1.000000000},

             NT_PO4[15] = {-0.263331254,
                           -0.289664380,
                           -0.318630818,
                           -0.350493899,
                           -0.385543289,
                           -0.424097618,
                           -0.466507380,
                           -0.513158118,
                           -0.564473930,
                           -0.620921323,
                           -0.683013455,
                           -0.751314801,
                           -0.826446281,
                           -0.909090909,
                           -1.000000000};

//********************** variables global to gene.c ******************

void init_orga(void) {
int c;
   for(c=0; c<=POPU; c++) { //include newborn one
      Orga[c].hl_iog0 = 0;
      Orga[c].hl_iog2 = 0;
      Orga[c].hl_iog4 = 0;
      Orga[c].nt_po0  = 0;
      Orga[c].nt_po2  = 0;
      Orga[c].nt_po4  = 0;
      Orga[c].delay2  = 0;
      Orga[c].delay4  = 0;
      Orga[c].fitness = 0;
      strcpy(Orga[c].name,"0123456789012345678\0");
      }
}//init_orga


void baptice(struct organism *doe) {
//must be longer than range of parameters
const char *cODE = "0123456789abcdef\0";
   doe->name[0] = cODE[doe->hl_iog0];
   doe->name[1] = cODE[doe->hl_iog2];
   doe->name[2] = cODE[doe->hl_iog4];
   doe->name[3] = cODE[doe->nt_po0];
   doe->name[4] = cODE[doe->nt_po2];
   doe->name[5] = cODE[doe->nt_po4];
   doe->name[6] = doe->delay2 + 'A';
   doe->name[7] = doe->delay4 + 'A';
   doe->name[8] = '\0';
}//baptice


void set_params(int which) {
   HL_IOG[0] = HL_IOG0[Orga[which].hl_iog0];
   HL_IOG[2] = HL_IOG2[Orga[which].hl_iog2];
   HL_IOG[4] = HL_IOG4[Orga[which].hl_iog4];
   NT_PO[0]  = NT_PO0[Orga[which].nt_po0];
   NT_PO[2]  = NT_PO2[Orga[which].nt_po2];
   NT_PO[4]  = NT_PO4[Orga[which].nt_po4];
   DELAY[2]  = Orga[which].delay2;
   DELAY[4]  = Orga[which].delay4;
   Orga[which].fitness = 0.0;
}//set_params


void seed_orga(int which,
               int hl_iog0, int hl_iog2, int hl_iog4,
               int nt_po0, int nt_po2, int nt_po4,
               int delay2, int delay4,
               double fitness) {
   Orga[which].hl_iog0 = hl_iog0;
   Orga[which].hl_iog2 = hl_iog2;
   Orga[which].hl_iog4 = hl_iog4;
   Orga[which].nt_po0  = nt_po0;
   Orga[which].nt_po2  = nt_po2;
   Orga[which].nt_po4  = nt_po4;
   Orga[which].delay2  = delay2;
   Orga[which].delay4  = delay4;
   set_params(which);
   Orga[which].fitness = fitness;
   baptice(&Orga[which]);
}//seed_orga


void copy_orga(int target, int source) {
   Orga[target].hl_iog0 = Orga[source].hl_iog0;
   Orga[target].hl_iog2 = Orga[source].hl_iog2;
   Orga[target].hl_iog4 = Orga[source].hl_iog4;
   Orga[target].nt_po0  = Orga[source].nt_po0;
   Orga[target].nt_po2  = Orga[source].nt_po2;
   Orga[target].nt_po4  = Orga[source].nt_po4;
   Orga[target].delay2  = Orga[source].delay2;
   Orga[target].delay4  = Orga[source].delay4;
   Orga[target].fitness = Orga[source].fitness;
   strcpy(Orga[target].name,Orga[source].name);
}//copy_orga


void dump_orga(struct organism which_orga, int orga_num) {
char line[256] =
"---------1---------2---------3---------4---------5---------6---------7\0";

   sprintf(line,"\n%3d: %2d %2d %2d %2d %2d %2d %2d %2d %10.4f %s",
           orga_num,
           which_orga.hl_iog0,
           which_orga.hl_iog2,
           which_orga.hl_iog4,
           which_orga.nt_po0,
           which_orga.nt_po2,
           which_orga.nt_po4,
           which_orga.delay2,
           which_orga.delay4,
           which_orga.fitness,
           which_orga.name);
   fprintf(gene_output,line);
   if(!SEE) printf("%s",line);
}//dump_orga
```

```
void dump_popu(void) {
int c;
   for(c=0; c<POPU; c++)
      dump_orga(Orga[c],c);
   fprintf(gene_output,"\n\n");
   if(!SEE) printf("\n");
}//dump_popu


int clone_test(struct organism *doe) {
int c;
   for(c=0; c<POPU; c++)
      if(strcmp(Orga[c].name,doe->name) == 0)
         return(1);
   return(0);
}//clone_test


void mutate(struct organism *doe) {
   switch(random() % 7) {
      case 0: doe->hl_iog0 = random() % 15; break;
      case 1: doe->hl_iog2 = random() % 15; break;
      case 2: doe->hl_iog4 = random() % 15; break;
      case 3: doe->nt_po0  = random() % 15; break;
      case 4: doe->nt_po2  = random() % 15; break;
      case 5: doe->nt_po4  = random() % 15; break;
      case 6: doe->delay2  = (random() % (R_P_SPAN-1)) + 1;
              doe->delay4  = (random() % doe->delay2) + 1;
              break;
      }
}//mutate


void seed_popu(void) {
int c;
   for(c=0; c<POPU; c++) {
      Orga[c].hl_iog0 = random() % 15;
      Orga[c].hl_iog2 = random() % 15;
      Orga[c].hl_iog4 = random() % 15;
      Orga[c].nt_po0  = random() % 15;
      Orga[c].nt_po2  = random() % 15;
      Orga[c].nt_po4  = random() % 15;
      Orga[c].delay2  = (random() % (R_P_SPAN-1)) + 1;
      Orga[c].delay4  = (random() % Orga[c].delay2) + 1;
      Orga[c].fitness = 0.0;
      baptice(&Orga[c]);
      }
}//seed_popu


char * sex(int woman, int man) {
//positive==looks like mami, neg.like papi 0==perfect mix
char *mix = "??????????????????\0";
int looks_like = 42;
   while((looks_like < -3) || (looks_like > 3)) { //force 2 swaps
      looks_like = 0;
      if(random()%2) { Baby.hl_iog0 = Orga[woman].hl_iog0;
         mix[0]='+'; ++looks_like; }
      else {           Baby.hl_iog0 = Orga[  man].hl_iog0;
         mix[0]='-'; --looks_like; }

      if(random()%2) { Baby.hl_iog2 = Orga[woman].hl_iog2;
         mix[1]='+'; ++looks_like; }
      else {           Baby.hl_iog2 = Orga[  man].hl_iog2;
         mix[1]='-'; --looks_like; }
      if(random()%2) { Baby.hl_iog4 = Orga[woman].hl_iog4;
         mix[2]='+'; ++looks_like; }
      else {           Baby.hl_iog4 = Orga[  man].hl_iog4;
         mix[2]='-'; --looks_like; }

      if(random()%2) { Baby.nt_po0 = Orga[woman].nt_po0;
         mix[3]='+'; ++looks_like; }
      else {           Baby.nt_po0 = Orga[  man].nt_po0;
         mix[3]='-'; --looks_like; }

      if(random()%2) { Baby.nt_po2 = Orga[woman].nt_po2;
         mix[4]='+'; ++looks_like; }
      else {           Baby.nt_po2 = Orga[  man].nt_po2;
         mix[4]='-'; --looks_like; }

      if(random()%2) { Baby.nt_po4 = Orga[woman].nt_po4;
         mix[5]='+'; ++looks_like; }
      else {           Baby.nt_po4 = Orga[  man].nt_po4;
         mix[5]='-'; --looks_like; }

      if(random()%2) {
         Baby.delay2 = Orga[woman].delay2;
         Baby.delay4 = Orga[woman].delay4;
         mix[6]='+';
         mix[7]='=';
         ++looks_like;
         }
      else {
         Baby.delay2 = Orga[  man].delay2;
         Baby.delay4 = Orga[  man].delay4;
         mix[6]='-';
         mix[7]='=';
         --looks_like;
         }
      }
   mix[8]='\0';

   Baby.fitness = 0.0; //There's no future, no future for you.

   baptice(&Baby);

   return(mix);

}//sex


void roulette_selection(long count) {
long acfi[POPU], //accumulated fitness array
     accufit = 0,
     ball = 0,
     mutations = 0;
int mother = 0,
    father = 0,
    c;
   if(count >= 1234567891) {
      printf("\nstuck after 1234567891 babys, press any key");
      getch();
      }
   for(c=0; c<POPU; c++) {
      accufit += (long) (Orga[c].fitness*100.0);
      acfi[c] = accufit;
      }
   //choose father and mother with different genomes
   while(strcmp(Orga[father].name,Orga[mother].name) == 0) {
      ball = random() % accufit;
      for(c=0; c<POPU; c++)
         if(ball <= acfi[c]) {
            mother = c;
            break;
            }
      ball = random() % accufit;
      for(c=0; c<POPU; c++)
         if(ball <= acfi[c]) {
            father = c;
            break;
            }
      }
/*
   fprintf(gene_output,"\nmami");
   dump_orga(Orga[mother],mother);
   fprintf(gene_output,"\npapi");
   dump_orga(Orga[father],father);
   fprintf(gene_output,
      "\n                                        %s mix",sex(mother,father));
*/
   sex(mother,father);
   while(clone_test(&Baby)) {
      mutate(&Baby);
      baptice(&Baby);
      ++mutations;
      }

   reset(0);
   HL_IOG[0] = HL_IOG0[Baby.hl_iog0];
   HL_IOG[2] = HL_IOG2[Baby.hl_iog2];
   HL_IOG[4] = HL_IOG4[Baby.hl_iog4];
   NT_PO[0]  = NT_PO0[Baby.nt_po0];
   NT_PO[2]  = NT_PO2[Baby.nt_po2];
   NT_PO[4]  = NT_PO4[Baby.nt_po4];
   DELAY[2]  = Baby.delay2;
   DELAY[4]  = Baby.delay4;
   Baby.fitness = two_screens();

   dump_orga(Baby,-1);
   fprintf(gene_output,"  muta:%3d",mutations);
   if(!SEE) printf("  muta:%3d",mutations);

   c=POPU-1;
   while(c>0) { //sort baby in population
      if(Baby.fitness < Orga[c-1].fitness) break;
      Orga[c]=Orga[c-1];
      --c;
      }
   if(Baby.fitness >= Orga[c].fitness) {
      Orga[c]=Baby;
      fprintf(gene_output," :) ");
      if(!SEE) printf(" :) ");
      }
   else {
      fprintf(gene_output," X( ");
      if(!SEE) printf(" X( ");
      }

}//roulette_selection


void bubble_sort(void) { //what a shame...
struct organism orgacopy[POPU];
int order[POPU];
int swaped = 1,temp,c;
   for(c=0; c<POPU; c++) order[c] = c;
   while(swaped) {
      swaped = 0;
      for(c=0; c<POPU-1; c++)
         if(Orga[order[c]].fitness < Orga[order[c+1]].fitness) {
           temp = order[c];
           order[c] = order[c+1];
           order[c+1] = temp;
           swaped = 1;
           }
      }
   for(c=0; c<POPU; c++)
      orgacopy[c] = Orga[order[c]];
   for(c=0; c<POPU; c++)
      Orga[c] = orgacopy[c];
}//bubble_sort


void accu_synch(int mode) { //accumulated synchronization measure
int c,c2,c3;               //mode:0 clean mode:1accumulate
   if(mode == 0) {
      for(c=0; c<AREAS; c++)
         for(c2=0; c2<R_P_SPAN; c2++)
            for(c3=0; c3<R_P_SPAN; c3++)
               His3D[c][c2][c3] = 0;
      for(c=0; c<R_P_SPAN; c++)
         for(c2=0; c2<R_P_SPAN; c2++)
            His2D[c][c2] = 0;
      Accumulated = 0;
      }
   else {
      for(c=0; c<AREAS; c++)
         for(c2=AREA_B[c]; c2<AREA_E[c]; c2++)
            His3D[c][State[c2]][WaveL[c2]]++;
      for(c=0; c<N_C; c++)
         His2D[State[c]][WaveL[c]]++;
      Accumulated++;
      }
}//accu_synch


double rate_accu_synch(int zonesrow) {
const double FIT2[R_P_SPAN][R_P_SPAN] = {
//0  1  2  3  4  5  6  7  8  9 10 11 12 Wavelenght
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 0 state

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 1

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 2

{ 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 3

{ 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0}, // 4

{ 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0}, // 5

{ 0, 0, 0, 3, 0, 0, 6, 0, 0, 0, 0, 0, 0}, // 6

{ 0, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0}, // 7

{ 0, 0, 0, 0, 4, 0, 0, 0, 8, 0, 0, 0, 0}, // 8

{ 0, 0, 0, 3, 0, 0, 0, 0, 0, 9, 0, 0, 0}, // 9
{ 0, 0, 0, 0, 0, 5, 0, 0, 0, 0,10, 0, 0}, //10

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,10, 0}, //11

{ 0, 0, 0, 3, 4, 0, 6, 0, 0, 0, 0, 0, 0}};//12

int zones[3][12] = {{1,1,1,1,1,1,1,1,1,1,1,1},
                    {1,0,1,0,0,0,1,0,0,0,1,0}, //color task, red input
                    {0,0,0,1,0,1,0,0,0,1,0,1}};//word task, green input
double his_add[R_P_SPAN][R_P_SPAN],
       his_sub[R_P_SPAN][R_P_SPAN],
       rate = 0.0;
int //big1sta = 12, big1wav = 12,
    //big2sta = 12, big2wav = 12,
    //big3sta = 12, big3wav = 12,
    add_cells = 0,
    sub_cells = 0,
    c,c2,c3;

   for(c=0; c<R_P_SPAN; c++)
      for(c2=0; c2<R_P_SPAN; c2++) {
         his_add[c][c2] = 0.0;
         his_sub[c][c2] = 0.0;
         }

   for(c=0; c<AREAS; c++)
      if(zones[zonesrow][c]) {
         for(c2=0; c2<R_P_SPAN; c2++)
            for(c3=0; c3<R_P_SPAN; c3++)
               his_add[c2][c3] += His3D[c][c2][c3];
         add_cells += AREA_E[c] - AREA_B[c] + 1;
         }
      else {
         for(c2=0; c2<R_P_SPAN; c2++)
            for(c3=0; c3<R_P_SPAN; c3++)
               his_sub[c2][c3] += His3D[c][c2][c3];
         sub_cells += AREA_E[c] - AREA_B[c] + 1;
         }

//   if(add_cells)
//      for(c=0; c<R_P_SPAN; c++)
//         for(c2=0; c2<R_P_SPAN; c2++)
//            his_add[c][c2] = his_add[c][c2] / add_cells;
//   if(sub_cells)
//      for(c=0; c<R_P_SPAN; c++)
//         for(c2=0; c2<R_P_SPAN; c2++)
//            his_add[c][c2] = his_add[c][c2] - (his_sub[c][c2] / sub_cells);
   rate = 0.0;
//   for(c=0; c<R_P_SPAN-1; c++)
//      for(c2=0; c2<R_P_SPAN-1; c2++)
//         rate += His2D[c][c2]*FIT2[c][c2];
   for(c=0; c<R_P_SPAN-1; c++)
      for(c2=0; c2<R_P_SPAN-1; c2++) {
         his_add[c][c2] -= his_sub[c][c2];
         his_add[c][c2] *= FIT2[c][c2];
         rate += his_add[c][c2];
         }
   if(rate<0.0) rate *= -0.01;//roulette needs positive numbers
   return(rate / Accumulated);
}//rate_accu_synch


double two_screens(void) {
double run = 0.0;
int c;
```

```
   reset(0);
   hl_factors();                 //transform half-lifes to factors
   if(EQUA) setEqu();
   if(SEE) plot_World();                 //show frog's world
   if(SEE) show_LMRG();

   accu_synch(0);
   for(c=0; c<SHOW_GEN; c++) { //first screen
      retina();
      compass();
      compute();
      accu_synch(1);
      if(check_cells() != -1) return(0.0);
      Gen++;
      motor_navigation(' ');  //unused return
      ++TimePos;
      if(SEE) plot_depo();
      }
   run = rate_accu_synch(1);

   if(SEE) {
      if(DispUp == 'B') bars();
      if(DispUp == 'H') {histogram(1); synchro_stats(); }
      }

   accu_synch(0);
   TimePos = 0;
   Frog_po_y = 10;
   plot_World();
   for(c=0; c<SHOW_GEN; c++) {   //second screen
      retina();
      compass();
      compute();
      accu_synch(1);
      if(check_cells() != -1) return(0.0);
      Gen++;
      motor_navigation(' ');  //unused return
      ++TimePos;
      if(SEE) plot_depo();
      }
   run += rate_accu_synch(2);

   if(SEE) {
      if(DispUp == 'B') bars();
      if(DispUp == 'H') {histogram(1); synchro_stats(); }
      }
   return(run / 2.0);
}//two_screens


void batch(void) { //if SEE==1:show depo,etc. 0:run blind, print text
long c;
int c2;
   reset(0);
   if( (gene_output = fopen("geneout.txt","w+")) == NULL) {
         clean_line(1,1);
         printf("can't save file geneout.txt");
         delay(2000);
         exit(1);
         }
   if(SEE) if(start_graphics()) exit(1);

   init_orga();

   seed_popu();
   for(c=0; c<POPU; c++) {
      reset(0);
      set_params(c);
      Orga[c].fitness = two_screens();
      if(!SEE) printf("%6D \n",c);
      }

   bubble_sort();

   dump_popu();

   srand48(1234567891L);//limits.h: #define LONG_MAX 2147483647L
   for(c=0; c<10000; c++) {
      roulette_selection(c);
      fprintf(gene_output," babys:%6D ",c);
      printf(" babys:%6D ",c);
      if(!SEE) if(c % 100 == 0) dump_popu();
      if(bioskey(1)) {
         getch();
         break;
         }
      }
   fprintf(gene_output,"\n\n after %D babys \n\n",c);
   dump_popu();

   fclose(gene_output);
   reset(0);
   hl_factors();
   if(EQUA) setEqu();
   set_params(0);
   save_params(0,"param00.txt");
   fclose(gene_output);
   exit(0);
}//batch
```

```
// ion_main.c first line
//  written by Aquiles Luna-Rodr¡guez
//  email: luna@unibw-hamburg.de
//  compiles on DJGPP with GRX graphics

#include <conio.h>    //getch gotoxy kbhit color-names

#include "ion.h"

const int ON = 0;

//cell's components
int         *Type,
            *State,
            *Axon,
            *WaveL,
            *SynNum,
            *Posflag;
float       *TotalIo;
float       (*Cell_Io)[MIO],
            (*SynEq)  [MIO];
int         (*SynType)[MAX_SYN],
            (*SynPos) [MAX_SYN];
int        *(*Syn)    [MAX_SYN];
float       (*SynWei) [MAX_SYN],
            (*SyIo)   [MAX_SYN];

//global parameters, user-defined
//these are only dummy values, definition is in params.bck and struc.bck
int EegB[WAVES] ={  0, 0, 0, 0},
    EegE[WAVES] ={ -1,-1,-1,-1};
int AREA_B[AREAS] = {0,0,0,0,0,0,0,0,0,0,0,0},
    AREA_E[AREAS] = {0,0,0,0,0,0,0,0,0,0,0,0};
char AREA_NAME[AREAS][20]={"00-----------------\0",
                           "01-----------------\0",
                           "02-----------------\0",
                           "03-----------------\0",
                           "04-----------------\0",
                           "05-----------------\0",
                           "06-----------------\0",
                           "07-----------------\0",
                           "08-----------------\0",
                           "09-----------------\0",
                           "10-----------------\0",
                           "11-----------------\0"};
int RANGE[4][MIO];

int    HL_IOG [ MIO ],
       MASTER_DELAY = 0, //minimum, 0 fires next generation
       DELAY  [MIO] = { 0,0,0,0,0,0,0,0},
       Spontaneous = 250,
       EQUA = 0;
double HL_IO [ MIO ],
       NT_TH [ MIO ],
       NT_PO[MIO];
double MOTOR_TH = 0.0,
       FIXDECAY[MIO][R_P_SPAN],
       REF_PER[MIO][R_P_SPAN],
       REF_PER_CELL[R_P_SPAN];

char SEED[81] = "0123456789\0",
     STRUCFIL[81] = "12345678.123\0",
     PARAMFIL[81] = "12345678.123\0";
int  RBOW[MAXRBOWCOLORS],//defined in start_graphics()
     NtCol[MIO] = {15,15,15,15,15,15,15,15},
     WO_COL[8] = {BLACK,LIGHTRED,LIGHTGREEN,LIGHTBLUE,
                  YELLOW,CYAN,LIGHTMAGENTA,WHITE};

