Modifying PLATO for neural network simulation

van Spaandonk, J.

Award date:
1990

Eindhoven University
of Technology
Department of Electrical Engeneering
Design Automation Section (ES)

Modifying PLATO for Neural Network Simulation

John van Spaandonk

Eindhoven, August 28, 1990

Coached by: ir. W.J.M. Philipsen and ir. H.W. Buurman

Abstract

Neural nets can be simulated using the new circuit simulator plato. Plato contains basically three improvements over other existing circuit simulators. These are a piecewise linear component modeling technique, the use of latency of sub circuits and the distributed storage of the circuit information. Plato furthermore uses some optimizations. For example, for most networks the system matrix is sparse, so plato offers sparse techniques that take advantage of this fact and thus reduce simulation time.

Neural nets can be used to solve optimization problems. Those networks consist of neurons and connections, and are based on the structures found in biological brains. Nets for optimization problems can be found by transforming individual problem constraints into separate contributions to the energy function of the neural net. This energy function describes the total energy of the circuit and determines its final state, which represents a solution to the problem. From the energy function the system matrix can easily be found.

Plato was modified to make it especially suited for neural net simulation. Because neural nets have very many interconnections, the circuit matrix of such a net is not sparse, causing the sparse techniques to impose an extra, undesired overhead. Therefore these sparse techniques were removed. Furthermore the matrix describing the neuron has some special properties. Routines that use this matrix could be considerably simplified, taking advantage of this special matrix structure. In addition to these modifications, plato was adapted to use the special vector/concurrency capabilities of the available alliant fx/8 computer.

To test the performance gain obtained by the various stages of the optimization process, the neural net for the tsp problem was used. A special program was written to produce the plato input from a functional specification of the tsp problem. Various instances of this problem were used to test the performance for different problem sizes.

The changes made to plato resulted in a performance gain of 70%. This gain cannot easily be further enhanced, because 77% of the total simulation time is spent in routines that already are optimized as far as possible.

The simulation time greatly depends upon the method of integration used by plato and upon the accuracy of integration. Best results were obtained with the trapezium integration method. An instable method is forward Euler, resulting in small time steps and hence long simulation times. The best value for the integration accuracy is 10^{-2}.

It is suggested that, if plato is going to be used for neural net simulation, a program is written that converts a generic description of a neural net directly into a data structure that can be used by plato. This saves much overhead.

Contents

LIST OF FIGURES

LIST OF TABLES

1. Introduction

Nowadays, electrical circuit simulation is performed mostly using software tools like SPICE that are based on the Newton-Raphson iteration method. Tools using this iteration scheme are widely in use, as well in industry as in academic environments. But they have a number of drawbacks. Firstly, they are computationally very expensive. In each iteration the complete set of circuit equations is solved. Secondly, as has been known for a long time, the Newton Raphson scheme is hampered by convergence problems. Thirdly the user cannot change existing component models or introduce new models.

To overcome these problems, the circuit simulator plato [7] has been developed. Its name is an acronym for *P*iecewise *L*inear *A*nalysis *TO*ol. It has for its components models piecewise linear models, and the user may update the component library. Furthermore latency of sub circuits is exploited, a different timestep is assigned to different parts of the circuit. Because with the piecewise linear modeling technique both analog and digital components can be defined, plato is also capable of mixed level simulation. Plato includes several optimizations, for example because nearly all practical networks have a low degree of interconnectivity (which causes the network matrix to be sparse) the simulator internally works with sparse data structures.

With its new approaches, plato offers for most networks a very nice alternative to the simulators based on the classic Newton-Raphson scheme. But when networks with a high degree of interconnectivity are simulated, some of the various techniques and tricks that are used in plato become abundant. An example of a network type with many connections is the *neural net.* An additional property of neural nets is that they are built from identical components, called *neurons.* There even are neural nets in which every neuron is connected to every other neuron, this type is called *completely connected.* An example of a completely connected neural net is the one that is used to solve the well known traveling salesman problem.

Because currently research is being done into the applicability of neural nets in the field of automatic system design, there is a need for a neural network simulation tool. It was decided to modify the existing version of the circuit simulator plato to make it especially suitable for the simulation of neural nets. This means taking advantage of the special properties of neural nets to speed up plato. To accomplish this goal, several modifications were made.

In chapter 2 a brief introduction is given on neural nets. Also discussed there is a neural net for the tsp problem. In chapter 3 are discussed those parts of plato that are important to understand how simplifications can be made if only neural nets are simulated. It is not attempted to gain a complete insight into the workings of plato, merely is tried to get a picture of the parts of plato that do not perform very well for neural net simulation. In chapter 4 are discussed the actual changes that were made to plato in order to speed up neural network simulation. In chapter 5 are evaluated the tests that were run to measure the obtained performance gain for the various modifications. Finally in chapter 6 are abstracted the conclusions and are made final recommendations.

2. Neural nets

Some properties and the general topology of neural nets are discussed. It is noted that this chapter merely serves as a brief, informal introduction on the subject. For a more elaborate treatment refer to the appropriate literature.

2.1 An introduction into neural nets

The last few years, much research has been done on the applicability of neural nets and on suitable methods for mapping optimization problems on such a net. A great deal of literature has been written on the subject, for instance [2] which gives a complete treatment or [6] which offers a more global introduction.

Generally speaking, neural nets offer the possibility to solve problems using a network based on the structures found in biological brains. In neural nets we find the abstract counterparts of physical structures such as neurons and synapses. It turns out that neural nets are suited for solving optimization problems. Some examples of such problems are: Given a circuit board with components, what is the most effective way to wire the board? Given a map, color it using as little colors as possible. Given a set of locations on a flat twodimensional surface, find the minimal distance that has to be traveled to visit every location. This last problem is called the Traveling Salesman Problem (tsp). The answer to an optimization problem is a solution which satisfies a certain constraint. Another class of problems is formed by *decision problems*, the answer to which is either yes or no. Because every optimization problem can be stated as a decision problem and vice versa, neural nets are suited for the solution of both optimization and decision problems. For instance: if the tsp problem is stated as an optimization problem, the solution is the minimum distance that has to be traveled to visit each location. The constraint that has to be satisfied is that the total traveled distance must be minimized. If the tsp problem is formulated as a decision problem, we ask whether we can visit every location while traveling a distance equal to or below a certain limit. Obviously the answer to this question is either yes or no. For more information refer to. [3]

The basic building block of a neural net is the Hopfield neuron, depicted in Fig. 2.1. This neuron basically is a non-linear analog summing device. The first stage consists of a summer with inhibitory inputs (circles) and excitatory inputs. The excitatory inputs are directly passed to the summing device, while the inhibitory inputs are first inverted. After addition, the inputs are amplified by a constant gain G. With this gain factor the input sensitivity of the neuron is adjusted. If the gain is very high the neuron is very sensitive and behaves like a comparator. The slightest positive input will then drive the output towards 1. If the gain is too low then the neuron does not easily choose a stable output value. Normally, when connected in a network, neurons all have an identical offset input. If the gain is very low, the neuron inputs are dominated by this offset value so the neuron outputs have approximately the same value. Finally the nonlinear element compresses the summed and amplified inputs, causing the neuron output value to be in the range [0 , 1]. Note that the transition function of the nonlinear device is a sigmoid. The

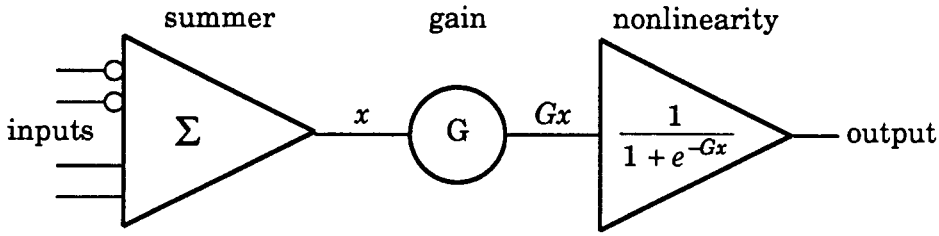

Figure 2.1. *Circuit model for a Hopfield neuron*

form of this sigmoid depends on G, for higher values of G the curve is sharper.

Typically, a neural net is composed of many neurons. Each neuron output is connected to one or more excitatory or inhibitory inputs of many other neurons. For example regard the neural net of Fig. 2.2.