//program's internal global variables
int  RndVec[RNDSZ]= {1,1,1,1,1,1,1},
     RndIdx = 0;
byte NeiSta[N_C][N_C];
int World[WO_SZ_Y][WO_SZ_X];
int WO_PO_X = SHOW_GEN*2+4,
    WO_PO_Y = 400; //magic??
double Left        = 0.0,
       Right       = 0.0,
       Move        = 0.0;

int    North  = 1,
       East   = 1,
       South  = 1,
       West   = 1;

int Pix[9][4] = {{0,0,0,0}, //ACHTUNG: 0:firing else:inactive
                 {0,1,1,1},
                 {0,1,1,1},
                 {0,1,1,1},
                 {0,1,1,1},
                 {0,1,1,1},
                 {0,1,1,1},
                 {0,1,1,1},
                 {0,1,1,1}};

long   Gen              = 0;
int    Frog_po_x        = 9,
       Frog_po_y        = 9,
       TimePos          = -1,
       Interactive_mode = 1,
       SEE              = 1;
char   Direc            = 'n',
       DispUp           = 'H';

//========================== functions ============================

void alarm(char *msg,char *str_fil, unsigned uLine) {
   fflush(NULL);
   if(str_fil!=NULL)
      fprintf(stderr,"\n %s %s, line %u \n",msg,str_fil, uLine);
   else fprintf(stderr,"\n %s\n",msg);
   fflush(stderr);
   while(getch()!='q') ;
}//alarm

//**********************************************
int rnd(int range) { //returns integer random numbers between 0 and range-1
   const int pRIME = 4001; //pRIME and RNDVEC *must* be prime numbers
   long templ = 0;
   int c;
   if(range <= 0)
      return(-1); //division by 0??
   if (++RndIdx >= RNDSZ) RndIdx = 0;
   for(c=0; c<RNDSZ; c++)
      templ += RndVec[c];
   templ %= pRIME;
   RndVec[RndIdx] = (int)templ;
   for(c=0; c<RNDSZ; c++)
      assert(RndVec[c] >= 0 && RndVec[c] < pRIME);
   assert(RndIdx >= 0 && RndIdx < RNDSZ);
//   return( (int) (templ * range / pRIME) );
```

```
   return( (int) (templ % range) );
}//rnd
//**********************************************
void reset_generator(void) {
long key[RNDSZ];
int c,c2;
   #ifdef TEST
      c = 0;
      while(SEED[c] != '\0') {
         if(SEED[c] < 0) alarm("not ASCII in SEED",NULL,NULL);
         c++;
         }
      if(c<RNDSZ) alarm("SEED to short",NULL,NULL);
   #endif
   for(c=0; c<RNDSZ; c++) key[c] = 0;
   c = c2 = 0;
   while(1) {
      if(SEED[c] == '\0') break;
      key[c2] = (key[c2] * 256 + SEED[c]) % 4001; //?? magic == pRIME
      c++;
      if(++c2>=RNDSZ) c2 = 0;
      }
   for(RndIdx = 0, c=0; c<RNDSZ; RndVec[c] = (int)key[c], c++) ;
}//reset_generator


//**********************************************
double half_life(int generations) {
//   for(c=0; c<100; c++) {
//       if(c%5==0) printf("\n");
//       printf(" %12.9f,",half_life(c));
//       }

const double HL_GEN[150] = {
  0.000000000,  0.500000000,  0.707106590,  0.793700218,  0.840895653,
  0.870551109,  0.890898705,  0.905724525,  0.917004585,  0.925875664,
  0.933033943,  0.938931465,  0.943873405,  0.948077202,  0.951695442,
  0.954840660,  0.957602501,  0.960045815,  0.962224007,  0.964175224,
  0.965935707,  0.967532158,  0.968983650,  0.970313072,  0.971531868,
  0.972655296,  0.973692894,  0.974654198,  0.975548744,  0.976382256,
  0.977160454,  0.977889061,  0.978571892,  0.979214668,  0.979819298,
  0.980391502,  0.980929375,  0.981440544,  0.981925011,  0.982384682,
  0.982821465,  0.983235359,  0.983632088,  0.984009743,  0.984370232,
  0.984715462,  0.985045433,  0.985360146,  0.985663414,  0.985953331,
  0.986231804,  0.986500740,  0.986758232,  0.987006187,  0.987246513,
  0.987477303,  0.987698555,  0.987914085,  0.988120079,  0.988320351,
  0.988514900,  0.988701820,  0.988883018,  0.989058495,  0.989228249,
  0.989392281,  0.989552498,  0.989706993,  0.989857674,  0.990004539,
  0.990147591,  0.990284920,  0.990418434,  0.990550041,  0.990675926,
  0.990799904,  0.990921974,  0.991038322,  0.991152763,  0.991265297,
  0.991374016,  0.991478920,  0.991581917,  0.991683006,  0.991782188,
  0.991877556,  0.991972923,  0.992064476,  0.992154121,  0.992241859,
  0.992327690,  0.992411613,  0.992493629,  0.992573738,  0.992653847,
  0.992730141,  0.992806435,  0.992878914,  0.992951393,  0.993023872,
  0.993092537,  0.993161201,  0.993227959,  0.993292809,  0.993357658,
  0.993420601,  0.993481636,  0.993542671,  0.993601799,  0.993660927,
  0.993718147,  0.993775368,  0.993830681,  0.993884087,  0.993937492,
  0.993990898,  0.994042397,  0.994093895,  0.994143486,  0.994193077,
  0.994240761,  0.994288445,  0.994334221,  0.994379997,  0.994425774,
  0.994469643,  0.994513512,  0.994557381,  0.994599342,  0.994641304,
  0.994681358,  0.994723320,  0.994763374,  0.994801521,  0.994839668,
  0.994877815,  0.994915962,  0.994954109,  0.994990349,  0.995026588,
  0.995060921,  0.995095253,  0.995131493,  0.995163918,  0.995198250,
  0.995230675,  0.995263100,  0.995295525,  0.995327950,  0.995358467};
```

```
   double hi = 1.0,
          lo = 0.0,
          tempd = 0.0;
   int c;
   if(generations < 0) return(0.0);
   if(generations >= 999) return (1.0);
   if(generations < 150) return HL_GEN[generations];
   for(c=0; c<20; c++) {
      tempd = (hi + lo) / 2.0;
      if (pow(tempd,generations) < 0.5) lo = tempd;
      if (pow(tempd,generations) > 0.5) hi = tempd;
      }
   return(tempd);
}//half_life

//**************************************************
void hl_factors(void) {
   int c;
   for(c=0; c<MIO; c++)
      HL_IO [c] = half_life(HL_IOG[c]);
}//hl_factors

//**************************************************
void setEqu(void) {
   int nt[MIO];
   int c,c2;
   for(c=0; c<N_C; c++) {
      for(c2=0; c2<MIO; c2++)
         nt[c2] = 0;
      for(c2=0; c2<SynNum[c]; c2++)
         nt[SynType[c][c2]]++;
      for(c2=0; c2<MIO; c2++)
         if(nt[c2] > 0) SynEq[c][c2] = 1.0 / nt[c2];
      }
}//setEqu

//*********************************************************************
void retina(void) { //looks ahead
int c;
   //using normal logic
   for(c=0; c<9; c++) {Pix[c][0]=0; Pix[c][1]=0; Pix[c][2]=0; Pix[c][3]=0; }
   switch (Direc) { //store world's colors in pixel[ ][0]
      case 'n':Pix[1][0] = World[Frog_po_y-2][Frog_po_x-1]; //up left
               Pix[2][0] = World[Frog_po_y-2][Frog_po_x  ]; //up
               Pix[3][0] = World[Frog_po_y-2][Frog_po_x+1]; //up right
               Pix[4][0] = World[Frog_po_y-1][Frog_po_x-1]; //middle left
               Pix[5][0] = World[Frog_po_y-1][Frog_po_x  ]; //middle
               Pix[6][0] = World[Frog_po_y-1][Frog_po_x+1]; //middle right
               Pix[7][0] = World[Frog_po_y  ][Frog_po_x-1]; //left
               Pix[8][0] = World[Frog_po_y  ][Frog_po_x+1]; //right
               break;
      case 'e':Pix[1][0] = World[Frog_po_y-1][Frog_po_x+2];
               Pix[2][0] = World[Frog_po_y  ][Frog_po_x+2];
               Pix[3][0] = World[Frog_po_y+1][Frog_po_x+2];
               Pix[4][0] = World[Frog_po_y-1][Frog_po_x+1];
               Pix[5][0] = World[Frog_po_y  ][Frog_po_x+1];
               Pix[6][0] = World[Frog_po_y+1][Frog_po_x+1];
               Pix[7][0] = World[Frog_po_y-1][Frog_po_x  ];
               Pix[8][0] = World[Frog_po_y+1][Frog_po_x  ];
               break;
      case 's':Pix[1][0] = World[Frog_po_y+2][Frog_po_x+1];
               Pix[2][0] = World[Frog_po_y+2][Frog_po_x  ];
               Pix[3][0] = World[Frog_po_y+2][Frog_po_x-1];
               Pix[4][0] = World[Frog_po_y+1][Frog_po_x+1];
               Pix[5][0] = World[Frog_po_y+1][Frog_po_x  ];
               Pix[6][0] = World[Frog_po_y+1][Frog_po_x-1];
               Pix[7][0] = World[Frog_po_y  ][Frog_po_x+1];
               Pix[8][0] = World[Frog_po_y  ][Frog_po_x-1];
               break;
      case 'w':Pix[1][0] = World[Frog_po_y+1][Frog_po_x-2];
               Pix[2][0] = World[Frog_po_y  ][Frog_po_x-2];
               Pix[3][0] = World[Frog_po_y-1][Frog_po_x-2];
               Pix[4][0] = World[Frog_po_y+1][Frog_po_x-1];
               Pix[5][0] = World[Frog_po_y  ][Frog_po_x-1];
               Pix[6][0] = World[Frog_po_y-1][Frog_po_x-1];
               Pix[7][0] = World[Frog_po_y+1][Frog_po_x  ];
               Pix[8][0] = World[Frog_po_y-1][Frog_po_x  ];
               break;
      }
   for(c=1; c<=8; c++) //inverted logic: 0:firing else:inactive
      switch(Pix[c][0]) {   //transform colors in red, green & blue pixels
         case 0:Pix[c][1] = 1; Pix[c][2] = 1; Pix[c][3] = 1; break; //black

         case 1:Pix[c][1] = 0; Pix[c][2] = 1; Pix[c][3] = 1; break; //red

         case 2:Pix[c][1] = 1; Pix[c][2] = 0; Pix[c][3] = 1; break; //green

         case 3:Pix[c][1] = 1; Pix[c][2] = 1; Pix[c][3] = 0; break; //blue

         case 4:Pix[c][1] = 0; Pix[c][2] = 0; Pix[c][3] = 1; break; //yellow

         case 5:Pix[c][1] = 1; Pix[c][2] = 0; Pix[c][3] = 0; break; //cyan

         case 6:Pix[c][1] = 0; Pix[c][2] = 1; Pix[c][3] = 0; break; //magenta

         case 7:Pix[c][1] = 0; Pix[c][2] = 0; Pix[c][3] = 0; break; //white
         default: assert(0);//(0,"funny RGB component in pixel");
         }
}//retina


void compass(void) {
   if(AREA_B[3]+AREA_E[3] > 0) //test for north
   switch (Direc) {            //inverted logic
      case 'n':North = 0; East = 1; South = 1; West = 1;
               break;
      case 'e':North = 1; East = 0; South = 1; West = 1;
               break;
      case 's':North = 1; East = 1; South = 0; West = 1;
               break;
      case 'w':North = 1; East = 1; South = 1; West = 0;
               break;
      }
}//compass


//**************************************************
char motor_navigation(char reset) {
#define wAIT  250
#define pAST  0.95
#define nOW   0.05
static int pause = wAIT;    //wait for cold-start/reset frog to warm-up
static char status = '\0'; // \0 no behavior
                           //  l left
                           //  r right
                           //  m move(free)
                           //  e no move(empty)
                           //  + move(food), reinforced
                           //  c crash, punished
```

```
double left = 0.0, move = 0.0, right = 0.0;
int c,opox,opoy;  //old position in X and Y axis

   switch(reset) {
      case ' ':break;
      case 'f':pause = wAIT; break; //frozen
      case '+':status = '+';
               pause = wAIT;
               break;
      case '-':status = 'c';
               pause = wAIT;
               break;
      default:printf("\n\007bad motor_navigation parameter"); getch();
      }
   if(pause > 0) {
      --pause;
      switch(status) {
         case'\0':
         case'e':
         case'l':
         case'r':
         case'm':break;
         case'+'://for(c=AREA_B[10]; c<=AREA_E[10]; AuxInp[c]+=STI, c++);
                 break; ///???
         case'c'://for(c=AREA_B[11]; c<=AREA_E[12]; AuxInp[c]+=STI, c++);
                 break; ///???
         default:printf("\n\007funny motor_navigation status"); getch();
         }
      return(status);
      }
   status = '\0';
   opox = Frog_po_x; //store actual position in case it tries to move and crashes
   opoy = Frog_po_y;

   for(c=AREA_B[7]; c<=AREA_E[7]; c++)
      if(!State[c]) left = left + 1.0;
   left /= AREA_E[7] - AREA_B[7] + 1;
   Left = (Left * pAST) + (left * nOW);

   for(c=AREA_B[8]; c<=AREA_E[8]; c++)
      if(!State[c]) move = move + 1.0;
   move /= AREA_E[8] - AREA_B[8] + 1;
   Move = (Move * pAST) +(move * nOW);

   for(c=AREA_B[9]; c<=AREA_E[9]; c++)
      if(!State[c]) right = right + 1.0;
   right /= AREA_E[9] - AREA_B[9] + 1;
   Right = (Right * pAST) +(right * nOW);

                         //order changed: move has priority
   if (Move > MOTOR_TH) {
      switch (Direc) {
         case 'n': --Frog_po_y; break;
         case 'e': ++Frog_po_x; break;
         case 's': ++Frog_po_y; break;
         case 'w': --Frog_po_x; break;
         }
      switch(World[Frog_po_y][Frog_po_x]) {
         case 0:status = 'e';   //empty space, no movement, no reinf.
                Frog_po_y = opoy;
                Frog_po_x = opox;
//                printf("\007");
                break;
         case 1://status = 'c';   //rock (red), no movement, punishment
                //Frog_po_y = opoy;
                //Frog_po_x = opox;
//                printf("\007");
                //break;
         case 2://status = 'm';
                //plot_n(WO_PO_X+((opox-2)*WS),WO_PO_Y+((opoy-2)*WS),
                //       WS,WO_COL[World[opoy][opox]]);
                //plot_frog();
                //break; //free (green), moves, no reinf.
         case 3:
         case 4:
         case 5:
         case 6://break;//??
         case 7://status = '+';
                //--World[Frog_po_y][Frog_po_x];
                plot_n(WO_PO_X+((opox-2)*WS),WO_PO_Y+((opoy-2)*WS),
                      WS,WO_COL[World[opoy][opox]]);
                plot_frog();
                break;
         default:printf("\n\007motor/navigation error"); getch();
         }
      pause = wAIT;
      Left = Move = Right = 0.0;
      plot_frog();
      }
   if ( (status =='\0') && (Left > MOTOR_TH) ) {
      status = 'l';
      switch (Direc) {
         case 'n': Direc = 'w'; break;
         case 'e': Direc = 'n'; break;
         case 's': Direc = 'e'; break;
         case 'w': Direc = 's'; break;
         }
      pause = wAIT;
      Left = Move = Right = 0.0;
      plot_frog();
      }
   if ( (status =='\0') && (Right > MOTOR_TH) ) {
      status = 'r';
      switch (Direc) {
         case 'n': Direc = 'e'; break;
         case 'e': Direc = 's'; break;
         case 's': Direc = 'w'; break;
         case 'w': Direc = 'n'; break;
         }
      pause = wAIT;
      Left = Move = Right = 0.0;
      plot_frog();
      }
   return(status);
}//motor_navigation

//cleans up all global vars. not defined on declaration (except RBOW)
//**********************************************
int cold_start(void) {
   int c,c2;

   Type    = (int *) malloc (N_C * sizeof(int));
   if(Type==NULL) return 1;

   State   = (int *) malloc (N_C * sizeof(int));
   if(State==NULL) return 1;

   Axon    = (int *) malloc (N_C * sizeof(int));
   if(Axon==NULL) return 1;
```

```
WaveL = (int *) malloc (N_C * sizeof(int));
if(WaveL==NULL) return 1;

SynNum  = (int *) malloc (N_C * sizeof(int));
if(SynNum==NULL) return 1;

Posflag  = (int *) malloc (N_C * sizeof(int));
if(Posflag==NULL) return 1;

TotalIo  = (float *) malloc (N_C * sizeof(float));
if(TotalIo==NULL) return 1;

Cell_Io  = (float (*)[MIO]) malloc (N_C * MIO * sizeof(float));
if(Cell_Io==NULL) return 1;

SynEq   = (float (*)[MIO]) malloc (N_C * MIO * sizeof(float));
if(SynEq==NULL) return 1;

SynType = (int (*)[MAX_SYN]) malloc (N_C * MAX_SYN * sizeof(int));
if(SynType==NULL) return 1;

SynPos = (int (*)[MAX_SYN]) malloc (N_C * MAX_SYN * sizeof(int));
if(SynPos==NULL) return 1;

Syn =  (int*(*)[MAX_SYN]) malloc (N_C * MAX_SYN * sizeof(int*));
if(Syn==NULL) return 1;

SynWei = (float (*)[MAX_SYN]) malloc (N_C * MAX_SYN * sizeof(float));
if(SynWei==NULL) return 1;

SyIo = (float (*)[MAX_SYN]) malloc (N_C * MAX_SYN * sizeof(float));
if(SyIo==NULL) return 1;

for(c=0; c<N_C; c++) {
   Type[c]    = 0; //types 0 and 1 are undefined; allowed: 2->MIO
   State[c]   = 0;
   Axon[c]    = R_P_SPAN;
   SynNum[c]  = 0;
   Posflag[c] = 1;
   TotalIo[c] = 0.0;
   for(c2 = 0; c2<MIO;  c2++) {
      Cell_Io[c][c2] = 0.0;
      SynEq  [c][c2] = 1.0; //no equalization
      }
   for(c2=0; c2<MAX_SYN; c2++) {
      SynType[c][c2] =    0;  //type 0 undefined
      SynPos [c][c2] =   -1;  //range of actual nei 0..N_C, -1 undefined
      Syn    [c][c2] = NULL;  //doesn't points to anything yet
      SynWei [c][c2] =  0.0;
      SyIo   [c][c2] =  0.0;
      }
   }

for(c=0; c<4; c+=2)
   for(c2=0; c2<MIO; c2++) { //??check
      RANGE[c][c2] = 1;
      RANGE[c+1][c2] = 0;
      }

for(c=0; c<MIO; c++) {
   HL_IOG[c] = 0;
   HL_IO[c] = 0.0;
   NT_TH[c] = 0.0;
   NT_PO[c] = 0.0;
   for(c2=0; c2<R_P_SPAN; c2++) {
```

```
            FIXDECAY[c][c2] = 0.0;
            REF_PER[c][c2] = 0.0;
            }
         }

   for(c=0; c<R_P_SPAN; c++)
      REF_PER_CELL[c] = 0.0;//??

   for(c=0; c<N_C; c++) //status of neighbor, 0 no conection, 127 itself
      for(c2=0; c2<N_C; c2++)
         NeiSta[c][c2] = (c != c2) ?  0 : 127;
   for(c=0; c<WO_SZ_Y; c++)
      for(c2=0; c2<WO_SZ_X; c2++)
         World[c][c2] = 0;
   return 0;
}//cold_start


void main_command_line_params(int argc,char *argv[]) {
   if(argc == 1) {
      if(load_struc_file("struc.txt") ) exit(1);
      if(load_param_file("param.txt") ) exit(2);
      }
   if(argc == 2) {
      if(load_struc_file("struc.txt") ) exit(1);
      if(load_param_file(argv[1]) ) exit(2);
      }
   if(argc == 3) {
      if(load_struc_file(argv[1]) ) exit(1);
      if(load_param_file(argv[2]) ) exit(2);
      }
   if(argc == 4) {
      Interactive_mode = 0;
      if(load_struc_file(argv[1]) ) exit(1);
      if(load_param_file(argv[2]) ) exit(2);
      if(strcmp(argv[3],"batch") == 0 ) {SEE=0; batch();}
      else if(strcmp(argv[3],"batchsee") == 0 ) {SEE = 1; batch();}
         else exit(3);
      }
}//main_command_line_params


//=========================== MAIN ================================
int main(int argc,char *argv[]) {
long loops,c; //main's variables
   clrscr();
   if(cold_start()) return 1;
   main_command_line_params(argc,argv);
   reset(0);
   if(start_graphics()) return 1;
   hl_factors();                 //transform half-lifes to factors
   if(EQUA) setEqu();
   plot_World();                 //show frog's world
   show_LMRG();
//   seed_orga(0,9,1,8,8,9,2,2,6,1.4189);
   while (1) {
      loops = menu_gr();         //main menu, wait for command
      if(!loops) break;          //exit ion on Quit || escape
      accu_synch(0);
      for(c=0; c<loops; c++) {   //if loops > 0  (main loop)
         retina();
         compass();
         compute();
         accu_synch(1);
         if(check_cells() != -1) {    //compute metabolism
```

```
            show_cell(check_cells());
            break;
            }
         Gen++;
         motor_navigation(' ');  //unused return
         //histogram(0);            //cleans lower-right corner too ??
         show_LMRG();
         if (++TimePos >= SHOW_GEN) {
           TimePos = 0;
           if (DispUp == 'B') display_up();
           if (DispUp == 'H') display_up();
           }
         //plot_waves();           //show EEG
         plot_depo();            //show "firing" cells (wavelength->color)
         if( kbhit() ) {
            getch();
            break;
            } //if a key was pressed, stop->menu
         }
      }
   printf("\nend\n");
   return 0;
}//main
// This is the end, my only friend, the end.
// files.c first line

#include <string.h>
#include <stdlib.h>
#include <conio.h>
#include <dos.h>
#include "ion.h"


void set_World_up(FILE *dta_fil, int load_fil) {
static byte world_model[WO_SZ_Y][WO_SZ_X];
   char line[80];
   int c,c2;
   if(load_fil) {
      for(c=0; c<WO_SZ_Y; c++) {
         fscanf(dta_fil,"%s",&line);
         if(strlen(line)!=WO_SZ_X)
            alarm("wrong line size in World",NULL,NULL);
         for(c2=0; c2<WO_SZ_X; c2++)
            world_model[c][c2] = line[c2];
         }
      assert(c==WO_SZ_Y);
      }
   reset_generator();
   for(c=0; c<WO_SZ_Y; c++)
      for(c2=0; c2<WO_SZ_X; c2++)
         switch (world_model[c][c2]) {
            case '0':
            case '.':World[c][c2] = 0; break; //darkgray (empty)
            case '1':
            case '-':World[c][c2] = 1; break; //red (stone)
            case '2':
            case '+':World[c][c2] = 2; break; //green (free)
            case '3':
            case '*':World[c][c2] = 3; break; //blue  (food)
            case '4':World[c][c2] = 4; break; //yellow
            case '5':World[c][c2] = 5; break; //cyan
            case '6':World[c][c2] = 6; break; //magenta
            case '7':World[c][c2] = 7; break; //white
            case 'r':World[c][c2] = rnd(3)+1; break;
            case 's':World[c][c2] = rnd(4)+1; break;
            case 't':World[c][c2] = rnd(5)+1; break;
            case 'u':World[c][c2] = rnd(6)+1; break;
            case 'v':World[c][c2] = rnd(7)+1; break;
            case 'F':World[c][c2] = 2;        //frog's position
                     Frog_po_y = c;
                     Frog_po_x = c2;
                     break;
            case   0:if(!load_fil) {World[c][c2] = 0; break;}
            default :alarm("unrecogniced char in World",NULL,NULL);
            }
}//set_World_up

//**************************************************
void reset(int noise) {
   int c,c2;
   for(c=0; c<MIO; c++)
      if((RANGE[0][c]-RANGE[1][c] < 1) ||
         (RANGE[2][c]-RANGE[3][c] < 1)) {
         alarm("\007 range top - range bottom <= 0",NULL,NULL);
         exit(1);
         }
   Left    = 0.0;
   Right   = 0.0;
   Move    = 0.0;
   Gen     = 0;
   Frog_po_x = 9;
   Frog_po_y = 9;
   TimePos = -1;
   Direc   = 'n';
   set_World_up(0,0); //resets generator anyway
   for(c=0; c<N_C; c++) {
      //Type determined by struc.txt
      State[c]   =  rnd(R_P_SPAN - 2) + 1;//initial states are random (!0)
      Axon[c] = R_P_SPAN; //??
      WaveL[c] =  State[c];//rnd(R_P_SPAN - 2) + 1;
      //SynNum is determined by struc.txt
      Posflag[c] = 1;
      TotalIo[c] = 0.0;
      for(c2 = 0; c2<MIO; c2++)
         Cell_Io[c][c2] = 0.0;
      for(c2=0; c2<MAX_SYN; c2++) {
         if(SynPos[c][c2] >= 0)
            SynWei [c][c2] = (rnd(RANGE[2][SynType[c][c2]]-
                                  RANGE[3][SynType[c][c2]])*0.01)+
                                 (RANGE[3][SynType[c][c2]]*0.01);
         SyIo[c][c2] =  0.0;
         }
      }
   if(noise)
      for(c=0; c<N_C; c++)
         for(c2=0; c2<SynNum[c]; c2++) {
            SyIo[c][c2] = (rnd(RANGE[0][SynType[c][c2]] -
                               RANGE[1][SynType[c][c2]])*0.01) +
                          (RANGE[1][SynType[c][c2]]*0.01);
            Cell_Io[c][SynType[c][c2]] += SyIo[c][c2];
            }
   accu_synch(0);
   accu_synch(1);
}//reset

//*************************************************
void read_nei(int reads,int readed,int syntype) { //>read---
   Syn    [reads][SynNum[reads]] = &Axon[readed]; //no check??
   SynType[reads][SynNum[reads]] = (syntype < 0) ? Type[readed] : syntype;
   SynPos [reads][SynNum[reads]] = readed;
   NeiSta [reads][readed] = (byte)SynType[reads][SynNum[reads]];
   ++SynNum[reads];
}//read_nei

//*************************************************
int wrap(int first, int last, int pos) {
   if (pos < first) pos += (last - first + 1);
      else if (pos > last)  pos -= (last - first + 1);
   return(pos);
}//wrap

//*************************************************
void loc_nei(int first, int last, int syn_typ, //local neighborhood
             int n0, int n1, int n2, int n3) {
   const int lOC_NEIS = 4;
   int c,c2,
       nei_pos,
       nei[lOC_NEIS];
   nei[0] = n0; nei[1] = n1; nei[2] = n2; nei[3] = n3;
   if( (first < 0) || (last >= N_C) || (first >= last) ) {
      printf("\n\007 out of range in loc_nei ");
      printf("\nfirst %sd  last %2d  type %2d    %3d  %3d  %3d  %3d",
             first,last,syn_typ,n0,n1,n2,n3);
      getch();
      return;
      }
   for(c=0; c<lOC_NEIS; c++)
      if( abs(nei[c]) > (last-first) ) {
         printf("\n\007 neighbor distance bigger than target size ");
         getch();
         return;
         }
   for(c=first; c<=last; c++)
      for(c2=0; c2<lOC_NEIS; c2++) {
         if(!nei[c2]) continue; //neighbor position == 0 -> no synapse
         if(SynNum[c] + 1 >= MAX_SYN) {
            printf("\n\007 too many neighbors on cell %2d",c);
            getch();
            continue;}
         nei_pos = wrap(first, last, c+nei[c2]);
         if(NeiSta[c][nei_pos] != 0) {
            printf("\n  taken: cell %d   neighbor %d (press a key)"
                   ,c,nei_pos);
            getch();
            continue;}
         Syn    [c][SynNum[c]] = &Axon[nei_pos]; //??
         SynType[c][SynNum[c]] = (syn_typ < 0) ? Type[nei_pos] : syn_typ;
         SynPos [c][SynNum[c]] = nei_pos;
         NeiSta[c][nei_pos] = (byte) SynType[c][SynNum[c]];
         ++SynNum[c];
         }
}//loc_nei

          //zone begin,id end ,quantity of syn,id type,neighbor's address
//**********************************************
void rnd_nei(int z_b, int z_e, int s_q, int s_t, int n_b, int n_e) {
   int c,c2,
       target_size,
       tempi;
   target_size = n_e - n_b + 1;
   for(c=z_b; c <= z_e; c++) {
      if(SynNum[c]+s_q >= MAX_SYN) {
         printf("\n\007too many synapses in cell:%d (%d)",c,SynNum[c]);
         printf("  %d %d %d %d %d %d",z_b,z_e,s_q,s_t,n_b,n_e);
         getch();
         continue;
         }
      else
         for(c2=SynNum[c]; c2<SynNum[c]+s_q; c2++) {
            do tempi = rnd(target_size);
               while (NeiSta[c][n_b + tempi]);
            Syn[c][c2] = &Axon [n_b + tempi]; //??
            //if s_t < 0 take the Type of neighbor
            SynType[c][c2] = (s_t < 0) ? Type[n_b + tempi] : s_t;
            SynPos [c][c2] = n_b + tempi;
            NeiSta[c][n_b + tempi] = (byte) SynType[c][c2];
            }
      SynNum[c] += s_q;
      }
}//rnd_nei

//**********************************************
//produce input from own state, sensors, constant, etc.
void virtual_syn(int first,int last,int syn_typ,float weight,char *target) {
char target_name[31][20] = {"on\0",               // 0
                            "state\0",            // 1
                            "wave_length\0",      // 2
                            "far_left_red\0",     // 3
                            "far_left_green\0",   // 4
                            "far_left_blue\0",     // 5
                            "far_middle_red\0",    // 6
                            "far_middle_green\0",  // 7
                            "far_middle_blue\0",   // 8
                            "far_right_red\0",     // 9
                            "far_right_green\0",   //10
                            "far_right_blue\0",    //11
                            "near_left_red\0",     //12
                            "near_left_green\0",   //13
                            "near_left_blue\0",    //14
                            "near_middle_red\0",   //15
                            "near_middle_green\0", //16
                            "near_middle_blue\0",  //17
                            "near_right_red\0",    //18
                            "near_right_green\0",  //19
                            "near_right_blue\0",   //20
                            "side_left_red\0",     //21
                            "side_left_green\0",   //22
                            "side_left_blue\0",    //23
                            "side_right_red\0",    //24
                            "side_right_green\0",  //25
                            "side_right_blue\0",   //26
                            "north\0",             //27
                            "east\0",              //28
                            "south\0",             //29
                            "west\0"};             //30
int recogniced = 0,
    c,c2;

   if( (first < 0) || (last >= N_C) || (first > last) ) {
      printf("\n\007 out of range in virtual syn ");
      printf("\nfirst %2d  last %2d  type %2d weight %f target %s",
             first,last,syn_typ,weight,target);
      getch();
      return;
      }
   if((syn_typ < 0) || (syn_typ >= MIO)) {
      printf("\n\007 wrong NT type in virtual syn ");
      printf("\nfirst %2d  last %2d  type %2d weight %f target %s",
             first,last,syn_typ,weight,target);
      getch();
      return;
      }

   for(c=0; c<31; c++)
      if( strcmp(target,target_name[c]) == 0 ) {
         recogniced = 1;
         if(Interactive_mode) printf("\n%s",target);
         for(c2=first; c2<=last; c2++) {
            if(SynNum[c2] + 1 >= MAX_SYN) {
               printf("\n\007 too many neighbors on cell %2d",c);
               getch();
               continue;
               }
            else switch(c) {
               case  0:SynPos [c2][SynNum[c2]] = -2;
                       //compiler bitches for lost of "const"
                       Syn    [c2][SynNum[c2]] = &ON;
                       break;
               case  1:SynPos [c2][SynNum[c2]] = -3;
                       Syn    [c2][SynNum[c2]] = &State[c2];
                       break;
               case  2:SynPos [c2][SynNum[c2]] = -4;
                       Syn    [c2][SynNum[c2]] = &WaveL[c2];//??bug: never 0
                       break;
```

```
case  3:SynPos [c2][SynNum[c2]] = -5;
        Syn    [c2][SynNum[c2]] = &Pix[1][1];
        break;
case  4:SynPos [c2][SynNum[c2]] = -5;
        Syn    [c2][SynNum[c2]] = &Pix[1][2];
        break;
case  5:SynPos [c2][SynNum[c2]] = -5;
        Syn    [c2][SynNum[c2]] = &Pix[1][3];
        break;
case  6:SynPos [c2][SynNum[c2]] = -5;
        Syn    [c2][SynNum[c2]] = &Pix[2][1];
        break;
case  7:SynPos [c2][SynNum[c2]] = -5;
        Syn    [c2][SynNum[c2]] = &Pix[2][2];
        break;
case  8:SynPos [c2][SynNum[c2]] = -5;
        Syn    [c2][SynNum[c2]] = &Pix[2][3];
        break;
case  9:SynPos [c2][SynNum[c2]] = -5;
        Syn    [c2][SynNum[c2]] = &Pix[3][1];
        break;
case 10:SynPos [c2][SynNum[c2]] = -5;
        Syn    [c2][SynNum[c2]] = &Pix[3][2];
        break;
case 11:SynPos [c2][SynNum[c2]] = -5;
        Syn    [c2][SynNum[c2]] = &Pix[3][3];
        break;
case 12:SynPos [c2][SynNum[c2]] = -5;
        Syn    [c2][SynNum[c2]] = &Pix[4][1];
        break;
case 13:SynPos [c2][SynNum[c2]] = -5;
        Syn    [c2][SynNum[c2]] = &Pix[4][2];
        break;
case 14:SynPos [c2][SynNum[c2]] = -5;
        Syn    [c2][SynNum[c2]] = &Pix[4][3];
        break;
case 15:SynPos [c2][SynNum[c2]] = -5;
        Syn    [c2][SynNum[c2]] = &Pix[5][1];
        break;
case 16:SynPos [c2][SynNum[c2]] = -5;
        Syn    [c2][SynNum[c2]] = &Pix[5][2];
        break;
case 17:SynPos [c2][SynNum[c2]] = -5;
        Syn    [c2][SynNum[c2]] = &Pix[5][3];
        break;
case 18:SynPos [c2][SynNum[c2]] = -5;
        Syn    [c2][SynNum[c2]] = &Pix[6][1];
        break;
case 19:SynPos [c2][SynNum[c2]] = -5;
        Syn    [c2][SynNum[c2]] = &Pix[6][2];
        break;
case 20:SynPos [c2][SynNum[c2]] = -5;
        Syn    [c2][SynNum[c2]] = &Pix[6][3];
        break;
case 21:SynPos [c2][SynNum[c2]] = -5;
        Syn    [c2][SynNum[c2]] = &Pix[7][1];
        break;
case 22:SynPos [c2][SynNum[c2]] = -5;
        Syn    [c2][SynNum[c2]] = &Pix[7][2];
        break;
case 23:SynPos [c2][SynNum[c2]] = -5;
        Syn    [c2][SynNum[c2]] = &Pix[7][3];
        break;
case 24:SynPos [c2][SynNum[c2]] = -5;
        Syn    [c2][SynNum[c2]] = &Pix[8][1];
```

```
                break;
            case 25:SynPos [c2][SynNum[c2]] = -5;
                    Syn    [c2][SynNum[c2]] = &Pix[8][2];
                break;
            case 26:SynPos [c2][SynNum[c2]] = -5;
                    Syn    [c2][SynNum[c2]] = &Pix[8][3];
                break;
            case 27:SynPos [c2][SynNum[c2]] = -6;
                    Syn    [c2][SynNum[c2]] = &North;
                break;
            case 28:SynPos [c2][SynNum[c2]] = -6;
                    Syn    [c2][SynNum[c2]] = &East;
                break;
            case 29:SynPos [c2][SynNum[c2]] = -6;
                    Syn    [c2][SynNum[c2]] = &South;
                break;
            case 30:SynPos [c2][SynNum[c2]] = -6;
                    Syn    [c2][SynNum[c2]] = &West;
                break;
            default:alarm("\nunrecogniced virtual synapse target",NULL,NULL);
            }
         SynType[c2][SynNum[c2]] = syn_typ;
         SynWei [c2][SynNum[c2]] = weight;
         ++SynNum[c2];
         }
      }
   if(!recogniced) {
      printf("\n\007 unrecogniced target in virtual syn. ");
      printf("\nfirst %2d  last %2d  type %2d weight %f target %s",
             first,last,syn_typ,weight,target);
      getch();
      return;
      }
}//virtual_syn


//***************************************************
int load_struc_file(char *struc_file_name) {
#define cODE_SIZE 15 //0..13, 14 == '\0'
#define pAS_SZ   256 //pascal size of strings
#define mAX_CODE  22 //total of recogniced codes
   FILE *struc_fil;
   char big_line[pAS_SZ] = "\0";
   char code [cODE_SIZE] = "\0";
   char code_list[mAX_CODE][cODE_SIZE] = { ">Eeg0---------\0", // 0
                                           ">Eeg1---------\0", // 1
                                           ">Eeg2---------\0", // 2
                                           ">Eeg3---------\0", // 3
                                           ">PLEAdummy----\0", // 4
                                           ">PAINdummy----\0", // 5
                                           ">area---------\0", // 6
                                           ">NtCol--------\0", // 7
                                           ">World--------\0", // 8
                                           ">Type---------\0", // 9
                                           ">readdummy----\0", //10
                                           ">loc_nei------\0", //11
                                           ">rnd_nei------\0", //12
                                           ">stop---------\0", //13
                                           ">virtual_syn--\0", //14
                                           ">eastdummy----\0", //15
                                           ">southdummy---\0", //16
                                           ">westdummy----\0", //17
                                           ">leftdummy----\0", //18
                                           ">movedummy----\0", //19
                                           ">rightdummy---\0", //20
```