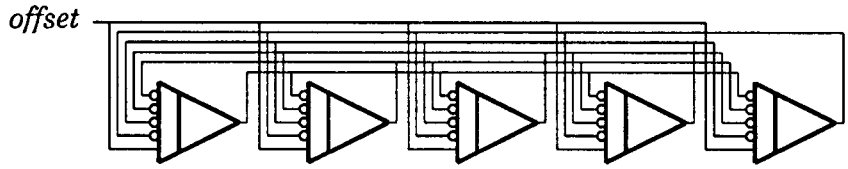

Figure 2.2. *An example of a neural net: the 5-flop*

It is composed of five neurons like that of Fig. 2.1, which are schematically represented. In this particular net, each neuron has its output coupled to the inhibitory inputs of the other four neurons. If the output of one neuron rises then the other neurons are suppressed by the inhibitory connections; each neuron strives to suppress the four others. The outputs can take any value in the range [0 , 1]. To start a calculation, we have to bring the net into an unstable state. This is a state in which the internal energy of the network is high. This is accomplished by giving all 5 outputs an initial value somewhere between 0 and 1. The state of the network now resembles a marble lying precisely on the top of a smooth hill. When we let the net go, starting from this situation its symmetric state will eventually be disturbed by random noise fluctuations, which are provided by rounding errors if the net is simulated on a digital computer. With its symmetry broken the net will rapidly converge to a stable state, like our marble rolling down the hill. Because of the inhibitory connections this is a state in which only one of the neurons is active. These connections can be seen as defining the landscape around the hilltop. According to our network there are five equally spaced valleys radiating from our hill. The hypothetical marble could choose any of these valleys with equal probability to reach a lower state of energy.

Because the discussed network has only five stable states, it is called a *5-flop*, a term derived from *flip-flop* which denotes a device that has only two stable states.

The general case of such a net is called an *n*-flop, in a stable state only 1 neuron out of *n* can be active. With such an n-flop we can represent an integer that can take n values. In fact the n-flop maps the integer programming problem to the binary programming problem, each bit is represented by an n-flop neuron.

Some problems arise with the actual implementation of a neural net. For instance, it is not exactly easy to let precisely one neuron win. With the network of Fig. 2.2, in the end more then 1 neuron could be active simultaneously. Also no neuron could be active at all or, if the neuron gain is too high, the net could end up in an oscillating state. An actual good implementation depends heavily on the correct choice of constant factors such as the neuron gain or the offset. Considerations like these are discussed in some more depth in chapter 5. The 5-flop merely serves as an example to gain insight in the ideas that lay behind such nets. More about the actual implementation of a neural net is said in section 2.3.

A few observations can be made about the calculation process of our network. It is seen that, in contrast to sequential computers, neural nets calculate a solution massively parallel. Each neuron contributes to the calculation of the final state during the time that elapses between the initialization of the network and the moment a final state is reached. Thus the complete network participates in the solving process. When the tsp problem would be solved on a sequential computer this amounts in an enormous amount of steps, with little being done in each step. In contrast, as seen in -for instance- the human brain, the neural net takes only a few neural time constants to reach a stable state, while a lot happens in one such time constant.

Another feature of neural nets is that the reached solution does not necessarily has to be optimal. In fact, the solution found by a neural net is generally just one of many very good solutions. Very good here means that the solution is only marginally worse than the best solution. For very many applications, like vision and robotics, this "good" solution suffices, because in these applications it is often more convenient to have a very good solution very fast than a marginally better exact solution that takes much more time to compute.

The natural question that arises at this stage is: how does one find a network that solves a certain problem? Note that the network has to represent a solution to the problem it solves. Therefore this solution must be cast into a form that can be represented by a neural net, after which a net must be found that by its topology represents the form of this encoded solution. It will be shown that, if a good solution encoding is chosen, this is accomplished rather easily. Another constraint that must be met by the net is that it should prefer good solutions. In sections 2.2 and 2.3 it is demonstrated that we can meet demands like this by adding specific inhibitory interconnections to our net. In section 2.2 an intuitive approach is presented for constructing a net for the tsp problem, while in section 2.3 is discussed a more correct, rigid approach.

2.2 The traveling salesman problem

Informally, the tsp problem is described as follows. Suppose that a salesman has to visit *n* cities to perform his trade. It is assumed that this hypothetical salesman is

transported by airplane, so that he is able to travel in straight lines. A route that takes him along each city and returns him to his starting point is called a *tour*. According to the time = money principle, the salesman tries to choose the best possible route, that is: the route which enables him to visit every city once while minimizing the total traveled distance. In other words: he will search for the shortest tour. The calculation of this shortest tour is called the tsp problem. To find the shortest tour is very hard, in fact this problem belongs to the class of NP-complete problems. The time needed to solve it increases exponentially with problem size (the number of cities). This is easily understood when the number of possible routes is evaluated. If there are n cities to be visited, there are $n!$ possible tours. Because each such tour has n different starting points and can be traveled in 2 directions, the total number of distinct, closed paths is $n! / 2n$, which grows exponentially with the number of cities n. Because all these possibilities will have to be evaluated, and because one tour can be evaluated in time $O(n)$, the time needed to find the optimal tour increases exponentially with problem size.

It turns out that a tsp problem solution is represented by a neural net in an elegant way. To find the net, the first step is to code the solution of the tsp problem into a form that resembles the topology of a neural net. Suppose that we have a tsp problem with $n = 5$ cities, labeled $A \cdots E$. The solution of the problem can be represented by a string of these letters. A possible solution (not necessarily optimal) is given by the string C–A–E–B–D. Such a string forms one of many possible encodings of that solution to the tsp problem. An encoding which more suits our purposes is one that associates with each city a string of 5 binary digits. Because in our solution, city C is the first visited city, according to this encoding scheme the string associated with C is

1 0 0 0 0

The complete solution is represented by the matrix that has for its rows the strings associated with city $A \cdots E$. A complete representation of our solution is thus given by the solution matrix

	1	2	3	4	5
A	0	1	0	0	0
B	0	0	0	1	0
C	1	0	0	0	0
D	0	0	0	0	1
E	0	0	1	0	0

meaning that first city C is visited, then city A etc. Note that a matrix of this form is called a *permutation matrix*, it contains exactly one 1 entry in each row or column.

It will be shown that this encoding scheme can be easily mapped on a neural net. Note that each row of the solution matrix has only one non-zero element. But this is just the property of the 5-flop of Fig. 2.2. Therefore, a matrix row can be represented by one 5-flop. Now suppose that we have a neural net, constructed of five stacked 5-flops, thus consisting of $5 * 5 = 25$ neurons. Because each row is a 5-flop, in the end only one neuron will be active in each row. Now add additional inhibitory connections between the neurons in each column, according to the 5-flop scheme, so each neuron of a row 5-flop will have inhibitory connections to the other

four neurons in its column. Then also in each column, only one neuron will be active in the stable state of our network. Therefore the network constructed in this manner represents a permutation matrix: it represents a syntactically correct solution to the 5 city tsp problem.

The last step towards the construction of the final network consists of the addition of inhibitory connections that will enforce the solution to be one of a small percentage of near-optimal solutions. To see how that can be accomplished, again examine the solution matrix. One row of this matrix represents one city, and has an entry for each position that this city can have in a tour. Of course only one of these entries is 1, the city is visited exactly once. Examine one of the n^2 matrix entries, and the neuron that represents this entry. Suppose our entry is in row *B* at position 3, representing tour position 3 of city *B*. Give this entry, and its corresponding neuron, label *B3*. The columns to the left and right of entry *B3* contain exactly one 1 entry. If entry *B3* would be 1 then these two 1 entries determine which cities are visited just before and just after city *B*. Therefore these columns can be regarded as the "coming from" and "going to" columns (in this scheme, the left- and rightmost columns of the matrix are considered to be adjoining). Assume that the intercity distances are normalized, so that the largest distance is 1 (this imposes no loss of generality). Now add inhibitory connections to neuron *B3*, coming from the neurons in its "coming from" and "going to" columns (2 and 4) that are not in row *B*. If the distance between city *B* and *C* is 0.7, then the inhibitory connections from neurons *C2* and *C4* to neuron *B3* have strength 0.7. It will be clear that if neurons in different rows and in two adjoining columns represent cities that are close to each other, these neurons have small mutual inhibitory connections and therefore a bigger chance of simultaneously being active than when the inter city distance is large. In other words: solutions that minimize the distance between these two cities have a higher probability of occurring than other solutions. Now give all neurons such additional inhibitory connections from their adjoining columns. Then solutions that minimize the total tour length have a higher probability of occurring. Therefore, by adding these connections, we have constructed a neural net that not only represents a syntactically correct solution to the tsp problem, but also generates solutions that minimize the total tour length.