```
                                                  ">end.---------\0" };
    int dumch,
        c,c2,
        dum1,dum2,dum3,dum4,dum5,dum6,dum7; //dummy integer 1, 2, ...??
    float dumfl;
    strcpy(big_line,struc_file_name);
    if( (struc_fil = fopen(big_line,"r+")) == NULL)
       while(1) {
          printf("\n\n[q]->quit  [space bar]->type structure-file's name ");
          dumch = getch();
          if((dumch == 'q')||(dumch =='Q')) return(1);
          else {
             printf("\n\nplease type [file_name.txt] [RETURN]: ");
             scanf("%s",&big_line);
             }
          if( (struc_fil = fopen(big_line,"r+")) == NULL)
             printf("\n\ncan't find file \"%s\"",big_line);
          else break;
          }
    strcpy(STRUCFIL,big_line);
    if(Interactive_mode) printf("\nloading structure file: %s",STRUCFIL);
    do {
       fgets(code,cODE_SIZE,struc_fil);
       if(code[0] != '>') continue;
       for(c=0; c<mAX_CODE; c++)
          if( strcmp(code,code_list[c]) == 0 ) {
             if(Interactive_mode) printf("\n%s",code);
             switch(c) {
                case  0:fscanf(struc_fil,"%d%d",&EegB[0],&EegE[0]);
                        if(Interactive_mode) printf(" %3d %3d",EegB[0],EegE[0]); break;
                case  1:fscanf(struc_fil,"%d%d",&EegB[1],&EegE[1]);
                        if(Interactive_mode) printf(" %3d %3d",EegB[1],EegE[1]); break;
                case  2:fscanf(struc_fil,"%d%d",&EegB[2],&EegE[2]);
                        if(Interactive_mode) printf(" %3d %3d",EegB[2],EegE[2]); break;
                case  3:fscanf(struc_fil,"%d%d",&EegB[3],&EegE[3]);
                        if(Interactive_mode) printf(" %3d %3d",EegB[3],EegE[3]); break;
                case  4:break;
                case  5:break;
                case  6:for(c2=0; c2<AREAS; c2++) {
                           fscanf(struc_fil,"%d%d %s",
                                  &AREA_B[c2],
                                  &AREA_E[c2],
                                  &AREA_NAME[c2]);
                           }
                        break; //ex-GREY
                case  7:fgets(big_line,pAS_SZ,struc_fil);
                        if(Interactive_mode) printf("\n");
                        for(c2=0; c2<MIO; c2++) {
                           fscanf(struc_fil,"%d",&NtCol[c2]);
                           if(Interactive_mode) printf("%4d",NtCol[c2]);
                           }
                         break;
                case  8:set_World_up(struc_fil,1); break;
                case  9:fscanf(struc_fil,"%d%d%d",&dum1,&dum2,&dum3);
                        if(Interactive_mode) printf(" %3d %3d %3d ",dum1,dum2,dum3);
                        if( (dum3 < 0) || (dum3 >= MIO) ) {
                           if(Interactive_mode) printf("\n\007wrong cell type:
%d\n",dum3);
                           getch();
                           break;
                           }
                        for(c2=dum1; c2 <= dum2; c2++)
                           Type[c2] = dum3;
                        break;
                case 10:fscanf(struc_fil,"%d%d%d",&dum1,&dum2,&dum3);
                        if(Interactive_mode) printf(" %3d %3d %3d",dum1,dum2,dum3);
                        read_nei(dum1,dum2,dum3);
                        break;
                case 11:fscanf(struc_fil,"%d%d%d%d%d%d%d",
                           &dum1,&dum2,&dum3,&dum4,&dum5,&dum6,&dum7);
                        if(Interactive_mode) printf(" %3d %3d %3d %3d %3d %3d %3d",
                           dum1,dum2,dum3,dum4,dum5,dum6,dum7);
                        loc_nei(dum1,dum2,dum3,dum4,dum5,dum6,dum7);
                        break;
                case 12:fscanf(struc_fil,"%d%d%d%d%d%d",
                           &dum1,&dum2,&dum3,&dum4,&dum5,&dum6);
                        if(Interactive_mode) printf(" %3d %3d %3d %3d %3d %3d",
                           dum1,dum2,dum3,dum4,dum5,dum6);
                        rnd_nei(dum1,dum2,dum3,dum4,dum5,dum6) ;
                        break;
                case 13:if(Interactive_mode) printf("    press any key");
                        getch();
                        break;
                case 14:fscanf(struc_fil,"%d%d%d%f %s",
                           &dum1,&dum2,&dum3,&dumfl,&big_line);
                        if(Interactive_mode) printf(" %3d %3d %3d %f %s",
                           dum1,dum2,dum3,dumfl,big_line);
                        virtual_syn(dum1,dum2,dum3,dumfl,big_line);
                        break;
                case 15:break;
                case 16:break;
                case 17:break;
                case 18:break;
                case 19:break;
                case 20:break;
                case 21:if(Interactive_mode) {
                           printf("\n  END OF DATA press any key");
                           getch();
                           clrscr();
                           }
                        goto terminate;
                }
          }
       }while(1);
    terminate:
    fclose(struc_fil);
    return(0);
 }//struc_file


 //**************************************************
 int load_param_file(char *param_file_name) {
 #define cODE_SIZE 15
 #define mAXCODES  22
 #define bIG      256
 const float fT = 100.01;
 FILE *in_fil;
 char cODE_LIST[mAXCODES][cODE_SIZE] = { ">HL_IOG-------\0",  // 0
                                         ">RANGE--------\0",  // 1
                                         ">dummy2-------\0",  // 2
                                         ">dummy3-------\0",  // 3
                                         ">NT_TH--------\0",  // 4
                                         ">NT_PO--------\0",  // 5
                                         ">MASTER_DELAY-\0",  // 6
                                         ">DELAY--------\0",  // 7
                                         ">dummy8-------\0",  // 8
                                         ">Spontaneous--\0",  // 9
                                         ">dummy10------\0",  //10
                                         ">dummy11------\0",  //11
                                         ">dummy12------\0",  //12
                                         ">MOTOR_TH-----\0",  //13
                                         ">REF_PER------\0",  //14
                                         ">REF_PER_CELL-\0",  //15
                                         ">EQUALIZE-----\0",  //16
                                         ">dummy17------\0",  //17
                                         ">FIXDECAY-----\0",  //18
                                         ">dummy19------\0",  //19
                                         ">dummy20------\0",  //20
                                         ">end.---------\0" };//21
 char code[cODE_SIZE] = "\0",
      big_line[bIG]   = "\0";
 int missing[mAXCODES],
     abort_flag = 0,
     //auto_flag = 1,
     dumch,
     c,c2,c3;

    for(c=0; c<mAXCODES; missing[c]=0, c++) ;
    //set dummies as read
    missing[ 2] = 1;
    missing[ 3] = 1;
    missing[ 8] = 1;
    missing[10] = 1;
    missing[11] = 1;
    missing[12] = 1;
    missing[17] = 1;
    missing[19] = 1;
    missing[20] = 1;
 //   gotoxy(1,1);
    strcpy(big_line,param_file_name);
    if( (in_fil = fopen(big_line,"r+")) == NULL)
       while(1) {
          //auto_flag = 0;
          printf("\n\n[q]->quit  [space bar]->type parameters-file's name ");
          dumch = getch();
          if((dumch == 'q')||(dumch =='Q'))
             return(1); //goto
          else {
             printf("\n\nplease type [file_name.txt] [RETURN]: ");
             scanf("%s",&big_line);
             }
          if( (in_fil = fopen(big_line,"r+")) == NULL)
             printf("\n\ncan't find file \"%s\"",big_line);
          else break;
          }
    strcpy(PARAMFIL,big_line);
    if(Interactive_mode) printf("\nloading parameter file: %s",PARAMFIL);
    while(!feof(in_fil)) {
       strcpy(code, "\0");
       fgets(code,cODE_SIZE,in_fil);
       if(code[0] != '>') continue;  //ignore lines not beginning with '>'
       for(c=0; c<mAXCODES; c++)
          if( strcmp(code,cODE_LIST[c]) == 0 ) {
             missing[c]++;
             if(Interactive_mode) printf("\n%s",code);         //print heading
             fgets(big_line,bIG,in_fil); //read rest of the line
             switch(c) {
                case  0:for(c2=0; c2<MIO; c2++) {
                           fscanf(in_fil,"%d",&HL_IOG[c2]);
                           if(Interactive_mode) printf(" %4d ",HL_IOG[c2]);
                           }
                        break;
                case  1:if(Interactive_mode) printf("\n");
                        for(c2=0; c2<MIO; c2++) {
                           fscanf(in_fil,"%d",&RANGE[0][c2]);
```

```
                    if(Interactive_mode) printf(" %4d ",RANGE[0][c2]) ;
                    }
                 if(Interactive_mode) printf("\n");
                 for(c2=0; c2<MIO; c2++) {
                    fscanf(in_fil,"%d",&RANGE[1][c2]);
                    if(Interactive_mode) printf(" %4d ",RANGE[1][c2]) ;
                    }
                 if(Interactive_mode) printf("\n");
                 for(c2=0; c2<MIO; c2++) {
                    fscanf(in_fil,"%d",&RANGE[2][c2]);
                    if(Interactive_mode) printf(" %4d ",RANGE[2][c2]) ;
                    }
                 if(Interactive_mode) printf("\n");
                 for(c2=0; c2<MIO; c2++) {
                    fscanf(in_fil,"%d",&RANGE[3][c2]);
                    if(Interactive_mode) printf(" %4d ",RANGE[3][c2]) ;
                    }
                 break;
         case  2:break;
         case  3:break;
         case  4:for(c2=0; c2<MIO; c2++) {
                    fscanf(in_fil,"%lf",&NT_TH[c2]);
                    if(Interactive_mode) printf(" %4d ",(int)(NT_TH[c2] * fT) );
                    }
                 break;
         case  5:for(c2=0; c2<MIO; c2++) {
                    fscanf(in_fil,"%lf",&NT_PO[c2]);
                    if(Interactive_mode) printf(" %4d ",(int)(NT_PO[c2] * fT) );
                    }
                 break;
         case  6:fscanf(in_fil,"%d",&MASTER_DELAY);
                 if(Interactive_mode) printf(" %4d",MASTER_DELAY);
                 --MASTER_DELAY;
                 break;
         case  7:for(c2=0; c2<MIO; c2++) {
                    fscanf(in_fil,"%d",&DELAY[c2]);
                    if(Interactive_mode) printf(" %4d ",DELAY[c2]);
                    --DELAY[c2];
                    }
                 break;
         case  8:break;
         case  9:fscanf(in_fil,"%d",&Spontaneous);
                 if(Interactive_mode) printf(" %4d",Spontaneous);
                 break;
         case 10:break;
         case 11:break;
         case 12:break;
         case 13:fscanf(in_fil,"%lf",&MOTOR_TH);
                 if(Interactive_mode) printf(" %4d ",(int)(MOTOR_TH * fT) );
                 break;
         case 14:for(c2=0; c2<MIO; c2++) {
                    if(Interactive_mode) printf("\n");
                    for(c3=0; c3<R_P_SPAN; c3++) {
                       fscanf(in_fil,"%lf",&REF_PER[c2][c3]);
                       if(Interactive_mode) printf(" %3d",(int)(REF_PER[c2][c3]
* 100.01));
                       }
                    }
                 break;
         case 15:if(Interactive_mode) printf("\n");
                 for(c2=0; c2<R_P_SPAN; c2++) {
                    fscanf(in_fil,"%lf",&REF_PER_CELL[c2]);
                    if(Interactive_mode) printf(" %4.2lf ",REF_PER_CELL[c2]);
                    }
                 break;
```

```
               case 16:fscanf(in_fil,"%d",&EQUA);
                       if(Interactive_mode) printf(" %4d ",EQUA);
                       break;
               case 17:break;
               case 18:for(c2=0; c2<MIO; c2++) {
                          if(Interactive_mode) printf("\n");
                          for(c3=0; c3<R_P_SPAN; c3++) {
                             fscanf(in_fil,"%lf",&FIXDECAY[c2][c3]);
                             if(Interactive_mode) printf(" %3d",(int)(FIXDECAY[c2][c3]
* 100.01));
                             }
                          }
                       break;
               case 19:break;
               case 20:if(Interactive_mode) {
                          printf("    press any key");//?? two stops & crash
                          getch();
                          }
                       break;
               case 21:if(Interactive_mode) {
                          printf("\n  END OF DATA press any key");
                          getch();
                          clrscr();
                          }
                       break;
               }
            }
      }
   fclose(in_fil);
   missing[16] = 1; //stop not required
   for(c=0; c<mAXCODES; c++)
      if(missing[c] != 1) {
         abort_flag = 1;
         printf("\n\n*** %s ",cODE_LIST[c]);
         printf("missing, duplicated or bad formated ***\n");
         getch();
         }
   if (abort_flag) return(1); else return(0);
}//load_params

//**************************************************
int save_params(int interactive, char *genename) {
#define bIG 256
   const char *oUT_NAME  = "param.txt\0";     //default output file
   FILE *out_fil;
   char big_line[bIG] = "\0";
   int dumch,c,c2;

   if(interactive)
      while(1) { //**************saving***************
         strcpy(big_line,oUT_NAME);
         clean_line(1,1);
         printf("[RETURN]->save as %s   [x]->exit ",big_line);
         printf("  [any letter]->type a file name ");
         dumch = getch();
         if(dumch != 13) { //if not a [RETURN]
            if((dumch == 'x')||(dumch == 'X'))
               return(1);
            else {
               clean_line(1,1);
               printf("please type [file name] [RETURN]  ");
               scanf("%s",&big_line);
               }
            }
         if( (out_fil = fopen(big_line,"w+")) == NULL) {
```

```
         clean_line(1,1);
         printf("can't save file \"%s\"",big_line);
         delay(2000);
         }
         else break;
      }
else
   if( (out_fil = fopen(genename,"w+")) == NULL) {
      clean_line(1,1);
      printf("can't save file \"%s\"",big_line);
      delay(2000);
      return(-1);
      }

fprintf(out_fil,"/*\n>HL_IOG-------\n");  //0
for(c=0; c<MIO; c++)
   fprintf(out_fil," %4d ",HL_IOG[c]);

fprintf(out_fil,"\n>RANGE--------\n");    //1
for(c=0; c<MIO; c++)
   fprintf(out_fil," %4d ",RANGE[0][c]);
fprintf(out_fil,"\n");
for(c=0; c<MIO; c++)
   fprintf(out_fil," %4d ",RANGE[1][c]);
fprintf(out_fil,"\n");
for(c=0; c<MIO; c++)
   fprintf(out_fil," %4d ",RANGE[2][c]);
   fprintf(out_fil,"\n");
for(c=0; c<MIO; c++)
   fprintf(out_fil," %4d ",RANGE[3][c]);

fprintf(out_fil,"\n>NT_TH--------\n");    //4
for(c=0; c<MIO; c++)
   fprintf(out_fil," %7.4lf ",NT_TH[c]);

fprintf(out_fil,"\n>NT_PO--------\n");    //5
for(c=0; c<MIO; c++)
   fprintf(out_fil," %7.4lf ",NT_PO[c]);

fprintf(out_fil,"\n>MASTER_DELAY-\n");    //6
fprintf(out_fil," %4d",MASTER_DELAY+1);

fprintf(out_fil,"\n>DELAY--------\n");    //7
for(c=0; c<MIO; c++)
   fprintf(out_fil," %4d ",DELAY[c]+1);

fprintf(out_fil,"\n>Spontaneous--\n");    //9
fprintf(out_fil," %4d",Spontaneous);

fprintf(out_fil,"\n>MOTOR_TH-----\n");    //13
fprintf(out_fil," %1.4lf",MOTOR_TH);

fprintf(out_fil,"\n>REF_PER------\n");    //14
for(c=0; c<MIO; c++) {
   for(c2=0; c2<R_P_SPAN; c2++)
      fprintf(out_fil," %1.2lf",REF_PER[c][c2]);
   fprintf(out_fil,"\n");
   }
fprintf(out_fil,"\n>REF_PER_CELL-\n");    //15
for(c2=0; c2<R_P_SPAN; c2++)
   fprintf(out_fil," %4.2lf",REF_PER_CELL[c2]);
fprintf(out_fil,"\n");

fprintf(out_fil,"\n>EQUALIZE-----\n");    //16
fprintf(out_fil," %d",EQUA);
```

```
   fprintf(out_fil,"\n>FIXDECAY-----\n");    //18
   for(c=0; c<MIO; c++) {
      for(c2=0; c2<R_P_SPAN; c2++)
         fprintf(out_fil," %5.2lf",FIXDECAY[c][c2]);
      fprintf(out_fil,"\n");
      }

   fprintf(out_fil,"\n>end.---------\n*/");  //21 end
   fclose(out_fil);
   menu_line_tx();
   return(0);
}//save_params

//files.c last line
```

```
//menus.c first line

#include <conio.h>    //gotoxy getch
#include <string.h>   //strcpy strlen
#include <dos.h>      //delay
#include <grx20.h>    //outtextxy
#include "ion.h"

char dumline[81];

//**************************************************
void clean_line(int first, int last) {
int c;
   dumline[0] = 'ú';
   for(c=1; c<79; c++) dumline[c] = ' ';
   dumline[79] = 'ú';
   dumline[80] = '\0';
   assert(first > 0 && last <= 25 && last >= first);//,"cleaning false line");//??
   for(c=first; c<=last; c++) {
      gotoxy(1,c);
      printf("%s",dumline);
      }
   gotoxy(1,first);
}//clean_line

//**************************************************
void menu_line_tx(void) {
   clean_line(1,1);
   setcolor(DARKGRAY); //?? to print screen grabs on white paper
   switch(DispUp) {
      case 'B':printf(" Quit 0 1 2 3 bars Histo paraMs ref-Per");
               printf(" Fix-io cEll Neig Wei Reset saVe       ");
               break;
      case 'H':printf(" Quit 0 1 2 3 Bars histo paraMs ref-Per");
               printf(" Fix-io cEll Neig Wei Reset saVe       ");
               break;
      case 'M':printf(" Quit 0 1 2 3 Bars Histo params  ref-Per");
               printf(" Fix-io cEll Neig Wei Reset saVe RETURN");
               break;
      case 'P':printf(" Quit 0 1 2 3 Bars Histo paraMs ref-per");
               printf(" Fix-io cEll Neig Wei Reset saVe RETURN");
               break;
      case 'F':printf(" Quit 0 1 2 3 Bars Histo paraMs ref-Per");
               printf(" fix-io cEll Neig Wei Reset saVe RETURN");
               break;
      default:alarm(" DispUp undefined",NULL,NULL);
      }
}//menu_line_tx

void menu_line_gr(void) {
   switch(DispUp) {
      case 'B':sprintf(dumline," Quit 0 1 2 3 bars Histo paraMs ref-Per ");
               GrTextXY(  0,  0, dumline, WHITE, BLACK);
               sprintf(dumline,"Fix-io cEll Neig Wei Reset saVe       ");
               GrTextXY(FX*34,  0,dumline , WHITE, BLACK);
               break;
      case 'H':sprintf(dumline," Quit 0 1 2 3 Bars histo paraMs ref-Per ");
               GrTextXY(  0,  0, dumline, WHITE, BLACK);
               sprintf(dumline,"Fix-io cEll Neig Wei Reset saVe       ");
               GrTextXY(FX*34,  0,dumline , WHITE, BLACK);
               break;
      case 'M':sprintf(dumline," Quit 0 1 2 3 Bars Histo params ref-Per ");
               GrTextXY(  0,  0,dumline , WHITE, BLACK);
               sprintf(dumline,"Fix-io cEll Neig Wei Reset saVe RETURN");
               GrTextXY(FX*34,  0,dumline , WHITE, BLACK);
```

```
                 break;
        case 'P':sprintf(dumline," Quit 0 1 2 3 Bars Histo paraMs ref-per ");
                 GrTextXY(  0,  0,dumline , WHITE, BLACK);
                 sprintf(dumline,"Fix-io cEll Neig Wei Reset saVe RETURN");
                 GrTextXY(FX*34,  0,dumline , WHITE, BLACK);
                 break;
        case 'F':sprintf(dumline," Quit 0 1 2 3 Bars Histo paraMs ref-Per ");
                 GrTextXY(  0,  0,dumline , WHITE, BLACK);
                 sprintf(dumline,"fix-io cEll Neig Wei Reset saVe RETURN");
                 GrTextXY(FX*34,  0,dumline , WHITE, BLACK);
                 break;
        default:alarm(" DispUp undefined",NULL,NULL);
        }
}//menu_line_gr


//**************************************************
double ask_double(char message[],double min,double max,double old) {
double tempd = 0.0;
   clean_line(1,1);
   printf("%s min:%1.3lf max:%1.3lf actual:%1.3lf new: ",message,min,max,old);
   gets(dumline);
   if(dumline[0] =='\0') { menu_line_tx(); return(old); }
      else {
      sscanf(dumline,"%lf",&tempd);
      menu_line_tx();
      if(tempd<min || tempd>max) return(old); else return(tempd);
      }
}//ask_double

//**************************************************
int ask_int(char message[],int min,int max,int old) {
int tempi = 0;
   clean_line(1,1);
   printf("%s min:%d max:%d actual:%d new: ",message,min,max,old);
   gets(dumline);
   if(dumline[0] =='\0') { menu_line_tx(); return(old); }
      else {
      sscanf(dumline,"%d",&tempi);
      menu_line_tx();
      if(tempi<min || tempi>max) return(old); else return(tempi);
      }
}//ask_int


//**************************************************
int change_cell() {
   int temp,c;
   gotoxy( 7,2); printf("®91");
   gotoxy(19,2); printf("®92");
   gotoxy(30,2); printf("®93");
   gotoxy(45,2); printf("®94");
   for(c=0; c<MIO; c++) {gotoxy(c*9+11,4); printf("®%d1",c); }
   temp = ask_int("type the digits after '®' ",1,94,0);
   return(temp);
}//change_cell

//**************************************************
void show_syn(int cell, int begin, int tempi) {
const float fT = 1.0;
   int c;
   printf("\n\nTY");
   for(c=begin; c<tempi; c++)
      printf("%7d",SynType[cell][c]);
   printf("\nNE");
   for(c=begin; c<tempi; c++)
      printf("%7d",SynPos[cell][c]);
   printf("\nIO");
   for(c=begin; c<tempi; c++)
      printf("% 6.4f",SyIo[cell][c] * fT);
   printf("\nWE");
   for(c=begin; c<tempi; c++)
      printf("% 6.4f",SynWei[cell][c] * fT);
}//show_syn

//**************************************************
void show_cell(int cell) {
   double ntinploc[MIO]; //nt input local-var
   int c = 0, tempi = 0, returned = 0, dumch = '-';

   GrSetMode(GR_default_text);

   do {
      clean_line(1,25);
      gotoxy(1,1);
      if(cell < 0) cell += N_C;
         else if(cell >= N_C) cell -= N_C;
      printf("cell %2d®[cursor]    [return]®changes   ",cell);
      printf("[space]®exit   generation: %d",Gen);
      printf("\ntype %d    state %2d    axon %2d",
             Type[cell],State[cell],Axon[cell]);
      printf("    w.length %2d    syn-num %2d    +-%d  total%6.3f",
             WaveL[cell],SynNum[cell],Posflag[cell],TotalIo[cell]);
      printf("\n     [0]      [1]      [2]      [3]");
      printf("      [4]      [5]      [6]      [7]");
      printf("\nIO");
      for(c=0; c<MIO; c++) {
         printf("  %6.3f ",Cell_Io[cell][c]*SynEq[cell][c]);
         ntinploc[c] = 0.0;
         }

      printf("\nio");               //debug
      for(c=0; c<SynNum[cell]; c++)
         ntinploc[SynType[cell][c]] += SyIo[cell][c];
      for(c=0; c<MIO; c++)
         printf(" %6.4f",ntinploc[c]);
      if(SynNum[cell] > 0) {
         tempi = (SynNum[cell] > 10) ? 10 : SynNum[cell];
         show_syn(cell, 0,tempi);
         }
      if(SynNum[cell] > 10) {
         tempi = (SynNum[cell] > 20) ? 20 : SynNum[cell];
         show_syn(cell,10,tempi);
         }
      if(SynNum[cell] > 20) {
         tempi = (SynNum[cell] > 30) ? 30 : SynNum[cell];
         show_syn(cell,20,tempi);
         }
      if(SynNum[cell] > 30) {
         tempi = (SynNum[cell] > 40) ? 40 : SynNum[cell];
         show_syn(cell,30,tempi);
         }
      dumch = getch();
      if(dumch==0)//double-byte char
         switch(getch()) {
            case 72: cell -= 10; break;
            case 77: cell++;     break;
            case 80: cell += 10; break;
            case 75: cell--;     break;
            }
         else if(dumch==13)
            switch(returned=change_cell()) {
               case  1: case 11: case 21: case 31:
               case 41: case 51: case 61: case 71:
                       Cell_Io[cell][returned/10] =
                          ask_double("NT/IOn",0.0,1000.0,
                          Cell_Io[cell][returned/10] *
                          SynEq[cell][returned/10])
                          / SynEq[cell][returned/10];
                       break;
               case  2: case 12: case 22: case 32:
               case 42: case 52: case 62: case 72:
                       break;
               case 91:Type[cell] = ask_int("cell type:",2,MIO,Type[cell]);
                       break;
               case 92:State[cell] =
                          ask_int("cell state:",0,1000,State[cell]);
                       break;
               case 93:Axon[cell] =
                          ask_int("axon state:",0,R_P_SPAN,Axon[cell]);
                       break;
               case 94:WaveL[cell] =
                          ask_int("wavelength:",1,1000,WaveL[cell]);
                       break;
               default:clean_line(1,1);
                       printf("DOES NOT COMPUTE");
                       delay(1000);
                       menu_line_tx();
               }
            else if(dumch == ' ') break;
      }
      while (1);
   start_graphics();
   plot_World();
   show_LMRG();
}//show_cell

//**************************************************
void def_menu_gr(void) {  //default menu, parameters
const int lAST_COL = 400;
char line[81]="\0";
int c;

   for(c=0; c<MIO; c++) {
      sprintf(line,"%d   %4d  %1.4f  %6.3f  %1.3f  %2d",
         c,HL_IOG[c],HL_IO[c],NT_PO[c],NT_TH[c],DELAY[c]+1);
      GrTextXY(0,(FY*c)+(2*FY),line,WHITE,BLACK);
      switch (c) {
         case 0:sprintf(line,"  %4.3lf  moto",MOTOR_TH);
                GrTextXY(lAST_COL,(FY*c)+(2*FY),line,WHITE,BLACK);break;
         case 1:sprintf(line," %3d    delay",MASTER_DELAY+1);
                GrTextXY(lAST_COL,(FY*c)+(2*FY),line,WHITE,BLACK);break;
         case 2:sprintf(line," %3d  Spontan",Spontaneous);
                GrTextXY(lAST_COL,(FY*c)+(2*FY),line,WHITE,BLACK);break;
         case 3:sprintf(line,"   %d     equa",EQUA);
                GrTextXY(lAST_COL,(FY*c)+(2*FY),line,WHITE,BLACK);break;
         }
      }
   sprintf(line,"  %s",STRUCFIL);
   GrTextXY(lAST_COL,(FY*6)+(2*FY),line,WHITE,BLACK);
   sprintf(line,"  %s",PARAMFIL);
   GrTextXY(lAST_COL,(FY*7)+(2*FY),line,WHITE,BLACK);
   sprintf(line,
      "IO  «l-g   «l-f     pow   thres  del");
   GrTextXY(0,FY,line,WHITE,BLACK);
   menu_line_gr();
}//def_menu_gr


//**************************************************
void show_table(int x,int y) {
   int c,c2;
   clean_line(2,11);
   switch (DispUp) {
      case 'P':for(printf("y:IO    x:time-step"), c=0; c<y; c++) {
                  printf("\n%d)",c);
                  for(c2=0; c2<x; c2++)
                     if(REF_PER[c][c2] == 0.0) printf(" 0.0 ");
                        else printf(" %1.2lf",REF_PER[c][c2]);
                  }
               for(printf("\n  "), c=0; c<x; c++)
                  if(c<10) printf(" [0%d]",c);
                  else printf(" [%2d]",c);
               printf("\nC8");
               for(c2=0; c2<x; c2++)
                  if(REF_PER_CELL[c2] == 0.0) printf(" 0.0 ");
                  else printf(" %1.2lf",REF_PER_CELL[c2]);
               break;
      case 'F':for(printf("y:IO    x:time-step"), c=0; c<y; c++) {
                  printf("\n%d",c);
                  for(c2=0; c2<x; c2++)
                     if(FIXDECAY[c][c2] == 0.0) printf("  0.0 ");
                     else printf(" %5.2lf",FIXDECAY[c][c2]);
                  }
               for(printf("\n  "), c=0; c<x; c++)
                  if(c<10) printf(" [0%d] ",c);
                  else printf(" [%2d] ",c);
               //printf("time");
               break;
      }
}//show_table

//**************************************************
void show_table_gr(int x,int y) {
   int c,c2;
   switch (DispUp) {
      case 'P':sprintf(dumline,"y:IO    x:time-step");
               GrTextXY(0,FY,dumline,WHITE,BLACK);
               for( c=0; c<y; c++) {
                  sprintf(dumline,"%d)",c);
                  GrTextXY(0,(FY*c)+(FY*2),dumline,WHITE,BLACK);
                  for(c2=0; c2<x; c2++) {
                     if(REF_PER[c][c2] == 0.0) sprintf(dumline," 0.0 ");
                        else sprintf(dumline," %1.2lf",REF_PER[c][c2]);
                     GrTextXY((50*c2)+15,(FY*c)+(FY*2),dumline,WHITE,BLACK);
                     }
                  }
               for(c=0; c<x; c++) {
                  if(c<10) sprintf(dumline," [0%d]",c);
                  else sprintf(dumline," [%2d]",c);
                  GrTextXY((50*c)+15,(FY*10),dumline,WHITE,BLACK);
                  }
               sprintf(dumline,"C  ");
               GrTextXY(0,FY*(y+3),dumline,WHITE,BLACK);
               for(c2=0; c2<x; c2++) {
                  if(REF_PER_CELL[c2] == 0.0) sprintf(dumline," 0.0 ");
                  else sprintf(dumline," %1.2lf",REF_PER_CELL[c2]);
                  GrTextXY((50*c2)+15,(FY*(y+3)),dumline,WHITE,BLACK);
                  }
               break;
      case 'F':sprintf(dumline,"y:IO    x:time-step");
               GrTextXY(0,FY,dumline,WHITE,BLACK);
               for( c=0; c<y; c++) {
                  sprintf(dumline,"%d)",c);
                  GrTextXY(0,(FY*c)+(FY*2),dumline,WHITE,BLACK);
                  for(c2=0; c2<x; c2++) {
                     if(FIXDECAY[c][c2] == 0.0) sprintf(dumline,"  0.0 ");
                        else sprintf(dumline," %5.2lf",FIXDECAY[c][c2]);
                     GrTextXY((50*c2)+15,(FY*c)+(FY*2),dumline,WHITE,BLACK);
                     }
                  }
               for(c=0; c<x; c++) {
                  if(c<10) sprintf(dumline," [0%d]",c);
                  else sprintf(dumline," [%2d]",c);
                  GrTextXY((50*c)+20,(FY*10),dumline,WHITE,BLACK);
                  }
               break;
      }
}//show_table_gr

//**************************************************
void display_up(void) {
   switch(DispUp) {
      case 'B':bars(); break;
      case 'H':histogram(1);
               synchro_stats();
               break;
      case 'M':def_menu_gr(); break;
      case 'P':show_table_gr(R_P_SPAN,MIO); break;
      case 'F':show_table_gr(R_P_SPAN,MIO); break;
      default:printf("DispUp invalid"); getch();
      }
}//display_up

//**************************************************
void change_params(void) {
int which,hidi,lodi,c,c2; //hi-digit, low-digit
   which = hidi = lodi = 0;
   gotoxy(1,2);
   printf("IO  «l-g     «l-f         pow      thres     del");
   for(c=0; c<MIO; c++) {
      gotoxy(1,c+3);
      printf("%d   %4d®1%d  %1.3lf®2%d  %6.3lf®3%d  %1.3lf®4%d  %2d®5%d",
             c,HL_IOG[c],c,HL_IO[c],c,NT_PO[c],c,NT_TH[c],c,DELAY[c]+1,c);
      switch (c) {
         case 1:printf("   %1.3lf  moto®1",MOTOR_TH);   break;
         case 2:printf("   %3d   delay®2",MASTER_DELAY+1);break;
         case 3:printf("   %3d Spontan®3",Spontaneous);break;
         case 4:printf("     %d    equa®4",EQUA);       break;
         }
      }
   printf("   %s",STRUCFIL);
   printf("   %s",PARAMFIL);

   clean_line(1,1);
   which = ask_int("please type the digits after '®' ",1,57,0);
   hidi = (int) which / 10;
   lodi = which - (hidi*10);
   switch(which) {
      case 10: case 11: case 12: case 13: case 14: case 15: case 16: case 17:
         HL_IOG[lodi] = ask_int("NT/IOn «-life generations",0,9999,
                                HL_IOG[lodi]);
         HL_IO[lodi] = half_life(HL_IOG[lodi]);
         break;
      case 20: case 21: case 22: case 23: case 24: case 25: case 26: case 27:
         HL_IO[lodi] = ask_double("NT/IOn «-life factor",0.0,1.0,HL_IO[lodi]);
         break;
      case 30: case 31: case 32: case 33: case 34: case 35: case 36: case 37:
         NT_PO[lodi] = ask_double("NT/IOn power ",-1000.0,1000.0,
                                  NT_PO[lodi]);
         break;
      case 40: case 41: case 42: case 43: case 44: case 45: case 46: case 47:
         NT_TH[lodi] = ask_double("NT/IOn ThresHold ",0.0,1000.0,NT_TH[lodi]);
         break;
      case 50: case 51: case 52: case 53: case 54: case 55: case 56: case 57:
         DELAY[lodi] = ask_int("NT/IOn axon Delay  ",1,R_P_SPAN-1,DELAY[lodi]+1)-1;
         break;
      case  1:MOTOR_TH = ask_double("motor threshold",0.0,1000.0,MOTOR_TH);
              break;
      case  2:MASTER_DELAY = ask_int("master axon delay",
                                       1,R_P_SPAN-1,MASTER_DELAY+1)-1;

              for(c=2; c<MIO; c++) DELAY[c] = MASTER_DELAY;
              break;
      case  3:Spontaneous = ask_int("spontaneous firing(one in x chances)",
                                       0,999,Spontaneous);
              break;
      case  4:if(EQUA) {
                 EQUA = 0;
                 for(c=0; c<N_C; c++)
                    for(c2=0; c2<MIO; c2++)
                       SynEq[c][c2] = 1.0;
                 }
              else {EQUA = 1; setEqu();}
              break;
      default:clean_line(1,1);
              printf("DOES NOT COMPUTE");
              delay(1000);
      }
   menu_line_tx();
}//change_params

//**************************************************
void change_table(void) {
   int y,x;
   if(DispUp == 'P') {
      y = ask_int("          IO ",0,MIO , 0);
      if((y<0) || (y>MIO)) return;
      x = ask_int("          time step ",0,R_P_SPAN-1,0);
      if((x<0) || (y>=R_P_SPAN)) return;
      if(y<MIO) REF_PER[y][x] = ask_double("ion % crossing membran ",
                                           0.0,100.0,REF_PER[y][x]);
      if(y==MIO) REF_PER_CELL[x] = ask_double("cell's refractory period ",
                                               0.0,100.0,REF_PER_CELL[x]);
      }
   if(DispUp == 'F') {
      y = ask_int("          IO ",0,MIO-1 , 0);
      if((y<0) || (y>=MIO)) return;
      x = ask_int("          time step ",0,R_P_SPAN-1,0);
      if((x<0) || (y>=R_P_SPAN)) return;
      FIXDECAY[y][x] = ask_double("add/substract ion amount ",
                                  -9.9,9.9,FIXDECAY[y][x]);
      }
}//change_table


//**************************************************
void warm_reset(int noise) {
   char dum_str[80] = "\0";
   int c;
   clean_line(1,1);
   printf(" [RETURN] \"%s\"   [s] new seed   [space] exit",SEED);
   switch (getch()) {
      case ' ':break;
      case 13 :strcpy(SEED,"0123456789\0");
               reset(noise);
               motor_navigation('f');
               break;
      case 's':clean_line(1,1);
               printf("type seed >= %d characters ",RNDSZ);
               scanf("%s",&dum_str);
               if(strlen(dum_str)<RNDSZ) break;
               for(c=0; c<strlen(dum_str); c++)
                  if(dum_str[c] < 0)
                     goto do_nothing;
               strcpy(SEED,dum_str);
               gotoxy(50,1);
               printf("readed: %s ",dum_str);
               delay(1000);
               reset(noise);
               motor_navigation('f');
               break;
      }
   do_nothing: c++;///???
}//warm_reset

//**************************************************
long menu_gr(void) {
   int opoy,opox,dumch;
   while (1) {
      dumch = '-';
      menu_line_gr();
      display_up();
      dumch = getch();
      switch( dumch ) {
         case '0':return(1);           //1 Generation
         case '1':return(10);          //10   Gen.
         case '2':return(SHOW_GEN);    //one spike-plot screen
         case '3':return(2147483647l); //2147483647 biggest signed long int

         case 'C':if(getbkcolor() == 0) setbkcolor(LIGHTGRAY);
                  else setbkcolor(BLACK);//??undocumented, just for screengrabs
                  break;
         case 'n':plot_nei(1); break;
         case 'N':plot_nei(2); break;
         case 'w':plot_wei();
                  plot_World();
                  show_LMRG();
                  break;
         case 'b':DispUp = 'B'; break;
         case 'h':DispUp = 'H'; break;
         case 'e':show_cell(0); break;
         case 13 :switch (DispUp) { //13=ASCII for RETURN
                     case 'M':GrSetMode(GR_default_text);
                              change_params();
                              start_graphics();
                              plot_World();
                              show_LMRG();
                              break;
                     case 'P':GrSetMode(GR_default_text);
                              show_table(R_P_SPAN,MIO);
                              change_table();
                              start_graphics();
                              plot_World();
                              show_LMRG();
                  break;
            case 'F':GrSetMode(GR_default_text);
                  show_table(R_P_SPAN,MIO);
                  change_table();
                  start_graphics();
                  plot_World();
                  show_LMRG();
                  break;
            }
         break;
case 'm':GrFilledBox(0,0,800,WO_PO_Y-2,BLACK); DispUp = 'M'; break;
case 'p':GrFilledBox(0,0,800,WO_PO_Y-2,BLACK); DispUp = 'P'; break;
case 'f':GrFilledBox(0,0,800,WO_PO_Y-2,BLACK); DispUp = 'F'; break;
case 'R'://GrSetMode(GR_default_text); ??
         //warm_reset(1);
         reset(1);
         //start_graphics();
         plot_World();
         show_LMRG();
         break;
case 'r'://GrSetMode(GR_default_text);
         //warm_reset(0);
         reset(0);
         //start_graphics();
         plot_World();
         show_LMRG();
         break;
case 'v':GrSetMode(GR_default_text); //save parameters
         save_params(1,"params.txt");
         start_graphics();
         plot_World();
         show_LMRG();
         break;
case '+':motor_navigation('+'); break; //reinforce
case '-':motor_navigation('-'); break; //punish
case  0 :switch(getch()) {
            case  77:switch(Direc) {          //cursor right
                        case 'n':Direc = 'e'; break;
                        case 'e':Direc = 's'; break;
                        case 's':Direc = 'w'; break;
                        case 'w':Direc = 'n'; break;
                        }
                  plot_frog();
                  break;
            case  72:opoy = Frog_po_y; opox = Frog_po_x;
                  plot_n(WO_PO_X+((Frog_po_x-2)*WS),
                     WO_PO_Y+((Frog_po_y-2)*WS),
                     WS,WO_COL[World[Frog_po_y][Frog_po_x]]);
                  switch(Direc) {          //cursor up
                     case 'n':--Frog_po_y; break;
                     case 'e':++Frog_po_x; break;
                     case 's':++Frog_po_y; break;
                     case 'w':--Frog_po_x; break;
                     }
                  if( (Frog_po_y<2) || (Frog_po_y > WO_SZ_Y-3) ||
                      (Frog_po_x<2) || (Frog_po_x > WO_SZ_X-3) ) {
                      Frog_po_y = opoy;
                      Frog_po_x = opox;
                      }
                  plot_frog();
                  break;
            case  75:switch(Direc) {          //cursor left
                        case 'n':Direc = 'w'; break;
                        case 'e':Direc = 'n'; break;
                        case 's':Direc = 'e'; break;
```