2.3 The tsp energy function

As remarked in the previous section, to obtain a solution with a neural net it is brought into a state of high energy. After it is turned loose it settles down into a state of lower energy. Exactly which state this is (stated differently: exactly what neurons are active in that state) depends on the topology of the network. It turned out that solution constraints can be formulated, and that these constraints intuitively lead to a specific network topology. But there exists a more general method, that starts by translating the constraints into different terms that together constitute the network *energy function*. This method, and the idea of an energy function as deduced by Hopfield are treated here.

Firstly the model of our neurons must be adjusted, because in the neuron of Fig. 2.1 no energy can be stored. To make this possible, we have to make our neurons more "physical", we have to introduce a delay that represents the propagation time of an

input change to the output. Here the analog model for the neuron is used that originally was designed by Hopfield and Tank [2]. The neural delay is represented by an RC network at the input of each neuron. This RC network models the cell membrane impedance of a biological neuron. Furthermore, unlike biological neurons, the neurons do not have inhibitory and excitatory inputs but instead each neuron has two identical amplifiers, one of which is inverting. The total energy that is stored in the neural net depends on the neuron parameters and the interconnections. The function that describes the total energy is called the *energy function*. Hopfield and Tank deduced the tsp energy function based on a circuit composed of analog summation devices.

A general model of such an analog circuit is depicted in Fig. 2.3.

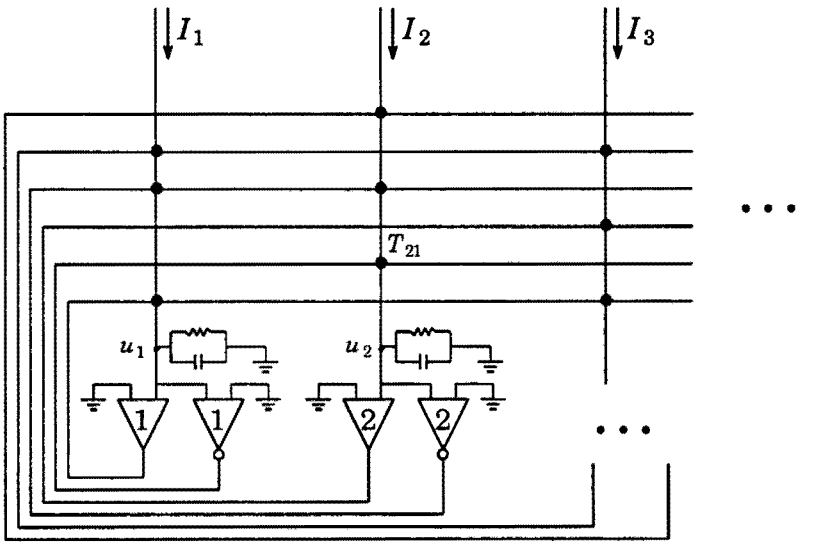

Figure 2.3. *The analog neural net*

The output of a neuron can be connected to any other neuron through a connection network. A dot at an intersection in this network represents a resistor that connects the intersecting lines. With this resistors we can influence the strength of an inhibitory or excitatory connection. The network energy is supplied by the input currents $I_1, I_2, ..., I_n$. Synapses are represented with resistors of value $R_{ij} = 1/T_{ij}$. If the synapse is excitatory ($T_{ij} > 0$), the resistor is connected to the normal output of neuron j. If the synapse is inhibitory ($T_{ij} < 0$), the resistor is connected to the inverted output of neuron j. So with matrix T the connectivity of the neurons is described. Therefore T is called the *connection matrix*. Note that for the 5-city tsp problem the dimension of T is $N = 25*25 = 625$, which is the square of the number of neurons in the circuit.

Let the non-inverting and inverting outputs of neuron i be V_i and $-V_i$. The Kirchoff current equation for the input of neuron i is given by

$$\begin{cases} C_i(du_i/dt) = \sum_{j=1}^{N} T_{ij}\, V_j - u_i/R_i + I_i \\ V_j = g(u_j) \end{cases} \tag{2.1}$$

Here R_i is a parallel combination of the input resistor ρ_i of neuron i and the resistor values of the connection network:

$$1/R_i = 1/\rho_i + \sum_{j=1}^{N} 1/R_{ij} \tag{2.2}$$

Furthermore $g(u_j)$ is the transition function of the nonlinear element of Fig. 2.1. Let all neurons have identical RC networks and let all values R_{ij} of the resistor network be equal, so that $R_i = R$ and $C_i = C$. Furthermore divide (2.1) by C and redefine T_{ij}/C and I_i/C as T_{ij} and I_i. Then (2.1) transforms to

$$\begin{cases} du_i/dt = -u_i/\tau + \sum_{j=1}^{N} T_{ij}V_j + I_i \\ \tau = RC \\ V_j = g(u_j) \end{cases} \tag{2.3}$$

If the input voltages of the neurons at $t = 0$ are given, this equation describes the evolution of the state of the network with time. Integration of this equation allows any hypothetical network to be simulated on a computer. The equation that describes the energy of the complete network is derived from (2.3) and is given by

$$E = -1/2 \sum_{i=1}^{N} \sum_{j=1}^{N} T_{ij}V_iV_j - \sum_{i=1}^{N} V_iI_i \tag{2.4}$$

If the curves of the non linearities are sharp, then the stable energy states of a neural net are the local minima of (2.4). In the terminology of Hopfield, the state space over which the circuit operates is the interior of the N-dimensional hypercube defined by $V_i = 0$ or 1. Therefore, in the high gain limit, the stable states of the network are the corners of this hypercube. It follows that by choosing interconnectivities T_{ij} and input bias currents I_i, a neural net can be constructed that describes the solution to an optimization problem. Then the network is brought in an unstable state by choosing initial input voltages u_i and converges to a stable state that represents a solution.

All necessary requirements for a correct and good solution to the tsp problem can be met by adding for each requirement an additional term in the energy function. The first requirement is that the energy function must favor stable states that have the form of a permutation matrix. According to Hopfield, this requirement is met by the energy function

$$E = A/2 \sum_X \sum_i \sum_{j \neq i} V_{Xi} V_{Xj} + B/2 \sum_i \sum_X \sum_{Y \neq X} V_{Xi} V_{Yi} + C/2 \left(\sum_X \sum_i V_{Xi} - n \right)^2 \qquad (2.5)$$

The n^2 neurons are labeled according to a scheme in which V_{Xi} means the i-th neuron of row (city) X. It is easily seen that the first triple sum is zero if, in each row, no more than one neuron output is 1. The second triple sum is zero if in each column no more than 1 neuron is one. To make sure that *exactly* n neurons have output value 1 in an n city tsp problem, the third term is present. Thus (2.5) has value 0 only for those solutions that have the form of a permutation matrix. To ensure that short tours are favored, an additional term is added:

$$D/2 \sum_X \sum_{Y \neq X} \sum_i d_{XY} V_{Xi} \left(V_{Y,i+1} + V_{Y,i-1} \right) \qquad (2.6)$$

Here d_{XY} is the distance of cities X and Y. Note that for a valid tour (2.6) represents the numerical length of the tour. This term amounts to connections that enforce a good solution, as discussed in section 2.2. The total energy function of the tsp problem is the summation of eqs. (2.5) and (2.6).

The connection matrix is easily deduced from the energy function. The elements T_{ij} are found when the energy equation for the tsp network is compared with the general form of the energy function (2.4). The elements are:

$$\begin{aligned} T_{Xi,Yj} = & -A \delta_{XY}(1 - \delta_{ij}) \\ & -B \delta_{ij}(1 - \delta_{XY}) \\ & -C \\ & -D d_{XY}(\delta_{j,i+1} + \delta_{j,i-1}) \end{aligned} \qquad (2.7)$$

Here $\delta_{ij} = 1 \Leftrightarrow i = j$. The four terms represent respectively the inhibitory connections within each row, the inhibitory connections within each column, the global inhibition and the data term. The external input currents are:

$$I_{Xi} = +Cn \qquad (2.8)$$

2.4 Conclusions

It was shown that, generally, a neural net is a circuit composed of interconnected neurons. Some properties of our neural nets are that all components are identical neurons and that these neurons have very many mutually inhibitory connections. A neuron is the circuit equivalent of its biological counterpart, The treated neurons have electrical equivalents for features of biological neurons like cell membrane impedance, inverting and non-inverting inputs, gain and non-linearity.