```
                              case 'w':Direc = 's'; break;
                              }
                           plot_frog();
                           break;
                  case  80:opoy = Frog_po_y; opox = Frog_po_x;
                           plot_n(WO_PO_X+((Frog_po_x-2)*WS),
                              WO_PO_Y+((Frog_po_y-2)*WS),
                              WS,WO_COL[World[Frog_po_y][Frog_po_x]]);
                           switch(Direc) {          //cursor up
                              case 'n':++Frog_po_y; break;
                              case 'e':--Frog_po_x; break;
                              case 's':--Frog_po_y; break;
                              case 'w':++Frog_po_x; break;
                              }
                           if( (Frog_po_y<2) || (Frog_po_y > WO_SZ_Y-3) ||
                               (Frog_po_x<2) || (Frog_po_x > WO_SZ_X-3) ) {
                               Frog_po_y = opoy;
                               Frog_po_x = opox;
                               }
                           plot_frog();
                           break;
                  }
               break;
         case 'q':                //feed-up with program...
         case  27:GrSetMode(GR_default_text);
                  return(0);      //ascii 27 == ESC, terminate program
         }
      }
}//menu_gr
//menus.c last line
```

```
//hires.c first line

#include <conio.h>    //getch gotoxy
#include <math.h>     //log sqrt
#include <grx20.h>
#include <grxkeys.h>
#include "ion.h"

const int VGA = 1,
          FX= 8, //graphics-default-Font horizontal size in pixels
          FY=13; //idem vertical


int Rgb[3][MAXRBOWCOLORS];

//**************************************************
int start_graphics(void) {
const int RGB[3][MAXRBOWCOLORS] = //red, green, blue
/*
   setrgbpalette(pal.colors[ 0], 0, 0, 0);//blac
   setrgbpalette(pal.colors[ 1],63, 0, 0);//red
   setrgbpalette(pal.colors[ 2],63,27, 0);//rry
   setrgbpalette(pal.colors[ 3],63,39, 0);//ry
   setrgbpalette(pal.colors[ 4],63,51, 0);//ryy
   setrgbpalette(pal.colors[ 5],63,63, 0);//yell
   setrgbpalette(pal.colors[ 6],47,63, 0);//lemo
   setrgbpalette(pal.colors[ 7], 0,63, 0);//gree used as text-color :-(
   setrgbpalette(pal.colors[ 8], 0,63,63);//cyan
   setrgbpalette(pal.colors[ 9], 0,43,63);//ccb
   setrgbpalette(pal.colors[10], 0,26,63);//cbb
   setrgbpalette(pal.colors[11], 0, 0,63);//blue
   setrgbpalette(pal.colors[12],63, 0,63);//mage
   setrgbpalette(pal.colors[13],33,33,33);//dark
   setrgbpalette(pal.colors[14],53,53,53);//ligh
   setrgbpalette(pal.colors[15],63,63,63);//whit
*/
{{  0,255,255,255,255,255,190,  0,  0,  0,  0,  0,255,134,202,255},
 {  0,  0,109,148,206,255,255,255,255,174,105,  0,  0,134,202,255},
 {  0,  0,  0,  0,  0,  0,  0,  0,255,255,255,255,255,134,202,255}};
/*
{{255,255,255,255,255,255,255,208,144,  0,
    0,  0,  0,  0,  0,  0,  0,  0,  0,  0},

 {  0,  0,  0, 80,128,192,255,255,255,255,
  255,255,192,128, 80,  0,  0,  0,  0,  0},

 {255,112,  0,  0,  0,  0,  0,  0,  0,  0,
  112,255,255,255,255,255,176,128, 96, 64}};
*/
int c;
   GrSetDriver(NULL);
   if(GrCurrentVideoDriver() == NULL) {
      printf("No graphics driver found\n");
      getch();
      exit(1);
      }
   if(VGA) GrSetMode(GR_width_height_bpp_graphics,640,480,8);
   else GrSetMode(GR_width_height_bpp_graphics,800,600,8);
   for(c=0;c<MAXRBOWCOLORS;c++) {
      Rgb[0][c] = RGB[0][c];
      Rgb[1][c] = RGB[1][c];
      Rgb[2][c] = RGB[2][c];
      }
   for(c=0;c<MAXRBOWCOLORS;c++) {
      GrSetColor(c+56,Rgb[0][c],Rgb[1][c],Rgb[2][c]);
      RBOW[c] = c+56;
      }
   for(c=0; c<16; c++) //GrSetColor(c+72,128-(c-72),(c-72)*2,(c-72)*2);
      GrSetColor(c+ 72, 63+c*12, 63, 63);
   for(c=0; c<16; c++)
      GrSetColor(c+ 88,255, 63+c*12, 63);
   for(c=0; c<16; c++)
      GrSetColor(c+104,255-c*12,255, 63);
   for(c=0; c<16; c++)
      GrSetColor(c+120, 63,255, 63+c*12);
   for(c=0; c<16; c++)
      GrSetColor(c+136, 63,255-c*12,255);
   for(c=0; c<16; c++)
      GrSetColor(c+152, 63+c*12, 63+c*12,255);
   for(c=168; c<256; c++) GrSetColor(c,255,0,255);
   if(VGA) {
      GrLine(0,479,639,479,WHITE);
      GrLine(639,0,639,479,WHITE);
      WO_PO_Y = 200;
      }
   return(0);
}//start_graphics

//************************************************
//substitute calls by GrFilledBox ??
void plot_n(int x, int y, int size, int color) {
   int c,c2;
   for(c=0; c<size; c++)
      for(c2=0; c2<size; c2++)
         GrPlot(x+c2,y+c,color);
}//plot_n

//************************************************
void plot_4(int x, int y, int color) {
   if(GrPixel(x,y) == color) return;//check if plot needed
   GrPlot(x  ,y  ,color);
   GrPlot(x  ,y+1,color);
   GrPlot(x+1,y  ,color);
   GrPlot(x+1,y+1,color);
}//plot_4

//************************************************
void plot_depo(void) {
int y,color;
   if(TimePos == 0)
      plot_4(SHOW_GEN*2-2,WO_PO_Y-2,BLACK);
   else
      plot_4(TimePos*2-2,WO_PO_Y-2,BLACK);
   plot_4(TimePos*2,WO_PO_Y-2,WHITE);
   for(y=0; y<N_C; y++) {
      if(State[y]==0)                     //on depolarization...
         color = RBOW[WaveL[y]];
      else
         color = BLACK;                  //no depo: erase
      plot_4(TimePos*2,y*2+WO_PO_Y,color);
      }
}//plot_depo

//************************************************
void plot_waves(void) {
#define tRUNCATE 22
static int eeg[250][WAVES],//TimePos range: 0..249
           wave_pos[WAVES];
int c,c2,wave[WAVES],wc; // wc: waves counter
```

```
   if(eeg[0][0] == 0)  { //static vars set to 0 in declaration
      wave_pos[0] = getmaxy()-24;
      wave_pos[1] = getmaxy()-16;
      wave_pos[2] = getmaxy()-8;
      wave_pos[3] = getmaxy();
      for(c=0; c<250; c++)
         for(c2=0; c2<WAVES; eeg[c][c2] = wave_pos[c2], c2++);
      }
   for(wc=0; wc<WAVES; wc++) {
      if(EegB[wc] + EegE[wc] <= 0) wave[wc] = 0;
      else for(wave[wc]=0, c=EegB[wc]; c<=EegE[wc]; c++)
             if(State[c] == 0) wave[wc]++;
      if (wave[wc] > tRUNCATE) wave[wc] = tRUNCATE;
      }
   for(c=0; c<WAVES; c++) {
      GrLine(TimePos *2,   eeg[TimePos][c],        //erase old lines
             TimePos *2+2, eeg[TimePos+1][c],
             BLACK);
      eeg[TimePos][c] = wave_pos[c] - (wave[c] * 2); //compute new values
      GrLine(TimePos *2,  eeg[TimePos][c],//draw new lines
             TimePos *2-2,eeg[TimePos-1][c],
             NtCol[Type[EegB[c]]]);       //use color of first cell
      }
}//plot_waves

//**************************************************
void plot_nei(int what) {
   const double sCALE = 4.0;
   int c,c2,
       syn_size;

   if(what==1) {
      GrFilledBox(0,WO_PO_Y,(N_C*2)-1,WO_PO_Y+(N_C*2)-1,BLACK);
      for(c=0; c<N_C; c++) //plot synaptic position & type
         for(c2=0; c2<SynNum[c]; c2++)
            if( (SynType[c][c2] >= 0) &&
                (SynType[c][c2] <= MIO) &&
                (SynPos[c][c2] >= 0) )
               plot_4(SynPos[c][c2]*2,
                      c*2+WO_PO_Y,
                      NtCol[SynType[c][c2]]);
            else
               if(SynPos[c][c2] >= 0)
                  plot_4(SynPos[c][c2]*2,c*2+WO_PO_Y,WHITE);
      }
   if(what==2) {
      GrFilledBox(0,WO_PO_Y,(N_C*2)-1,WO_PO_Y+(N_C*2)-1,DARKGRAY);
      for(c=0; c<N_C; c++) //plot weights
         for(c2=0; c2<SynNum[c]; c2++){
            syn_size = (int) (SynWei[c][c2] * sCALE);
            if( (syn_size<MAXRBOWCOLORS) &&
                (SynType[c][c2] >= 0) &&
                (SynType[c][c2] <= MIO) &&
                (SynPos[c][c2] >= 0) )
               plot_4(SynPos[c][c2]*2,
                      c*2+WO_PO_Y,
                      RBOW[MAXRBOWCOLORS-syn_size-1]);
            else
               if(SynPos[c][c2] >= 0)
                  plot_4(SynPos[c][c2]*2,c*2+WO_PO_Y,WHITE);
         }
      }
}//plot_nei

//**************************************************
```

```
void plot_wei(void) {
const int tOP    =   0,
          dEFA   = 100, //position of default SynWei[][] (1.0)
          bOTTOM = 400,
          S      =   6; //s-1 = spacing pixels between cells
const double tRESHOLD = 0.5; //show syns over this weight

char dumline[81] = "X";
//"XXXXXXXXX1XXXXXXXXX2XXXXXXXXX3XXXXXXXXX4XXXXXXXXX5XXXXXXXXX6XXXXXXXXX7XXXXXXXXX8";
int cell = 0,c,c2;
   if(VGA) return;

   GrFilledBox(0,0,GrMaxX(),bOTTOM,BLACK);
   GrLine(0, (int)(log( 1.0)*dEFA+dEFA), N_C*S, (int)(log( 1.0)*dEFA+dEFA),WHITE);
   GrLine(0, (int)(log( 2.0)*dEFA+dEFA), N_C*S, (int)(log( 2.0)*dEFA+dEFA),WHITE);
   GrLine(0, (int)(log( 4.0)*dEFA+dEFA), N_C*S, (int)(log( 4.0)*dEFA+dEFA),WHITE);
   GrLine(0, (int)(log( 8.0)*dEFA+dEFA), N_C*S, (int)(log( 8.0)*dEFA+dEFA),WHITE);
   GrLine(0, (int)(log(16.0)*dEFA+dEFA), N_C*S, (int)(log(16.0)*dEFA+dEFA),WHITE);
   for(c=0; c<N_C; c++) {  //show all synapses of all cells
      plot_4(c*S,   tOP,NtCol[Type[c]]);
      plot_4(c*S,bOTTOM-5,NtCol[Type[c]]);
      for(c2=0; c2<SynNum[c]; c2++)
         if(SynPos[c][c2] >= 0) {  //plot every non-free_syn synapse
            if(SynWei[c][c2] < 16.0)
               GrPlot(SynPos[c][c2]*S,(int)(log(SynWei[c][c2])*dEFA+dEFA),
               NtCol[SynType[c][c2]] );
            else GrLine(c*S,tOP+3,SynPos[c][c2]*S,bOTTOM-5,WHITE);
            }
      }
//   for(c=0;c<25;c++)
//      for(c2=0; c2<80; c2++)
//         GrTextXY(c2*FX,c*FY,dumline , WHITE, BLACK);
   do { //show a single cell
      if(cell < 0) cell += N_C; //wrap borders
      else if(cell >= N_C) cell -= N_C;
//      gotoxy(76,1); printf("%3d",cell); //show number of cell..??
      sprintf(dumline,"cell:%3d",cell);
      GrTextXY(72*FX,0*FY,dumline , WHITE, BLACK);
      plot_4(cell*S,   tOP,WHITE);      //..and position
      plot_4(cell*S,bOTTOM-5,WHITE);
      for(c=0; c<SynNum[cell]; c++)
         if(SynPos[cell][c] >= 0) { //plot this cell's synapses
            plot_4(SynPos[cell][c]*S, bOTTOM-1, NtCol[SynType[cell][c]]);
            if(SynWei[cell][c] > tRESHOLD) {   //and a line for those grown
               if(SynWei[cell][c] < 16.0)
                  GrLine(cell*S, tOP+3,SynPos[cell][c] * S,
                        (int)(log(SynWei[cell][c]) * dEFA + dEFA),WHITE);
               else GrLine(cell*S,tOP+3,SynPos[cell][c]*S,bOTTOM-5,WHITE);
               }
            }
      if(getch() == 0) {
         plot_4(cell*S,   tOP,NtCol[Type[cell]]);
         plot_4(cell*S,bOTTOM-5,NtCol[Type[cell]]);
         for(c=0; c<SynNum[cell]; c++)
            if(SynPos[cell][c] >= 0) {
               plot_4(SynPos[cell][c]*S, bOTTOM-1, BLACK); //erase !grown syn.
               if(SynWei[cell][c] > tRESHOLD) {
                  if(SynWei[cell][c] < MAXRBOWCOLORS)
                     GrLine(cell*S,
                            tOP+3,
                            SynPos[cell][c] * S,
                            (int)(log(SynWei[cell][c]) * dEFA + dEFA),
                            RBOW[NtCol[SynType[cell][c]]] );
                  else GrLine(cell*S,
                              tOP+3,
                              SynPos[cell][c]*S,
                              bOTTOM-5,
                              NtCol[SynType[cell][c]]);//??
                     }
                  }
         switch(getch()) {
            case 72: cell -= 10; break;
            case 77: cell++;     break;
            case 80: cell += 10; break;
            case 75: cell--;     break;
            }
         }

      else break;
      }
      while (1);
   GrFilledBox(0,0,GrMaxX(),bOTTOM,BLACK);
}//plot_wei

//**********************************************
void plot_frog(void) {
int x = WO_PO_X + ((Frog_po_x-2)*WS),
    y = WO_PO_Y + ((Frog_po_y-2)*WS);
   GrFilledBox(x, y,x+WS-1,y+WS-1,WHITE); //"body"
   switch(Direc) {         //"eyes"
      case 'n':plot_4(   x+1,    y+1, DARKGRAY);
               plot_4(x+WS-3,    y+1, DARKGRAY);
               break;
      case 's':plot_4(   x+1, y+WS-3, DARKGRAY);
               plot_4(x+WS-3, y+WS-3, DARKGRAY);
               break;
      case 'e':plot_4(x+WS-3,    y+1, DARKGRAY);
               plot_4(x+WS-3, y+WS-3, DARKGRAY);
               break;
      case 'w':plot_4(   x+1,    y+1, DARKGRAY);
               plot_4(   x+1, y+WS-3, DARKGRAY);
               break;
      }
}//plot_frog

//**********************************************
void plot_World(void) {
int x,y;
   for(y=0; y<WO_SZ_Y-4; y++)
      for(x=0; x<WO_SZ_X-4; x++)
         GrFilledBox(WO_PO_X+(x*WS),
                     WO_PO_Y+(y*WS),
                     WO_PO_X+(x*WS)+WS-1,
                     WO_PO_Y+(y*WS)+WS-1,
                     WO_COL[World[y+2][x+2]]);
   plot_frog();
   for(x=0; x<MAXRBOWCOLORS; x++) //show spectrum,
      plot_n(WO_PO_X+(x*WS),WO_PO_Y+((WO_SZ_Y-3)*WS),WS,RBOW[x]);

   for(x=0; x<MIO; x++) //show NT colors
      plot_n(WO_PO_X+(x*WS),WO_PO_Y+((WO_SZ_Y-1)*WS),WS,NtCol[x]);
}//plot_World


//**********************************************
void show_synchro_tableVGA(void) {
int mAXY = 100;
char line[20] = "\0";
int ypos = mAXY+FY+FY+4,
    c,c2;
   sprintf(line,"ra:%7.3f",rate_accu_synch(0));
   GrTextXY(AREAS*60,mAXY+2,line,WHITE,DARKGRAY);
   mAXY += R_P_SPAN * 5 + 10;
   for(c=0; c<R_P_SPAN; c++)
      for(c2=0; c2<R_P_SPAN; c2++)
         if(His2D[c][c2]<183)
            GrFilledBox(c2*5+520,
                        mAXY-c*5,
                        c2*5+523,
                        mAXY-c*5+3,
                        72+(int)(His2D[c][c2]) );
//                        RBOW[(int)(His2D[c][c2])]);
//                        RBOW[(int)((His2D[c][c2] / Accumulated)*R_P_SPAN)]);
//                        RBOW[(int)(His2D[c][c2] / Accumulated)] );
         else
            GrFilledBox(c2*5+520,
                        mAXY-c*5,
                        c2*5+523,
                        mAXY-c*5+3,
                        255);
}//show_synchro_tableVGA


//**************************************************
void show_synchro_table(void) {
int mAXY = FY + N_C * 3;//GrMaxY();
char line[20] = "\0";
int ypos = mAXY+FY+FY+4,
    c,c2;
   if(VGA) {show_synchro_tableVGA(); return;}
   sprintf(line,"ra:%7.3f",rate_accu_synch(0));
   GrTextXY(AREAS*60,mAXY+2,line,WHITE,DARKGRAY);
   mAXY += R_P_SPAN * 5 + 10;
   for(c=0; c<R_P_SPAN; c++)
      for(c2=0; c2<R_P_SPAN; c2++)
         if(His2D[c][c2]<183)
            GrFilledBox(c2*5+720,
                        mAXY-c*5,
                        c2*5+723,
                        mAXY-c*5+3,
                        72+(int)(His2D[c][c2]) );
//                        RBOW[(int)(His2D[c][c2])]);
//                        RBOW[(int)((His2D[c][c2] / Accumulated)*R_P_SPAN)]);
//                        RBOW[(int)(His2D[c][c2] / Accumulated)] );
         else
            GrFilledBox(c2*5+720,
                        mAXY-c*5,
                        c2*5+723,
                        mAXY-c*5+3,
                        255);
}//show_synchro_table


//**************************************************
void synchro_stats(void) {
int mAXY = FY + N_C * 3;//GrMaxY();
char line[20] = "\0";
int his[R_P_SPAN][R_P_SPAN] = {{0,0,0,0,0,0,0,0,0,0,0,0,0,0}, //??
                               {0,0,0,0,0,0,0,0,0,0,0,0,0,0},
                               {0,0,0,0,0,0,0,0,0,0,0,0,0,0},
                               {0,0,0,0,0,0,0,0,0,0,0,0,0,0},
                               {0,0,0,0,0,0,0,0,0,0,0,0,0,0},
                               {0,0,0,0,0,0,0,0,0,0,0,0,0,0},
                               {0,0,0,0,0,0,0,0,0,0,0,0,0,0},
                               {0,0,0,0,0,0,0,0,0,0,0,0,0,0},
                                {0,0,0,0,0,0,0,0,0,0,0,0,0},
                                {0,0,0,0,0,0,0,0,0,0,0,0,0},
                                {0,0,0,0,0,0,0,0,0,0,0,0,0},
                                {0,0,0,0,0,0,0,0,0,0,0,0,0},
                                {0,0,0,0,0,0,0,0,0,0,0,0,0}},
   big1,big2,
   ypos = mAXY+FY+FY+4,
   c,c2;

   if(VGA) {show_synchro_table(); return;}

   GrFilledBox(0,mAXY+2,GrMaxX(),mAXY+80,DARKGRAY);
   for(c=0; c<256; c++) GrFilledBox(c*3,mAXY-1,c*3+3,mAXY+1,c);
   for(c=0; c<AREAS; c++) {

      for(c2=AREA_B[c]; c2<AREA_E[c]; c2++)
         his[c][State[c2]]++;
      for(c2=0; c2<R_P_SPAN; c2++) {
         his[c][c2] *= 4;   //show it big...
         GrLine(c2*4+c*60+2,
                ypos,
                c2*4+c*60+2,
                ypos+his[c][c2],
                //RBOW[c2]);
                LIGHTGRAY);
         GrLine(c2*4+c*60+3,
                ypos,
                c2*4+c*60+3,
                ypos+his[c][c2],
                //RBOW[c2]);
                LIGHTGRAY);
         his[c][c2] /= 4;   //return to original size
         }
      his[c][12] = 0; //state 12 is minimum activity, ignore it
      big1 = 12; big2 = 12;
      for(c2=0; c2<R_P_SPAN; c2++) //choose biggest two
         if(his[c][c2] > his[c][big1]) { big2 = big1; big1 = c2; }
         else if(his[c][c2] > his[c][big2]) big2 = c2;
      sprintf(line,"%2d %2d",big1,big2);
      GrTextXY(c*60,mAXY+2,line,WHITE,DARKGRAY);
      sprintf(line,"%2d %2d",his[c][big1],his[c][big2]);
      GrTextXY(c*60,mAXY+FY+2,line,WHITE,DARKGRAY);
      }
   show_synchro_table();
}//synchro_stats