A neural net that represents a solution to an optimization problem may be found in two steps, namely by firstly coding the solution in a form that can be represented by the network topology and secondly adding inhibitory connections that ensure that the found solution is near-optimal. To illustrate how a neural net for an optimization problem can be found, the neural net for the tsp problem was constructed. It was shown that the tsp problem can be formulated in terms of

several constraints, a valid and good solution to the problem has to meet these individual constraints.

A formal approach to the construction of a neural net for a specific problem is to translate the various constraints into individual terms that together constitute the network *energy function*. By comparing this energy function with the general form of the energy function as given by eq. (2.4), the *connection matrix* is deduced. This connection matrix describes the connections between the network neurons and therefore defines the network topology. The energy function and connection matrix of the tsp problem were deduced.

3. Plato

Plato is a very complicated piece of software. To explain the complete user interface, program structure and simulation process would require a lot of space and is out of the scope of this report. In this place only those parts are discussed that are important to understand how the changes that are discussed in chapter 4 result in a more efficient simulation of neural nets. No special attention is paid to the simulation process itself.

3.1 Sparse implementation

In general, a network that is simulated with plato is composed of components and interconnections. In most simulators, the information about the circuit elements and interconnections is contained in one large system matrix. However in plato this information is stored in a distributed form. Plato knows leafcells that hold all information about the individual circuit components, which in our case are all neurons. The description of each neuron is stored in the leafcell matrix of its corresponding leafcell. Information about the interconnections is contained in a separate system matrix. The leafcell matrix and linear mapping are discussed in section 3.2. Here is discussed the large system matrix.

Evidently for most networks the number of interconnections will be moderate, and therefore the system matrix will be sparse. Plato was written to take into account this fact. The procedures that involve the system matrix were written to create little fill-in, so that the system matrix remains sparse throughout the entire simulation process. To take full advantage of the sparsity of this matrix and the source vectors, they are implemented as sparse data structures.

The nonzero elements of the matrix are stored in a bi-threaded list, as seen in Fig. 3.1. Because there are performed updates on this system matrix S with regular intervals, it is decomposed into its LU factorization, according to:

$$S = L\ U$$

The updates are then composed on the factorization of the system matrix, which turns out to be cheaper.

The system matrix S and the triangular matrices L and U are stored in a sparse data structure depicted in Fig. 3.1. This structure is a bi-threaded list of matrix elements. Every element contains 6 fields:

1. *row_id*: row index of this element
2. *col_id*: column index of this element
3. *val*: value of this system matrix element
4. *lu_val*: value of the LU decomposition of this system matrix element
5. *next_row*: pointer to the next row

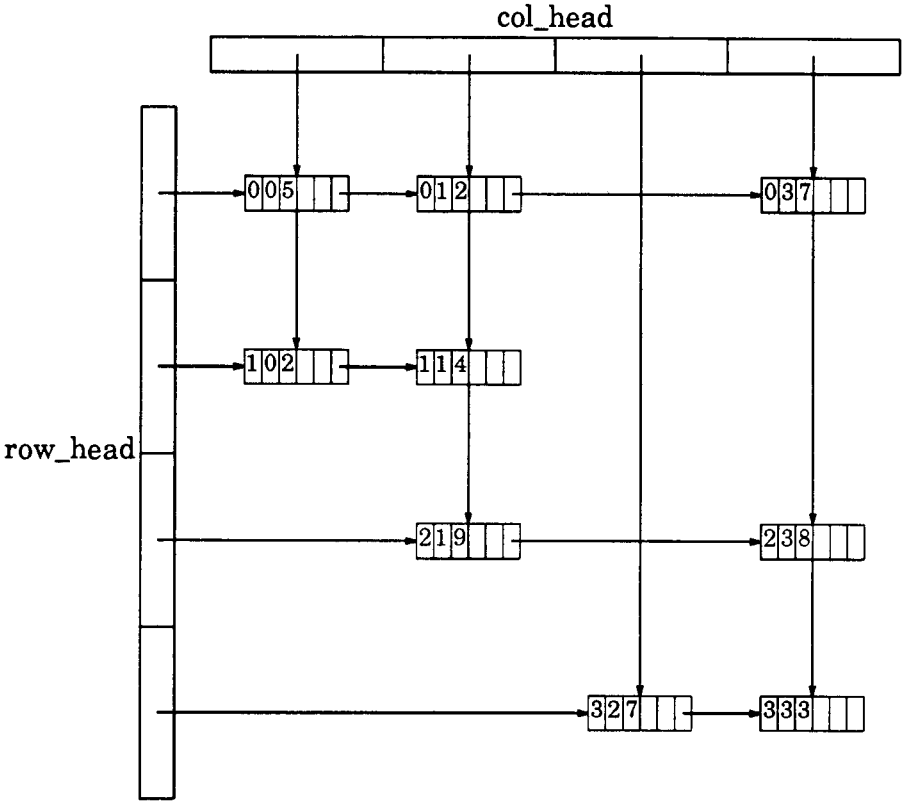

Figure 3.1. *Sparse matrix data structure*

6. *next_col*: pointer to the next column

No lu-decomposition has yet been performed on this matrix, so field *lu_val* of each element is empty. The system matrix has *tot_eqs* rows and *tot_vars* columns. For any physical circuit, the system matrix will be square, so *tot_eqs* = *tot_vars*. With *row_index* = *r* and *col_index* = *c*:

$$lu\_val = \begin{cases} L_{rc} \; ; \; r > c \\ U_{rc} \; ; \; r \leq c \end{cases}$$

The rows and columns can be accessed through pointer arrays *row_head* and *col_head*. The diagonal elements can be accessed directly through a third pointer array *diag* that is not depicted in Fig. 3.1. This array is important because the update algorithm has iteration loops that start at the pivot elements. If *diag* would not have been implemented, the diagonal elements would have to be found by traversing the linked lists, which of course is very time consuming.

Generally the elements of the system matrix or source vectors are called sparse elements. The sparse vectors are stored as linked lists, like one system matrix row or column. Each vector element is composed of 3 fields:

1. *index*: index of the element
2. *val*: value of the element
3. *next*: pointer to the next element

To facilitate operations on the sparse matrix and vectors, several procedures are used. Amongst others, there are procedures for the following operations:

- create sparse elements
- add sparse elements to an existing vector or matrix
- delete sparse elements from an existing vector or matrix
- merge sparse vectors

Note that because generally the connection matrix of a neural net is not sparse, for these networks the sparse techniques impose an extra overhead. It is expected that routines which perform much operations on the sparse structures will be considerably less expensive if the sparse techniques are removed. Precisely how this is done is discussed in section 4.2.

3.2 Pwl matrix of the hopfield neuron

The piecewise linear modeling technique that is used for the dynamical description of a Hopfield neuron is based on a technique developed by van Bokhoven [1]. Van Bokhoven discovered that a large class of pwl models for electrical circuit models can be described by a linear resistive multiport terminated by ideal diodes. For a complete description of this technique refer to [1]. A good introduction to the subject of pwl modeling (with some examples) is given in [5]. A short introduction is given below.

A general description of a continuous piecewise linear dynamical system conceived by van Bokhoven is given by:

$$\begin{bmatrix} 0 \\ \dot{u} \\ p \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} x \\ u \\ q \end{bmatrix} + \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{3.1}$$

$$\text{where } \dot{u} = \frac{\partial u}{\partial t}, \text{ and} \tag{3.2}$$

$$p \,.\, q = 0,\ p \geq 0 \text{ and } q \geq 0. \tag{3.3}$$

The system's terminal variables are represented by vector x. The vector u represents the system's dynamical variables. With this vector, the dynamic behavior of the system is modeled. The vectors p and q are used to describe the piecewise linear model of the circuit component which is represented by the matrix of equation (3.1).

To explain the construction of such a model, the dynamical pwl model of a Hopfield neuron is deduced. This pwl model is based on a more abstract description of the Hopfield neuron, as given in Fig. 3.2.

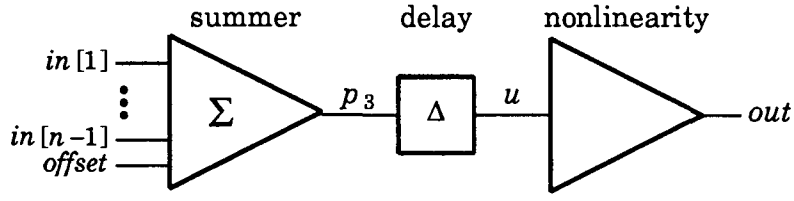

Figure 3.2. *Abstract model for a Hopfield neuron*

Before summation, the inputs are multiplied with individual weighing factors. These weighing factors provide for the resistor values of the connection network of Fig. 2.3. To enable the global sensitivity to be adjusted, each neuron has an additional offset value added to its inputs. The result of the summation is assigned to help variable p_3:

$$p_3 = \sum_{i=0}^{n-1} w[i]\, in[i] + offset \tag{3.4}$$

The delay provides our model with a representation of the RC networks that are present in Hopfield Neurons. The relation between p_3, u and $\dot{u}$ is given by:

$$\dot{u} = \frac{1}{\Delta}\,[\,p_3 - u\,] \tag{3.5}$$

The transition function of the non-linear element is given by

$$\frac{1}{1 + e^{-G(x - 0.5)}}$$

This function is depicted in Fig. 3.3 (dotted line) and is approximated by three linear segments (solid line). This model can be described by three equations:

$$\begin{cases} p_1 = u + q_1 + c_1 \\ p_2 = -u + q_2 + c_2 \\ out = G \times (u + q_1 - q_2 + c_1) \end{cases} \tag{3.6}$$

with

$$c_1 = (\frac{1}{G})\, init\_out$$

$$c_2 = (\frac{1}{G})\,(1 - init\_out)$$

The constant G is the gain factor of Fig. 2.1. The constant *init_out* is the initial output value that is discussed in section 2.1. It is easily seen that (3.6) describes the

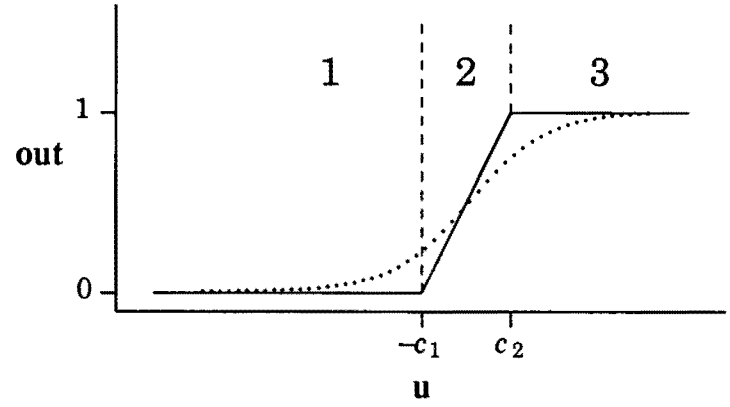

Figure 3.3. *Three segment pwl approximation of the nonlinear element*

relation depicted in Fig. 3.3. The key to understanding the model is given by (3.3), according to which we have 4 different combinations of $p_1; p_2; q_1$ and q_2. According to van Bokhoven's model, these variables represent the current-voltage relation of an ideal diode. The pwl simulator solves for p and q when u is changed. According to Fig. 3.3, u passes through three trajects, labeled 1, 2 and 3. Each traject is represented by another combination of p_1, p_2, q_1 and q_2:

$$u \le -c_1 \Leftrightarrow \left\{ \begin{array}{l} p_1 = 0;\ q_1 \ge 0 \\ p_2 \ge 0;\ q_2 = 0 \end{array} \Leftrightarrow \begin{array}{l} q_1 = -u - c_1 \\ p_2 = -u - c_1 \\ out = G\,(\,u - u - c_1 + c_1\,) \end{array} \right\} \Leftrightarrow out = 0 \qquad (3.7)$$

$$-c_1 \le u \le c_2 \Leftrightarrow \left\{ \begin{array}{l} p_1 \ge 0;\ q_1 = 0 \\ p_2 \ge 0;\ q_2 = 0 \end{array} \Leftrightarrow \begin{array}{l} out = G\,(\,u + c_1\,) \\ p_1 = u + c_1 \\ p_2 = -u + c_2 \end{array} \right\} \Leftrightarrow out = G{*}u + init\_out$$

$$u \ge c_2 \Leftrightarrow \left\{ \begin{array}{l} p_1 \ge 0;\ q_1 = 0 \\ p_2 = 0;\ q_2 \ge 0 \end{array} \Leftrightarrow \begin{array}{l} p_1 = u + c_1 \\ q_2 = u - c_2 \\ out = G\,(\,u - u + c_2 + c_1\,) \end{array} \right\} \Leftrightarrow out = 1$$

$$\left\{ \begin{array}{l} p_1 = 0;\ q_1 \ge 0 \\ p_2 = 0;\ q_2 \ge 0 \end{array} \right. \quad \text{this combination cannot occur}$$

Combining eqs. (3.4), (3.5), (3.6) and (3.7) and eliminating help variable p_3 gives:

$$\begin{bmatrix} 0 \\ \dot{u} \\ p \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} x \\ u \\ q \end{bmatrix} + \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

with:

$$A_{11} = [0 \ \cdots \ 0 \ -1 \ ]; \quad A_{12} = [G]; \quad A_{13} = [G \ -G];$$

$$A_{21} = \left[\frac{1}{\Delta} w[0] \ \cdots \ \frac{1}{\Delta} w[n-1] \ 0 \right]; \quad A_{22} = [-1]; \quad A_{23} = [0 \ 0];$$

$$A_{31} = \begin{bmatrix} 0 \cdots 0 \\ \vdots \quad \vdots \\ 0 \cdots 0 \end{bmatrix}; \quad A_{32} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}; \quad A_{33} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

and

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} init\_out \\ \frac{1}{\Delta} offset \\ \frac{1}{G} init\_out \\ \frac{1}{G} (1 - init\_out) \end{bmatrix}$$

Some observations can be made about the special structure of the leafcell matrix for the Hopfield neuron. The pwl model of the non-linear element is contained in A_{33}. The dimension of A_{33} depends on the number of line segments by which the curve of Fig. 3.3 is approximated. The only way in which the pwl model of the neuron will be changed in the future is this number of segments. Therefore A_{33} is the only matrix that possibly is altered. Stated differently: for the optimization of plato the special structure of the other sub matrices of the pwl matrix can be used. For instance: A_{31} is always 0, so all calculations involving A_{31} can be omitted. Furthermore the number of dynamic variables is always 1. Observations like these lead to a quicker version of plato, as is discussed in section 4.3.

3.3 Ndml description of the tsp problem

Ndml++ is shorthand for Network Description and Modeling Language. The original form of ndml++ is ndml, as designed by G.L.J.M. Janssen [4]. The primary goal of the language is to create a textual user interface for the plato user. Ndml++ enables the user to describe the network components and interconnections, and the terminal values. The network components can be described in *leafcells*, which can be used in a *compound system*. A compound system is composed of one or more interconnected leafcells or compounds. Like this any digital or analog network can

be defined in a modular way. When the ndml description of a certain problem is given, the ndml compiler translates this description to a simpler form of ndml which serves as the plato input.

However the plato input for the tsp problem was not written in ndml++ by hand but generated by a program (*city*) that is created specifically for that purpose. The reason for this automated input generation is that it is necessary to experiment with neural nets of various sizes , furthermore it is necessary to be able to quickly adjust several parameters like those appearing in eq. (2.7). That this is a problem for hand written plato input becomes clear when the size of this input is considered: for a 10-city tsp problem it is about 470 kB.

The generation of the ndml output is rather straightforward. Firstly a functional problem specification is read and the values of the connection matrix are calculated, according to eq. (2.7). The pwl description is present in city's source code. With the T values and the pwl description the ndml description is generated. In appendix 1 is explained how city is used, by means of an example. The city input is given there, with the generated output, for a 3 city tsp problem. The constants A, B, C and D are the constants that appear in the energy function (eqs. (2.5) and (2.6)). The delay Δ, gain G and initial output value *init_out* are discussed in section 3.2. City also generates a random noise that is present on the initial output value of each neuron. The reason for this random noise value is that when it is present, the starting state of the network will not be precisely symmetric. Therefore rounding errors in the digital computer on which plato is executed will not influence the choice of the final state of the network. Thus the simulation will give the same results, even if the network is simulated on different computers. Also plato modifications that change rounding values will not influence the result of the simulation. When another tsp problem must be solved, the locations of the cities and other parameters can easily be put into a small file, that will be converted by city to plato input. The format that this file must have is discussed in appendix 1.