//**********************************************
void histogram(int full) {
const int mAXY = FY + N_C * 3;//GrMaxY();
char line[20] = "\0";
int his[AREAS][R_P_SPAN] = {{0,0,0,0,0,0,0,0,0,0,0,0,0}, //??
                             {0,0,0,0,0,0,0,0,0,0,0,0,0},
                             {0,0,0,0,0,0,0,0,0,0,0,0,0},
                             {0,0,0,0,0,0,0,0,0,0,0,0,0},
                             {0,0,0,0,0,0,0,0,0,0,0,0,0},
                             {0,0,0,0,0,0,0,0,0,0,0,0,0},
                             {0,0,0,0,0,0,0,0,0,0,0,0,0},
                             {0,0,0,0,0,0,0,0,0,0,0,0,0},
                             {0,0,0,0,0,0,0,0,0,0,0,0,0},
                             {0,0,0,0,0,0,0,0,0,0,0,0,0},
                             {0,0,0,0,0,0,0,0,0,0,0,0,0},
                             {0,0,0,0,0,0,0,0,0,0,0,0,0}},

    c,c2;
   if(VGA) return;
   if(full) {
      GrFilledBox(0,mAXY-180,GrMaxX(),mAXY,DARKGRAY);
      for(c=0; c<AREAS; c++) {
         sprintf(line,"%s",AREA_NAME[c]);
         GrTextXY(c*60,mAXY-FY,line,WHITE,DARKGRAY);
         for(c2=AREA_B[c]; c2<AREA_E[c]; c2++)
            his[c][WaveL[c2]]++;
         for(c2=0; c2<R_P_SPAN; c2++) {
            his[c][c2] *= 4; //make it bigger
            GrLine(c2*4+c*60+2,
                   mAXY-FY,
                   c2*4+c*60+2,
                   mAXY-his[c][c2]-FY,
                   RBOW[c2]);
            GrLine(c2*4+c*60+3,
                   mAXY-FY,
                   c2*4+c*60+3,
                   mAXY-his[c][c2]-FY,
                   RBOW[c2]);
            }
         }
      for(c=0; c<AREAS; c++) his[0][c] = 0;
      sprintf(line,"total",AREA_NAME[c]);
      GrTextXY(740,mAXY-FY,line,WHITE,DARKGRAY);
      }

   for(c=0; c<N_C; c++)
      his[0][WaveL[c]]++;

   for(c=0; c<R_P_SPAN; c++) {
      GrLine(c*4+740+2,
             mAXY-FY,
             c*4+740+2,
             mAXY-his[0][c]-FY,
             RBOW[c]);

      GrLine(c*4+740+3,
             mAXY-FY,
             c*4+740+3,
             mAXY-his[0][c]-FY,
             RBOW[c]);
      }
   GrFilledBox(796,596,800,600,WHITE);
}//histogram


//************************************************
void show_LMRG(void) { //show Left Move Right Generations
char line[81] = "\0";
   sprintf(line,"%6.4lf %6.4lf %6.4lf",Left,Move,Right);
   GrTextXY(WO_PO_X,WO_PO_Y+(WO_SZ_Y*WS)+FY,line,WHITE,BLACK);
   sprintf(line,"   %9ld",Gen);
   GrTextXY(WO_PO_X,WO_PO_Y+(WO_SZ_Y*WS)+(FY*2),line,WHITE,BLACK);
} //show_LMRG


//************************************************
void bars(void) {
   const int y_MARGIN = FY,
             ySZ = 3;      //Y-space between bars, in pixels
   const float cAL[4] = {0.7071,1.0,1.4142,2.0},//sqrt(0.5 1.0 2.0 4.0)
               xSZ = 38.0; //X expansion factor
   int level[MIO] = {  0, 80,160,240,320,400,480,560},
       levelT = 720,   //for TotalIo
       y_tot,
       c,c2;

   GrFilledBox(0,y_MARGIN,GrMaxX(),y_MARGIN+N_C*ySZ,BLACK);
   for(c=0; c<MIO; c++) { //calibration marks for IO[n]
      GrLine(level[c]+(int)(cAL[0]*xSZ), y_MARGIN,
             level[c]+(int)(cAL[0]*xSZ), y_MARGIN+10,WHITE);
      GrLine(level[c]+(int)(cAL[1]*xSZ), y_MARGIN,
             level[c]+(int)(cAL[1]*xSZ), y_MARGIN+15,WHITE);
      GrLine(level[c]+(int)(cAL[2]*xSZ), y_MARGIN,
             level[c]+(int)(cAL[2]*xSZ), y_MARGIN+10,WHITE);
      GrLine(level[c]+(int)(cAL[3]*xSZ), y_MARGIN,
             level[c]+(int)(cAL[3]*xSZ), y_MARGIN+10,WHITE);
      }
   GrLine(levelT, y_MARGIN,levelT, y_MARGIN+15,LIGHTMAGENTA);//for totals
   GrLine(levelT+(int)(cAL[0]*xSZ), y_MARGIN,
          levelT+(int)(cAL[0]*xSZ), y_MARGIN+10,LIGHTMAGENTA);
   GrLine(levelT+(int)(cAL[1]*xSZ), y_MARGIN,
          levelT+(int)(cAL[1]*xSZ), y_MARGIN+15,LIGHTMAGENTA);
   GrLine(levelT+(int)(cAL[2]*xSZ), y_MARGIN,
          levelT+(int)(cAL[2]*xSZ), y_MARGIN+10,LIGHTMAGENTA);
   GrLine(levelT+(int)(cAL[3]*xSZ), y_MARGIN,
          levelT+(int)(cAL[3]*xSZ), y_MARGIN+10,LIGHTMAGENTA);
   GrLine(levelT-(int)(cAL[0]*xSZ), y_MARGIN,
          levelT-(int)(cAL[0]*xSZ), y_MARGIN+10,LIGHTMAGENTA);
   GrLine(levelT-(int)(cAL[1]*xSZ), y_MARGIN,
          levelT-(int)(cAL[1]*xSZ), y_MARGIN+15,LIGHTMAGENTA);
   GrLine(levelT-(int)(cAL[2]*xSZ), y_MARGIN,
          levelT-(int)(cAL[2]*xSZ), y_MARGIN+10,LIGHTMAGENTA);
   GrLine(levelT-(int)(cAL[3]*xSZ), y_MARGIN,
          levelT-(int)(cAL[3]*xSZ), y_MARGIN+10,LIGHTMAGENTA);
   for(c=0; c<N_C; c++) {
      for(c2=MIO-1; c2>=0; --c2)
         GrLine( level[c2],
            c * ySZ + y_MARGIN,
            level[c2]+(int)(sqrt(Cell_Io[c][c2]*SynEq[c][c2])*xSZ),
            c * ySZ + y_MARGIN,NtCol[c2]);
      y_tot = c * ySZ + y_MARGIN;
      if(TotalIo[c] >= 0.0) {
         GrLine( levelT,
            y_tot,
            levelT+(int)(sqrt(TotalIo[c])*xSZ),
            y_tot,LIGHTGRAY);
         GrPlot(levelT+(int)(sqrt(TotalIo[c]*REF_PER_CELL[State[c]])*xSZ),
               y_tot+1,
               WHITE);
         }
      else {
         GrLine( levelT,
            y_tot,
            levelT-(int)(sqrt( fabs (TotalIo[c]) )*xSZ),
            y_tot,LIGHTGRAY);
         GrPlot(levelT-(int)(sqrt(fabs(TotalIo[c])*REF_PER_CELL[State[c]])*xSZ),
               y_tot,WHITE);
         }
      }
}//bars


void restore_text_mode(void) {
   closegraph();
}

//hires.c last line
```

```
//gene.c first line

#include "ion.h"
#include <bios.h> //bioskey()
//population, number of organisms/genomes
#define POPU 2000

FILE *gene_output;

struct organism Orga[POPU+1]; //organisms, Orga[POPU] or...
struct organism Baby;         //...Baby is the newborn

long His2D[R_P_SPAN][R_P_SPAN],  //Histogram 2-dimensional used by hires.c
     His2Ds[R_P_SPAN][R_P_SPAN]; //idem, only for spikes

int RATE_SYNCH[R_P_SPAN][R_P_SPAN] = {
//0  1  2  3  4  5  6  7  8  9 10 11 12 Wavelenght
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 0 state

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 1

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 2

{ 0, 0, 0, 0, 2, 4, 4, 0, 0, 0, 0, 0, 0}, // 3

{ 0, 0, 0, 2, 0, 4, 4, 4, 0, 0, 0, 0, 0}, // 4

{ 0, 0, 0, 4, 4, 0, 4, 4, 4, 0, 0, 0, 0}, // 5

{ 0, 0, 0, 4, 4, 4, 0, 4, 4, 4, 0, 0, 0}, // 6

{ 0, 0, 0, 0, 4, 4, 4, 0, 4, 4, 4, 0, 0}, // 7

{ 0, 0, 0, 0, 0, 4, 4, 4, 0, 4, 4, 4, 0}, // 8

{ 0, 0, 0, 0, 0, 0, 4, 4, 4, 0, 4, 4, 0}, // 9

{ 0, 0, 0, 0, 0, 0, 0, 4, 4, 4, 0, 2, 0}, //10

{ 0, 0, 0, 0, 0, 0, 0, 0, 4, 4, 2, 0, 0}, //11

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0}};//12

int RATESYNCHCAP = N_C * SHOW_GEN;


long Accumulated = 0;

//int SEE = 0; //to avoid printf in hires

const int HL_IOG0[15] = {12,18,24,30,36,42,48,54,60,66,72,84,96,108,120},
          HL_IOG2[15] = { 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,14,16,18,20};

const double REF_PER_GEN[25][R_P_SPAN] =
   {{1.00,1.00,0.98,0.86,0.74,0.62,0.50,0.38,0.26,0.14,0.02,0.00,0.00},
    {1.00,1.00,0.94,0.83,0.72,0.61,0.50,0.39,0.28,0.17,0.06,0.00,0.00},
    {1.00,1.00,0.90,0.80,0.70,0.60,0.50,0.40,0.30,0.20,0.10,0.00,0.00},
    {1.00,0.95,0.86,0.77,0.68,0.59,0.50,0.41,0.32,0.23,0.14,0.05,0.00},
    {0.98,0.90,0.82,0.74,0.66,0.58,0.50,0.42,0.34,0.26,0.18,0.10,0.02},
    {0.92,0.85,0.78,0.71,0.64,0.57,0.50,0.43,0.36,0.29,0.22,0.15,0.08},
    {0.86,0.80,0.74,0.68,0.62,0.56,0.50,0.44,0.38,0.32,0.26,0.20,0.14},
    {0.80,0.75,0.70,0.65,0.60,0.55,0.50,0.45,0.40,0.35,0.30,0.25,0.20},
    {0.74,0.70,0.66,0.62,0.58,0.54,0.50,0.46,0.42,0.38,0.34,0.30,0.26},
    {0.68,0.65,0.62,0.59,0.56,0.53,0.50,0.47,0.44,0.41,0.38,0.35,0.32},
    {0.62,0.60,0.58,0.56,0.54,0.52,0.50,0.48,0.46,0.44,0.42,0.40,0.38},
    {0.56,0.55,0.54,0.53,0.52,0.51,0.50,0.49,0.48,0.47,0.46,0.45,0.44},
    {0.50,0.50,0.50,0.50,0.50,0.50,0.50,0.50,0.50,0.50,0.50,0.50,0.50},
    {0.44,0.45,0.46,0.47,0.48,0.49,0.50,0.51,0.52,0.53,0.54,0.55,0.56},
    {0.38,0.40,0.42,0.44,0.46,0.48,0.50,0.52,0.54,0.56,0.58,0.60,0.62},
    {0.32,0.35,0.38,0.41,0.44,0.47,0.50,0.53,0.56,0.59,0.62,0.65,0.68},
    {0.26,0.30,0.34,0.38,0.42,0.46,0.50,0.54,0.58,0.62,0.66,0.70,0.74},
    {0.20,0.25,0.30,0.35,0.40,0.45,0.50,0.55,0.60,0.65,0.70,0.75,0.80},
    {0.14,0.20,0.26,0.32,0.38,0.44,0.50,0.56,0.62,0.68,0.74,0.80,0.86},
    {0.08,0.15,0.22,0.29,0.36,0.43,0.50,0.57,0.64,0.71,0.78,0.85,0.92},
    {0.02,0.10,0.18,0.26,0.34,0.42,0.50,0.58,0.66,0.74,0.82,0.90,0.98},
    {0.00,0.05,0.14,0.23,0.32,0.41,0.50,0.59,0.68,0.77,0.86,0.95,1.00},
    {0.00,0.00,0.10,0.20,0.30,0.40,0.50,0.60,0.70,0.80,0.90,1.00,1.00},
    {0.00,0.00,0.06,0.17,0.28,0.39,0.50,0.61,0.72,0.83,0.94,1.00,1.00},
    {0.00,0.00,0.02,0.14,0.26,0.38,0.50,0.62,0.74,0.86,0.98,1.00,1.00}};

const double REF_PER_CELL_GEN[63][R_P_SPAN] = {
{0.000,0.500,1.000,1.000,1.000,1.000,1.000,1.000,1.000,1.000,1.000,1.000,1.0},
{0.000,0.333,0.667,1.000,1.000,1.000,1.000,1.000,1.000,1.000,1.000,1.000,1.0},
{0.000,0.250,0.500,0.750,1.000,1.000,1.000,1.000,1.000,1.000,1.000,1.000,1.0},
{0.000,0.200,0.400,0.600,0.800,1.000,1.000,1.000,1.000,1.000,1.000,1.000,1.0},
{0.000,0.167,0.333,0.500,0.667,0.833,1.000,1.000,1.000,1.000,1.000,1.000,1.0},
{0.000,0.143,0.286,0.429,0.571,0.714,0.857,1.000,1.000,1.000,1.000,1.000,1.0},
{0.000,0.125,0.250,0.375,0.500,0.625,0.750,0.875,1.000,1.000,1.000,1.000,1.0},
{0.000,0.111,0.222,0.333,0.444,0.556,0.667,0.778,0.889,1.000,1.000,1.000,1.0},
{0.000,0.100,0.200,0.300,0.400,0.500,0.600,0.700,0.800,0.900,1.000,1.000,1.0},

{0.000,0.000,0.500,1.000,1.000,1.000,1.000,1.000,1.000,1.000,1.000,1.000,1.0},
{0.000,0.000,0.333,0.667,1.000,1.000,1.000,1.000,1.000,1.000,1.000,1.000,1.0},
{0.000,0.000,0.250,0.500,0.750,1.000,1.000,1.000,1.000,1.000,1.000,1.000,1.0},
{0.000,0.000,0.200,0.400,0.600,0.800,1.000,1.000,1.000,1.000,1.000,1.000,1.0},
{0.000,0.000,0.167,0.333,0.500,0.667,0.833,1.000,1.000,1.000,1.000,1.000,1.0},
{0.000,0.000,0.143,0.286,0.429,0.571,0.714,0.857,1.000,1.000,1.000,1.000,1.0},
{0.000,0.000,0.125,0.250,0.375,0.500,0.625,0.750,0.875,1.000,1.000,1.000,1.0},
{0.000,0.000,0.111,0.222,0.333,0.444,0.556,0.667,0.778,0.889,1.000,1.000,1.0},
{0.000,0.000,0.100,0.200,0.300,0.400,0.500,0.600,0.700,0.800,0.900,1.000,1.0},

{0.000,0.000,0.000,0.500,1.000,1.000,1.000,1.000,1.000,1.000,1.000,1.000,1.0},
{0.000,0.000,0.000,0.333,0.667,1.000,1.000,1.000,1.000,1.000,1.000,1.000,1.0},
{0.000,0.000,0.000,0.250,0.500,0.750,1.000,1.000,1.000,1.000,1.000,1.000,1.0},
{0.000,0.000,0.000,0.200,0.400,0.600,0.800,1.000,1.000,1.000,1.000,1.000,1.0},
{0.000,0.000,0.000,0.167,0.333,0.500,0.667,0.833,1.000,1.000,1.000,1.000,1.0},
{0.000,0.000,0.000,0.143,0.286,0.429,0.571,0.714,0.857,1.000,1.000,1.000,1.0},
{0.000,0.000,0.000,0.125,0.250,0.375,0.500,0.625,0.750,0.875,1.000,1.000,1.0},
{0.000,0.000,0.000,0.111,0.222,0.333,0.444,0.556,0.667,0.778,0.889,1.000,1.0},
{0.000,0.000,0.000,0.100,0.200,0.300,0.400,0.500,0.600,0.700,0.800,0.900,1.0},

{0.000,0.000,0.000,0.000,0.500,1.000,1.000,1.000,1.000,1.000,1.000,1.000,1.0},
{0.000,0.000,0.000,0.000,0.333,0.667,1.000,1.000,1.000,1.000,1.000,1.000,1.0},
{0.000,0.000,0.000,0.000,0.250,0.500,0.750,1.000,1.000,1.000,1.000,1.000,1.0},
{0.000,0.000,0.000,0.000,0.200,0.400,0.600,0.800,1.000,1.000,1.000,1.000,1.0},
{0.000,0.000,0.000,0.000,0.167,0.333,0.500,0.667,0.833,1.000,1.000,1.000,1.0},
{0.000,0.000,0.000,0.000,0.143,0.286,0.429,0.571,0.714,0.857,1.000,1.000,1.0},
{0.000,0.000,0.000,0.000,0.125,0.250,0.375,0.500,0.625,0.750,0.875,1.000,1.0},
{0.000,0.000,0.000,0.000,0.111,0.222,0.333,0.444,0.556,0.667,0.778,0.889,1.0},

{0.000,0.000,0.000,0.000,0.000,0.500,1.000,1.000,1.000,1.000,1.000,1.000,1.0},
{0.000,0.000,0.000,0.000,0.000,0.333,0.667,1.000,1.000,1.000,1.000,1.000,1.0},
{0.000,0.000,0.000,0.000,0.000,0.250,0.500,0.750,1.000,1.000,1.000,1.000,1.0},
{0.000,0.000,0.000,0.000,0.000,0.200,0.400,0.600,0.800,1.000,1.000,1.000,1.0},
{0.000,0.000,0.000,0.000,0.000,0.167,0.333,0.500,0.667,0.833,1.000,1.000,1.0},
{0.000,0.000,0.000,0.000,0.000,0.143,0.286,0.429,0.571,0.714,0.857,1.000,1.0},
{0.000,0.000,0.000,0.000,0.000,0.125,0.250,0.375,0.500,0.625,0.750,0.875,1.0},

{0.000,0.000,0.000,0.000,0.000,0.000,0.500,1.000,1.000,1.000,1.000,1.000,1.0},
{0.000,0.000,0.000,0.000,0.000,0.000,0.333,0.667,1.000,1.000,1.000,1.000,1.0},
{0.000,0.000,0.000,0.000,0.000,0.000,0.250,0.500,0.750,1.000,1.000,1.000,1.0},
{0.000,0.000,0.000,0.000,0.000,0.000,0.200,0.400,0.600,0.800,1.000,1.000,1.0},
{0.000,0.000,0.000,0.000,0.000,0.000,0.167,0.333,0.500,0.667,0.833,1.000,1.0},
{0.000,0.000,0.000,0.000,0.000,0.000,0.143,0.286,0.429,0.571,0.714,0.857,1.0},