3.4 Conclusions

Plato stores the information about the individual circuit components in *leafcells*, while the interconnections are defined in a large system matrix. Because generally the number of connection will be moderate, this system matrix, together with all source vectors, is contained in sparse data structures. It is expected that the resulting sparse techniques result in a less efficient simulation for neural nets. This is caused by the fact that neural nets have a high degree of connectivity, which in its turn causes the system matrix to be non-sparse. No reason is present for maintaining these techniques, they just impose an extra, redundant overhead.

A more abstract model was constructed for a neuron with which the pwl description of a neuron was deduced. It was shown that, for the neuron model of Fig. 3.2, the leafcell matrix has a special structure. For instance, in this model the number of dynamic variables is always 1. Therefore loops in the plato leafcell routines that involve this variable can be removed. Furthermore, because sub matrix A_{31} contains only 0's, calculations involving this matrix can be completely deleted. It is expected that by examining the leafcell routines and adjusting them to the special structure of the leafcell sub matrices, the simulation of neural nets can be made

more efficient.

Normally, the plato input is a compiled version of ndml. For the tsp problem, a program was written that directly generates this input from a functional description of the tsp problem. This is further illustrated in appendix 1. This is very convenient, because now it is easy to generate various instances of the tsp problem without having to rewrite a large file.

4. Changing plato

All changes are discussed that were made to plato. In section 4.1 is presented the general method that has been followed for changing plato. In the sections following 4.1 are discussed the actual changes that were made. The discussions are quite thorough. Wherever needed, pieces of program code are presented to illustrate certain changes. Note that the changes are presented in chronological order. The performance gain that was obtained by each modification is discussed in chapter 5.

4.1 The changing process

The method that was used to guide the process of changing plato can be abstracted as follows. Firstly the unix tool *gprof* is used for the original version of plato to locate the most expensive (time consuming) procedures. These are monitored by gprof during runtime, so when a suitable test case is used, the routines that are expensive for neural net simulation are easily located. As a representative test case various instances of the tsp problem are used (this is valid for the entire changing process). The tsp problem is very suitable, because the neural net that represents this problem is completely connected. Therefore it is expected that especially with this problem, the routines that use the sparse data structures take much time.

For an example of gprof output, regard Table 4.1.

TABLE 4.1. *Time spent per procedure for the original plato version, run on the alliant computer. Only the 10 most expensive procedures are listed. Problem size is 100 neurons.*

% time	calls	total (ms/call)	procedure name
33.7	5715	201.81	ludec_upd
25.0	611588	1.39	i_calc_pwlder
12.5	632962	0.68	i_calc_udotbar
8.8	10439	28.68	non_sparse_solve
8.5	629507	0.51	i_upd_module_vars
5.3	10620	46.33	nonzero_list
1.0	827000	0.04	output_column_variable
0.9	589825	0.73	i_rec_event
0.7	182	125.86	sparse_solve
0.4	95461	0.14	YYlook

In the various columns of Table 4.1 is listed for each procedure respectively the time spent as a percentage of the total simulation time, the number of invocations made,

the time spent in the procedure per call and the time spent in the procedure and its children per call. Note that, according to this table, in the 10 listed procedures is spent 90.9% of the total simulation time. This justifies the implicit assumption that for modifying plato, only the 10 most expensive procedures have to be regarded, sometimes including their children.

Once they are located, procedures are examined to see if they can be modified, keeping in mind the results of chapter 3. After possible changes are made, another test case is run to examine the results of the modifications. Then is evaluated if the new code needs debugging and additional candidates for change are selected. This process is abstracted as follows:

1. Use gprof to select an expensive procedure.
2. Modify it.
3. Test plato by comparing the simulation output with that produced by the original plato.
4. If the new code produces output that is equivalent to the output produced by the original code then go to step 1, otherwise first debug the new code.

4.2 Sparse technique removal

Here is discussed why the sparse techniques were removed and exactly how this was done.

4.2.1 Motivations for removing the sparse techniques

As indicated in section 3.1, the overhead inherent to the sparse techniques is expected to be large when non-sparse system matrices are involved. Indeed it turns out that for a problem size of 100 neurons (10 cities), some 25% of the total program execution time is spent in *ludec_upd* which performs an update on the lu-decomposition of the system matrix. Note that, according to Table 4.1, the time spent in *ludec_upd* relative to the other procedures is not 25% but 16.6% for this smaller problem size. This decrease in time is logical, since the number of iterations in that routine increases quadratically with the dimension of the system matrix, while this is not true for the three leafcell routines (with prefix *i_*) that appear in the top part of table 4.1.

In section 3.1 it is assumed that for neural networks the system matrix is non-sparse throughout the entire simulation. This turns out not to be true. Plato starts by setting the system matrix equal to the identity matrix. Each time this is necessary, an update is done on the system matrix by *ludec_upd*. Therefore it will certainly not be completely full throughout the entire simulation. It is estimated that,on the average, the system matrix is half full, so the gain obtained by removing the sparse techniques will be less than can be concluded from section 3.1. Notice that the additional advantages of removing the sparse techniques (such as elimination of the routine to insert a new sparse element in a vector) still remain.

But these are not the only motivations for removing the sparse techniques. Because routines involving the sparse data structures were vectorized and/or called concurrently wherever possible, the involved routines must work on fixed arrays

instead of linked lists. This is an additional reason for removing the sparse techniques.

4.2.2 Removal of the sparse techniques

The problem that arises with the removal of the sparse techniques is that the original sparse routines and the routines that already are modified to their non-sparse equivalents have to exchange vectors arguments and even are supposed to work on the same global system matrix. This is a problem because evidently sparse and non-sparse variables (vectors, matrices) cannot be intermingled. A possible solution is to change all sparse routines at once, thereby avoiding the problem of exchanging different data structures. Because plato is very large, definitely making it necessary to test modifications on a per routine basis, the problem was solved by a kind of divide and conquer strategy. Four procedures were written to transform a sparse matrix or vector to its non-sparse equivalent and vice-versa, thus making it possible to use a non-sparse procedure in its original sparse environment.

The new non-sparse vectors and system matrix are implemented in plato as global arrays. Each vector is an array of *tot_vars* doubles and the system matrix is stored in two 2-dimensional arrays each containing *tot_vars*·*tot_eqs* doubles. The first of these arrays, *ns_mat*, contains the matrix elements labeled *val* (refer to section 3.1). The second array, *ns_lu_mat*, contains the fields labeled *lu_val*. Note that for clarity all non-sparse vectors and routines, as well as the non-sparse system matrix have the prefix *ns_* added to their original names.

To illustrate the use of the conversion routines regard the routine *ludec_upd*. Suppose for now that this is the first plato routine that was substituted with its non-sparse equivalent *ns_ludec_upd*. Then by using the four mentioned routines it can be tested with the rest of plato still in its original form. Suppose that the call to the original routine looks like

```
ludec_upd( source_vec1, source_vec2 )
```

with *source_vec1* and *source_vec2* sparse vectors. Then after modification of *ludec_upd* to *ns_ludec_upd*, this call is replaced by

```
/* copy the sparse matrix and vectors to the fixed arrays: */
to_non_sparse_vec( source_vec1, ns_source_vec1 );
to_non_sparse_vec( source_vec2, ns_source_vec2 );
to_non_sparse_mat();

/* invoke the non-sparse version of ludec_upd: */
ns_ludec_upd( ns_source_vec1, source_vec2 );

/* build sparse data structures that contain the matrix and vectors: */
to_sparse_mat();
to_sparse_vec( ns_source_vec1, source_vec1 );
to_sparse_vec( ns_source_vec2, source_vec2 );
```

Note that at the time *ns_ludec_upd* is called, the system matrix is stored in the 2-dimensional arrays *ns_mat* and *ns_lu_mat*. At all other times, the system matrix is contained in a sparse data structure pointed to by arrays *row_head*, *col_head* and *diag*.

This method of isolating parts of plato and changing these individually makes it possible to modify plato procedure by procedure. Note that it is inevitable that the conversion routines take a time linear in the number of input elements, which makes the matrix conversion routines to have an efficiency of $O(n^2)$, with n the number of system matrix rows or columns. Thus the matrix conversion routines generally take up more time than the routines they are isolating. Therefore, to obtain a plato version that is more efficient than the original, it is necessary to change *all* sparse plato routines, at least up to a level that is high enough for these routines to be called relatively seldom. It turned out that indeed almost every sparse routine had to be changed to obtain a plato version that can simulate the tsp problem faster. To get an idea of the amount of work done, note that the total number of modified routines is 22.

Now it will be illustrated with some examples basically how the form of the original plato code looks and to what form this code was transformed. Because most of the time is spent in loops so these are treated here.

Firstly, there are pieces of code that use sparse vectors. These pieces contain loops like

```
/* vect_head points to the first element of the linked list
 * that constitutes a sparse vector
 */
for( ptr = vect_head; ptr != NIL; ptr = ptr->next )
{
    ptr->val = <some_expression(ptr->index)>;
}
```

The execution time for this piece of code is small for vectors that contain many zero-elements, because only non-zero elements are handled. For instance, if a vector with *tot_vars* elements has only 4 non-zero entries, then the loop has only four iterations. Loops of this form were modified to

```
for( index = 0; index < tot_vars; index++ )
{
    ns_vector[index] = <some_expression(index)>;
}
```

It will be clear that if there are few zero elements in the vector, this loop will be executed faster because less time is spent in each iteration.

Secondly there are pieces of code that use the sparse system matrix. These contain loops of the form

```
for( row = 0; row < tot_eqs; row++ )
{
    for( elmntp = row_head[row];
         elmntp != NIL;
         elmntp = elmntp->next )
    {
        elmntp->val = <some_expression>;
    }
}
```

All elements of the matrix are assigned the value of <some_expression> on a per row basis. Loops of this form are replaced with:

```
for( row = 0; row < tot_eqs; row++ )
    for( col = 0; col < tot_vars; col++ )
        ns_mat[row][col] = <some_expression>;
```

Furthermore, some routines now become redundant, like the one that is used to insert an element in a sparse vector. This saves much time, as will become clear when this routine is examined in some more detail. To insert an element in a sparse vector, the place at which to insert it must be found by scanning the linked list, a new sparse element must be created, old links must be broken and new links must be made. In contrast to this elaborate operation stands the insertion of a new element in a vector represented by an array, costing only one simple assignment operation.

That the non-sparse implementation more suits our purposes shows from the fact that *ns_ludec_upd* is about three times as fast as *ludec_upd*. For a more elaborate discussion of the performance gain refer to section 5.1.

4.3 Leafcell routine optimization

Here are discussed all changes that were made to the leafcell routines. The changes are motivated by observations like those in section 3.2.

The leafcell routines perform operations on the leafcells, and specifically on the leafcell matrices that are contained in the data structure that represents one leafcell. The three most expensive leafcell routines are found in the top part of Table 4.1. These routines form the core of the changed leafcell routines. It is strained that no attention is paid to the precise function of the procedures in the simulation process. Instead for each routine is examined what the code of the body looks like and how this code may be optimized, in accordance with the special structure of the leafcell sub matrices as deduced in section 3.2.

4.3.1 procedure i_calc_pwlder

This leafcell routine was considerably simplified. The changes that were made and the motivations for these changes can be abstracted as follows.

1. According to our neuron pwl model (refer to section 3.2) the number of dynamic variables (*N_U*) is always 1 (independent of the number of segments that is used to approach the non-linear curve of Fig. 3.3). Therefore each of the 3 loops with limit *N_U* is replaced by one assignment, namely the first iteration.
2. Also use is made of the fact that leafcell sub matrix $A_{31} = [0]$. With this knowledge a combination of two nested loops is replaced by one assignment.
3. *I_calc_pwlder* uses the non-sparse source vector *xbarvec*. This vector is accessed through a permutation array that is found in the leafcell passed to *i_calc_pwlder* as its argument. Because *xbarvec* is accessed rather often, in the original version of *i_calc_pwlder* it is copied to a local array that can be accessed directly, and therefore more quickly. As mentioned above, in the neural version of *i_calc_pwlder* some loops are eliminated. Because these

loops involve *xbarvec*, this vector is accessed less frequently. A consequence of this is that the time needed to initialize the local vector is now large compared to the gain obtained by using it, therefore this speed-up vector is eliminated.

Concluding it can be said that when the new version of *i_calc_pwlder* is compared with its original it is seen that indeed it has changed to a much simpler form, containing only two loops. One loop has *N_PWL* (the number of pwl segments) iterations, which is at most 5. Therefore not much time is spent in this loop. The other loop has more iterations, how this loop is accelerated is explained in section 4.4.

4.3.2 procedure i_calc_udotbar

Most of the changes made to this procedure are analog to the changes made to *i_calc_pwlder*:

1. Again the loops with *N_U* iterations are omitted.
2. No speed-up vector is used.
3. Since sub matrix $A_{22} = [-1]$, the loop that uses this matrix is replaced by a single assignment.

The changes made resulted in a version of *i_calc_udotbar* that contains only one loop.

4.3.3 procedure i_upd_module_vars

To this procedure, only one important simple change could be made:

1. Omit all loops with *N_U* iterations.

There remains an important loop that contains in its body an invocation of procedure *output_column_variable*. It is in this loop that most time is spent. Exactly how this problem is solved will be discussed in section 4.5.1.

4.4 Vectorization and concurrency

As stated earlier in this report, the new version of plato is especially adapted to make full use of the vector and concurrency capabilities of the available alliant fx/8 mini-super computer. In this section is treated how an additional performance gain was obtained by using vector/concurrency instructions. First it will be made clear what is meant by vectorization and concurrency. Then the actual adaptions will be discussed that were made to several routines to enable them to be executed vector/concurrent.

The alliant computer that is available has, amongst others, 8 computational elements (CE's) that can be used to execute processes. Four of these units are coupled in a so called *complex* that can be claimed as a whole by a certain process. The new version of plato claims this complex to obtain a more efficient simulation.

A CE contains special vector instructions that are capable of working on 32 elements at a time. Suppose that a certain loop is executed on one of the four CE's in the complex, say to initialize an array to all 0's. When this is done iteratively, normally one element is initialized with each iteration. However when using the special vector capability of the CE, with one instruction 32 elements are initialized. For

instance, when there are 384 elements in the array that must be initialized, using the vector instructions it takes $384/32 = 12$ time units to initialize the complete array, in stead of 384 time units. However the gain is not a factor 32 because the vector instructions impose an extra overhead.

It is also possible to spread iterations over the four CE's available in the complex. The iterations are then also executed concurrently, during each time unit each processor performs 32 initializations, raising the total performance to 128 initializations per time unit. So using the vector/concurrent capabilities of the alliant, we now need only 3 time units, in stead of the original 384 units.

It is noted that a few considerations must be kept in mind. Firstly, because of the overhead caused by vector/concurrent calls, the practical gain almost never exceeds a factor 4. Secondly it is remarked that to be able to be vectorized and/or called concurrent, loops and procedures must comply with stringent demands, the loop control structure, as well as the loop body must have an optimizable form. Furthermore the data used by concurrent procedures must not contain a critical section, the data used in concurrent calls must be mutually independent. For example, the various iterations in a loop must be mutually independent to be executed concurrently, which means that calculations made in an iteration may not depend on the results of calculations made in earlier iterations. This condition, amongst many others, puts an upper limit on the amount of code that may be vectorized and/or called concurrently. It is evident that because of these conditions many loops can not be vectorized or called concurrently at all.

The routines that were modified will now be treated.

4.4.1 The leafcell routines

The principle that justifies concurrent calls for some leafcell routines is that different invocations of these routines are mutually independent. This is easily understood when we observe how the information of the different leafcells is stored in plato: each leafcell has associated with it a different data structure. Insofar the leafcell routines are concerned these datastructures are mutually independent, they do not have any common variables. Therefore, a leafcell routine that changes or uses some data for all leafcells may just as well be called concurrently, handling different leafcells at the same time. Because the leafcell routines are called concurrently, the loops that occur in the routines themselves do not use concurrency, but are vectorized wherever possible. Abstracting the above, the general scheme followed for changing the leafcell routines is to vectorize the individual routines as much as possible, whereas calls to these routines are concurrent. Like this, optimal use is made of the possibilities of the alliant computer.

As deduced in section 4.3.1, only one inefficient loop remains in routine *i_calc_pwlder*. This loop is converted into a somewhat simpler form and vectorized, using special compiler pragmas supported by the alliant fxc compiler. Calls to *i_calc_pwlder* are made concurrently, so generally the four CE's in the complex are executing different invocations of this routine, while each CE uses vector instructions. *I_calc_pwlder* is called iteratively from routine *ns_handle_related_leafs*. Because in one iteration are also called other leafcell routines, calls to these routines too have to be made concurrently. Thus the routines that are invoked concurrently from *ns_handle_related_leafs* are *i_calc_pwlder*, *i_event* and *i_rec_event*.