{0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.500,1.000,1.000,1.000,1.000,1.0},
{0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.333,0.667,1.000,1.000,1.000,1.0},
{0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.250,0.500,0.750,1.000,1.000,1.0},
{0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.200,0.400,0.600,0.800,1.000,1.0},
{0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.167,0.333,0.500,0.667,0.833,1.0},

{0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.500,1.000,1.000,1.000,1.0},
{0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.333,0.667,1.000,1.000,1.0},
{0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.250,0.500,0.750,1.000,1.0},
{0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.200,0.400,0.600,0.800,1.0},

{0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.500,1.000,1.000,1.0},
{0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.333,0.667,1.000,1.0},
{0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.250,0.500,0.750,1.0},

{0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.500,1.000,1.0},
{0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.333,0.667,1.0},

{0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.500,1.0}};

// 0.163507991, 0.179858790, 0.197844669, 0.217629136, 0.239392049,
// 0.263331254, 0.289664380, 0.318630818, 0.350493899, 0.385543289,
// 0.424097618, 0.466507380, 0.513158118, 0.564473930, 0.620921323,
// 0.683013455, 0.751314801, 0.826446281, 0.909090909, 1.000000000,
// 1.100000000, 1.210000000, 1.331000000, 1.464100000, 1.610510000,
// 1.771561000, 1.948717100, 2.143588810, 2.357947691, 2.593742460,
// 2.853116706, 3.138428377, 3.452271214, 3.797498336, 4.177248169,
// 4.594972986, 5.054470285, 5.559917313, 6.115909045,    *= 1.1

#define LOG_RANGE 95

const double LOG_05[LOG_RANGE] = {
0.01019074,0.01070028,0.01123530,0.01179706,0.01238691,
0.01300626,0.01365657,0.01433940,0.01505637,0.01580919,
0.01659965,0.01742963,0.01830111,0.01921617,0.02017698,
0.02118582,0.02224512,0.02335737,0.02452524,0.02575150,
0.02703908,0.02839103,0.02981058,0.03130111,0.03286617,
0.03450948,0.03623495,0.03804670,0.03994903,0.04194648,
0.04404381,0.04624600,0.04855830,0.05098621,0.05353552,
0.05621230,0.05902291,0.06197406,0.06507276,0.06832640,
0.07174272,0.07532986,0.07909635,0.08305117,0.08720373,
0.09156391,0.09614211,0.10094921,0.10599668,0.11129651,
0.11686133,0.12270440,0.12883962,0.13528160,0.14204568,
0.14914797,0.15660536,0.16443563,0.17265741,0.18129029,
0.19035480,0.19987254,0.20986617,0.22035947,0.23137745,
0.24294632,0.25509364,0.26784832,0.28124073,0.29530277,
0.31006791,0.32557131,0.34184987,0.35894236,0.37688948,
0.39573396,0.41552065,0.43629669,0.45811152,0.48101710,
0.50506795,0.53032135,0.55683742,0.58467929,0.61391325,
0.64460892,0.67683936,0.71068133,0.74621540,0.78352617,
0.82270247,0.86383760,0.90702948,0.95238095,1.00000000};

//*********************** variables global to gene.c ******************

void init_orga(void) {
int c;
   for(c=0; c<=POPU; c++) { //include newborn one
      Orga[c].hl_iog0     = 0;
      Orga[c].hl_iog2     = 0;
      Orga[c].fixdec00    = 0;
      Orga[c].fixdec0rest = 0;
      Orga[c].fixdec2all  = 0;
      Orga[c].nt_po0      = 0;
      Orga[c].nt_po2      = 0;
      Orga[c].refper0     = 0;
      Orga[c].refper2     = 0;
      Orga[c].refpercell  = 0;
      Orga[c].delay2      = 0;
      Orga[c].fitness = 1.0;
      strcpy(Orga[c].name,"0123456789012345678\0");
      }
}//init_orga


void baptice(struct organism *doe) {
//must be longer than range of parameters
const char *cODE =
"0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ\
!#$&'()*+,-./:;<=>?@[]^_`{|}~□€□,ƒ„…†‡ˆ‰Š‹Œ□Ž□\0";
   doe->name[ 0] = cODE[doe->hl_iog0];
   doe->name[ 1] = cODE[doe->hl_iog2];
   doe->name[ 2] = cODE[doe->fixdec00];
   doe->name[ 3] = cODE[doe->fixdec0rest];
   doe->name[ 4] = cODE[doe->fixdec2all];
   doe->name[ 5] = cODE[doe->nt_po0];
   doe->name[ 6] = cODE[doe->nt_po2];
   doe->name[ 7] = cODE[doe->refper0];
   doe->name[ 8] = cODE[doe->refper2];
   doe->name[ 9] = cODE[doe->refpercell];
   doe->name[10] = cODE[doe->delay2];
   doe->name[11] = '\0';
}//baptice


void set_params(struct organism *doe) {
int c;
   HL_IOG[0] = HL_IOG0[doe->hl_iog0];
   HL_IOG[2] = HL_IOG2[doe->hl_iog2];
   FIXDECAY[0][0] = LOG_05[doe->fixdec00] * -1.0;
   for(c=1; c<R_P_SPAN; c++)
      FIXDECAY[0][c] = LOG_05[doe->fixdec0rest];
   for(c=0; c<R_P_SPAN; c++)
      FIXDECAY[2][c] = LOG_05[doe->fixdec2all];
   IO_PO[0]  = LOG_05[doe->nt_po0];
   IO_PO[2]  = LOG_05[doe->nt_po2];

   for(c=0; c<R_P_SPAN; c++)
      REF_PER[0][c] = REF_PER_GEN[doe->refper0][c];

   for(c=0; c<R_P_SPAN; c++)
      REF_PER[2][c] = REF_PER_GEN[doe->refper2][c];

   for(c=0; c<R_P_SPAN; c++)
      REF_PER_CELL[c] = REF_PER_CELL_GEN[doe->refpercell][c];

   DELAY[2]  = doe->delay2;
   doe->fitness = 1.0;
}//set_params


void copy_orga(int target, int source) {
   Orga[target].hl_iog0 = Orga[source].hl_iog0;
   Orga[target].hl_iog2 = Orga[source].hl_iog2;
   Orga[target].fixdec00 = Orga[source].fixdec00;
   Orga[target].fixdec0rest = Orga[source].fixdec0rest;
   Orga[target].fixdec2all = Orga[source].fixdec2all;

   Orga[target].nt_po0  = Orga[source].nt_po0;
   Orga[target].nt_po2  = Orga[source].nt_po2;

   Orga[target].refper0 = Orga[source].refper0;
   Orga[target].refper2 = Orga[source].refper2;
   Orga[target].refpercell = Orga[source].refpercell;

   Orga[target].delay2  = Orga[source].delay2;

   Orga[target].fitness = Orga[source].fitness;
   strcpy(Orga[target].name,Orga[source].name);
}//copy_orga


void dump_orga(struct organism which_orga, int orga_num) {
char line[256] =
"---------1---------2---------3---------4---------5---------6---------7\0";

   sprintf(line,"\n%4d:%2d %2d %2d %2d %2d %2d %2d %2d %2d %2d %2d %7.4f %s",
           orga_num,
           which_orga.hl_iog0,
           which_orga.hl_iog2,
           which_orga.fixdec00,
           which_orga.fixdec0rest,
           which_orga.fixdec2all,
           which_orga.nt_po0,
           which_orga.nt_po2,
           which_orga.refper0,
           which_orga.refper2,
           which_orga.refpercell,
           which_orga.delay2,
           which_orga.fitness,
           which_orga.name);
   fprintf(gene_output,line);
   if(!SEE) printf("%s",line);
}//dump_orga


void dump_popu(void) {
int c;
   for(c=0; c<POPU; c++)
      dump_orga(Orga[c],c);
   fprintf(gene_output,"\n\n");
   if(!SEE) printf("\n");
}//dump_popu


int clone_test(struct organism *doe) {
int c;
   for(c=0; c<POPU; c++)
      if(strcmp(Orga[c].name,doe->name) == 0)
         return(1);
   return(0);
}//clone_test


void mutate(struct organism *doe) {
   switch(random() % 11) {
      case  0: doe->hl_iog0 = random() % 15; break;
      case  1: doe->hl_iog2 = random() % 15; break;
      case  2: doe->fixdec00 = random() % LOG_RANGE; break;
      case  3: doe->fixdec0rest = random() % (doe->fixdec00 +1); break; //***
      case  4: doe->fixdec2all = random() % LOG_RANGE; break;
      case  5: doe->nt_po0  = random() % LOG_RANGE; break;
      case  6: doe->nt_po2  = random() %  LOG_RANGE; break;
      case  7: doe->refper0 = random() % 25; break;
      case  8: doe->refper2 = random() % 25; break;
      case  9: doe->refpercell = random() % 63; break;
      case 10: doe->delay2  = (random() % (R_P_SPAN-1)) + 1; break;
      }
   switch(random() % 11) {
      case  0: doe->hl_iog0 = random() % 15; break;
      case  1: doe->hl_iog2 = random() % 15; break;
      case  2: doe->fixdec00 = random() % LOG_RANGE; break;
      case  3: doe->fixdec0rest = random() % (doe->fixdec00 +1); break; //***
      case  4: doe->fixdec2all = random() % LOG_RANGE; break;
      case  5: doe->nt_po0  = random() % LOG_RANGE; break;
      case  6: doe->nt_po2  = random() %  LOG_RANGE; break;
      case  7: doe->refper0 = random() % 25; break;
      case  8: doe->refper2 = random() % 25; break;
      case  9: doe->refpercell = random() % 63; break;
      case 10: doe->delay2  = (random() % (R_P_SPAN-1)) + 1; break;
      }
}//mutate


void seed_popu(void) {
int c;
   for(c=0; c<POPU; c++) {
      Orga[c].hl_iog0 = random() % 15;
      Orga[c].hl_iog2 = random() % 15;
      Orga[c].fixdec00 = random() % LOG_RANGE;
      Orga[c].fixdec0rest = random() % (Orga[c].fixdec00+1); //***
      Orga[c].fixdec2all = random() % LOG_RANGE;
      Orga[c].nt_po0  = random() % LOG_RANGE;
      Orga[c].nt_po2  = random() % LOG_RANGE;
      Orga[c].refper0 = random() % 25;
      Orga[c].refper2 = random() % 25;
      Orga[c].refpercell = random() % 63;
      Orga[c].delay2  = (random() % (R_P_SPAN-1)) + 1;
      Orga[c].fitness = 1.0;
      baptice(&Orga[c]);
      }
}//seed_popu


char * sex(int woman, int man) {
//positive==looks like mami, neg.like papi 0==perfect mix
char *mix = "??????????????????????\0";
int looks_like = 42;
   while((looks_like < -3) || (looks_like > 3)) { //force 2 swaps
      looks_like = 0;
      if(random()%2) { Baby.hl_iog0 = Orga[woman].hl_iog0;
         mix[0]='+'; ++looks_like; }
      else {           Baby.hl_iog0 = Orga[  man].hl_iog0;
         mix[0]='-'; --looks_like; }

      if(random()%2) { Baby.hl_iog2 = Orga[woman].hl_iog2;
         mix[1]='+'; ++looks_like; }
      else {           Baby.hl_iog2 = Orga[  man].hl_iog2;
         mix[1]='-'; --looks_like; }

      if(random()%2) { Baby.fixdec00 = Orga[woman].fixdec00;
         mix[2]='+'; ++looks_like; }
      else {           Baby.fixdec00 = Orga[  man].fixdec00;
         mix[2]='-'; --looks_like; }
      if(random()%2) { Baby.fixdec0rest = Orga[woman].fixdec0rest;
         mix[3]='+'; ++looks_like; }
      else {           Baby.fixdec0rest = Orga[  man].fixdec0rest;
         mix[3]='-'; --looks_like; }

      if(random()%2) { Baby.fixdec2all = Orga[woman].fixdec2all;
         mix[4]='+'; ++looks_like; }
      else {           Baby.fixdec2all = Orga[  man].fixdec2all;
         mix[4]='-'; --looks_like; }

      if(random()%2) { Baby.nt_po0 = Orga[woman].nt_po0;
         mix[5]='+'; ++looks_like; }
      else {           Baby.nt_po0 = Orga[  man].nt_po0;
         mix[5]='-'; --looks_like; }

      if(random()%2) { Baby.nt_po2 = Orga[woman].nt_po2;
         mix[6]='+'; ++looks_like; }
      else {           Baby.nt_po2 = Orga[  man].nt_po2;
         mix[6]='-'; --looks_like; }

      if(random()%2) { Baby.refper0 = Orga[woman].refper0;
         mix[7]='+'; ++looks_like; }
      else {           Baby.refper0 = Orga[  man].refper0;
         mix[7]='-'; --looks_like; }

      if(random()%2) { Baby.refper2 = Orga[woman].refper2;
         mix[8]='+'; ++looks_like; }
      else {           Baby.refper2 = Orga[  man].refper2;
         mix[8]='-'; --looks_like; }

      if(random()%2) { Baby.refpercell = Orga[woman].refpercell;
         mix[9]='+'; ++looks_like; }
      else {           Baby.refpercell = Orga[  man].refpercell;
         mix[9]='-'; --looks_like; }

      if(random()%2) { Baby.delay2 = Orga[woman].delay2;
         mix[10]='+'; ++looks_like; }
      else { Baby.delay2 = Orga[  man].delay2;
         mix[10]='-'; --looks_like; }
      }
   mix[11]='\0';

   Baby.fitness = 1.0; //There's no future, no future for you.

   baptice(&Baby);

   return(mix);

}//sex


void roulette_selection(long count) {
long acfi[POPU], //accumulated fitness array
     accufit = 0,
     ball = 0,
     mutations = 0;
int mother = 0,
    father = 0,
    c;
   if(count >= 1234567891) {
      printf("\nstuck after 1234567891 babys, press any key");
      getch();
      }
   for(c=0; c<POPU; c++) {
      accufit += (long) (Orga[c].fitness*100.0);
      acfi[c] = accufit;
      }
   //choose father and mother with different genomes
   while(strcmp(Orga[father].name,Orga[mother].name) == 0) {
      ball = random() % accufit;
      for(c=0; c<POPU; c++)
         if(ball <= acfi[c]) {
            mother = c;
            break;
            }
      ball = random() % accufit;
      for(c=0; c<POPU; c++)
         if(ball <= acfi[c]) {
            father = c;
            break;
            }
      }
/*
   fprintf(gene_output,"\nmami");
   dump_orga(Orga[mother],mother);
   fprintf(gene_output,"\npapi");
   dump_orga(Orga[father],father);
   fprintf(gene_output,
      "\n                                        %s mix",sex(mother,father));
*/
   sex(mother,father);
   while(clone_test(&Baby)) {
      mutate(&Baby);
      baptice(&Baby);
      mutations+=2;
      }

   reset(1);
   set_params(&Baby);
   Baby.fitness = 1.0 + run1screen();

   dump_orga(Baby,-1);
   fprintf(gene_output," mut%2d",mutations);
   if(!SEE) printf(" mut%2d",mutations);

   c=POPU-1;
   while(c>0) { //sort baby in population
      if(Baby.fitness < Orga[c-1].fitness) break;
      Orga[c]=Orga[c-1];
      --c;
      }
   if(Baby.fitness >= Orga[c].fitness) {
      Orga[c]=Baby;
      fprintf(gene_output," :)");
      if(!SEE) printf(" :)");
      }
   else {
      fprintf(gene_output," X(");
      if(!SEE) printf(" X(");
      }

}//roulette_selection


void bubble_sort(void) { //?? what a shame
struct organism orgacopy[POPU];
int order[POPU];
int swaped = 1,temp,c;
   for(c=0; c<POPU; c++) order[c] = c;
   while(swaped) {
      swaped = 0;
      for(c=0; c<POPU-1; c++)
         if(Orga[order[c]].fitness < Orga[order[c+1]].fitness) {
           temp = order[c];
           order[c] = order[c+1];
           order[c+1] = temp;
           swaped = 1;
           }
      }
   for(c=0; c<POPU; c++)
      orgacopy[c] = Orga[order[c]];
   for(c=0; c<POPU; c++)
      Orga[c] = orgacopy[c];
}//bubble_sort


void accu_synch(int mode) { //accumulated synchronization measure
int c,c2,c3;               //mode:0 clean mode:1accumulate
   if(mode == 0) {
      for(c=0; c<R_P_SPAN; c++)
         for(c2=0; c2<R_P_SPAN; c2++) {
            His2D[c][c2] = 0;
            His2Ds[c][c2] = 0;
            }
      Accumulated = 0;
      }
   else {
      for(c=0; c<N_C; c++) { //His2D is read by show_synchro_table in hires.c
         His2D[State[c]][WaveL[c]]++;
         if(!State[c])
            His2Ds[WaveL[c]][WaveL1[c]]++;
         }
      Accumulated++;
      }
}//accu_synch


double rate_accu_synch(void) {
double rate = 0.0;
int c,c2,c3;

   for(c=0; c<R_P_SPAN-1; c++)
      for(c2=0; c2<R_P_SPAN-1; c2++) {
         if(His2Ds[c][c2] > RATESYNCHCAP)
            rate += RATESYNCHCAP * RATE_SYNCH[c][c2]; //***
         else
            rate += His2Ds[c][c2] * RATE_SYNCH[c][c2]; //***
         }
   //roulette needs positive numbers /0error
   if(rate <= 0) rate = 0.01;
   return(rate / Accumulated);
}//rate_accu_synch


double run1screen(void) {
double run = 0.0;
int c;
   reset(1);
   hl_factors();                 //transform half-lifes to factors
   if(EQUA) setEqu();
   if(SEE) plot_World();                 //show frog's world
   if(SEE) show_LMRG();

   accu_synch(0);
   for(c=0; c<SHOW_GEN; c++) { //first screen
      retina();
      compass();
      compute();
      if(check_cells() != -1) return(0.0);
      accu_synch(1);
      Gen++;
//      motor_navigation(' ');  //unused return
      ++TimePos;
      if(SEE) plot_depo();
      }
   run = rate_accu_synch();

   if(SEE) {
      if(DispUp == 'B') bars();
      if(DispUp == 'H') {histogram(1); synchro_stats(); }
      }

   return(run);
}//run1screen


void batch(long loops) { //if SEE==1:show depo,etc. 0:run blind, print text
long c;
char *babycount = "01234567890123456789";
int c2;
   reset(1);
   if( (gene_output = fopen("geneout.txt","w+")) == NULL) {
         clean_line(1,1);
         printf("can't save file geneout.txt");
         delay(2000);
         exit(1);
         }
   if(SEE) if(start_graphics()) exit(1);

   init_orga();

   seed_popu();

   for(c=0; c<POPU; c++) {
      reset(1);
      set_params(&Orga[c]);
      Orga[c].fitness = 1.0 + run1screen();
      if(!SEE) printf("%6D \n",c);
      }

   bubble_sort();

   dump_popu();

   srand48(1234567891L);//limits.h: #define LONG_MAX 2147483647L
   for(c=0; c<loops; c++) {
      roulette_selection(c);
      fprintf(gene_output," babys:%6D",c);
      if(!SEE) printf(" babys:%6D",c);
      if(SEE) {
         sprintf(babycount," babys:%6D",c);
         GrTextXY(500,214,babycount,WHITE,DARKGRAY);
         }
//      if(!SEE) if(c % 100 == 0) dump_popu();
      if(bioskey(1)) {
         getch();
         break;
         }
      }
   fprintf(gene_output,"\n\n after %D babys\n\n",c);
   dump_popu();

   fclose(gene_output);
   reset(1);
   hl_factors();
   if(EQUA) setEqu();
   set_params(&Orga[0]);
   save_params(0,"param00.txt");
   fclose(gene_output);
   exit(0);
}//batch
```