For the other two expensive leafcell routines, it is not so easy to obtain a major performance gain. The one remaining loop in *i_calc_udotbar* was vectorized. The calls to *i_calc_udotbar* can not be made concurrently because these calls are made from various other leafcell routines, and therefore may not be made simultaneously.

It is not possible to vectorize the expensive loop in *i_upd_module_vars*, because in the loop body a call is made to a procedure that writes variables to a file. How *i_upd_module_vars* was improved is discussed in section 4.5.1.

4.4.2 Routine ns_non_sparse_solve

Routine ns_non_sparse_solve solves the circuit equations by doing forward/backward substitution on the LU-decomposition of the system matrix. Generally, the system that has to be solved is of the form

$$L \cdot U \cdot x = b$$

According to the standard forward/backward substitution process, x is solved in two steps:

1. solve y from $L \cdot y = b$
2. solve x from $U \cdot x = y$

Originally, the algorithms used for these steps were:

```
/* forward substitution: */
for( k=0; k<tot_eqs; k++ )
    if( b[k] == 0.0 ) continue;
    for( i=0; i<k; i++ )
        b[i] -= L[i][k] * b[k];
/* y[0] ... y[tot_eqs-1] are stored in b[0] ... b[tot_eqs-1] */

/* backward substitution: */
for( k=tot_eqs-1; k>=0; k-- )
{
    if( b[k] == 0.0 ) continue;
    b[k] /= U[k][k];
    for( i=k-1; i>=0; i-- )
        b[i] -= U[i][k] * b[k];
}
/* x[0] ... x[tot_eqs-1] are stored in b[0] ... b[tot_eqs-1] */
```

Note that for clarity the sparse techniques already are deleted from these algorithms. The advantage of the original algorithms is that if there are 0 elements in the vectors b or y, the inner loops can be skipped. But since, for reasons unknown, these algorithms are not optimizable, they are replaced by:

```
/* forward substitution: */
for( k=0; k<tot_eqs; k++ ) {
    y[k] = b[k];
    for( i=0; i<k; i++ )
        y[k] -= L[k][i]*y[i];
}
/* y[0]...y[tot_eqs-1] are calculated */


/* backward substitution: */
for( k=tot_eqs-1; k>=0; k--  ) {
    x[k] = y[k];
    for( i=k+1; i<tot_eqs; i++ )
        x[k] -= U[k][i]*x[i];
    x[k] /= U[k][k];
}
/* x[0]...x[tot_eqs-1] are calculated */
```

The inner loops were vectorized, but the iterations of the outer loops depend on the result of all previous iterations, and therefore these iterations can not be executed concurrently.

4.4.3 Routine ns_ludec_upd

The inner loops of this routine were vectorized, resulting in a more efficient version. Ns_ludec_upd can not be further optimized because the outer loops consist of interdependent iterations.

4.5 Additional optimizations

some additional optimizations were tried, which are discussed in this section.

4.5.1 Implementation of a lock/wait mechanism

The problem with procedure *i_upd_module_vars* is that it calls *output_column_variable* that writes to standard output. Because only one routine may write to standard output at a time, it is not possible to optimize *i_upd_module_variable* in its original form.

To solve this problem, a lock-unlock mechanism was implemented for procedure *i_upd_module_vars*. This ensures that only a single procedure is writing to standard output.

In procedure *i_upd_module_vars* occurs the following code:

```
lock( output );
output_column_variable( ... );
unlock( output );
```

Initially semaphore output is initialized to 1. When a lock operation is applied, the value of *output* is decreased with 1, so it becomes 0. Then one call is made to routine *output_column_variable*. If during this call another procedure is going to call *output_column_variable*, it first checks via the lock command the value of semaphore *output*. It finds that this value is 0 and therefore halts execution until

this variable is increased to 1 by the procedure that originally called it.

Via this scheme it is ensured that only one procedure uses the standard output device. Unfortunately tests showed that the lock mechanism implies so much overhead that the version of plato that features this mechanism is effectively slower that the previous version. Therefore this lock mechanism is not implemented.

4.5.2 Updating the global xvec

In the original form of plato, the global xvec, comprising the values of the all neuron inputs and outputs, is updated on a per leafcell basis. All leafcells are scanned and the part of the xvec that concerns the leafcell currently evaluated is updated. Because neural nets have many connections, there is a great deal of overlap in this xvec: because many neurons share connections, many times the same xvec entry is updated for different leafcells.

To eliminate this redundancy, the xvec is not updated on a per leafcell basis, but rather it is scanned once iteratively, using *tot_vars* iterations. This resulted in an additional performance gain.

4.6 Conclusions

Plato was modified according to the following scheme. Routines that are candidates for change are located with *gprof*. For the removal of the sparse techniques a *divide and conquer* like technique was used. Parts of code that needed to be changed were isolated and these parts were changed. According to this scheme it was possible to modify and test small parts of plato. This is necessary because plato is so complicated that it is hardly possible to modify it as a whole. For the removal of the sparse techniques four routines were written that isolate the various parts of sparse code. These isolated parts were changed and tested with the rest of plato still in its original form. Because the mentioned routines are expensive ($O(n^2)$, with n the number of neurons), all sparse code had to be removed, up to the highest level. Only then became negligible the degrade of performance gain introduced by these routines.

Special attention was paid to the leafcell routines. they were optimized, first using the special properties of the leafcell matrix and then using the special parallel/concurrent capabilities of the alliant fx/8 computer. This is possible because each leafcell routine performs the same sequence of operations on every leafcell. Information in each leafcell is independent of all other leafcells so these routines are called concurrently whenever possible. Iteration loops within the routines were vectorized whenever the loop body allowed it. It is noted that the performance gain obtained with the vector/concurrent capabilities becomes notable for rather large problem sizes (100 neurons) because a lot of extra overhead is introduced. For these problem sizes, a considerable gain was obtained, as will be further discussed in the next chapter.

Also vectorized were the routines that perform an update on the lu decomposition of the system matrix and the routine that solves x from $Ax = b$ using a forward-backward substitution process.

Finally, an additional optimizations was implemented, concerning a more efficient update of the global *xvec*. It turned out that the implementation of a lock/unlock mechanism is not effective because the large overhead inherent to the use of the lock/wait instructions.

5. Testresults

After each major step in changing plato some testcases were run to evaluate the performance. The order in which these tests are discussed is analog to the order in which the modifications were treated in the previous chapter. Besides tests used to evaluate the performance gain of nsplato due to implemented improvements, also tested is how the simulation speed is influenced by various other parameters like integration method or -precision. Furthermore is evaluated the graph coloring problem as an additional testcase.

5.1 Performance before implementation of vector/concurrency

Here the performance gain is evaluated that is obtained after removal of the sparse techniques and optimizations of the leafcell routines. The reason for an evaluation at this stage is that firstly it is interesting to trace the reasons for some observations, and secondly here the performance of the alliant computer can be compared with that of the hp computer. The results of several testruns are abstracted in Fig. 5.1.

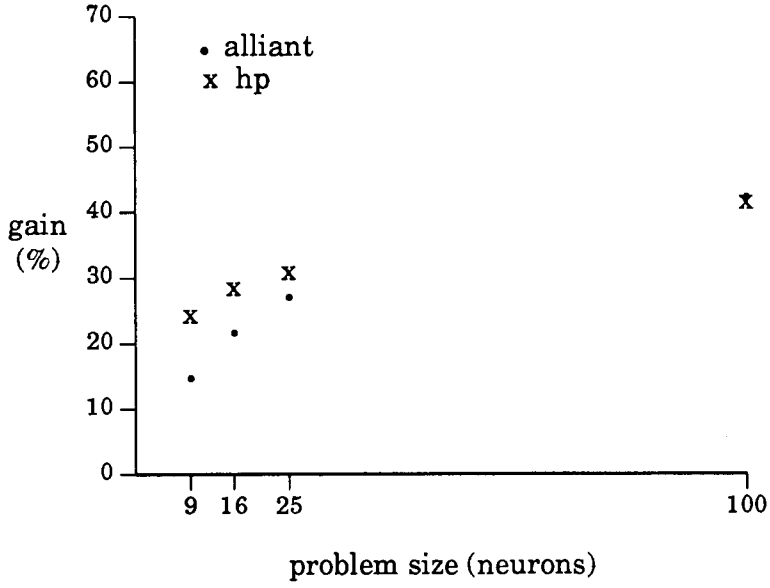

Figure 5.1. *Performance gain after removal of the sparse techniques and optimization of the leafcell procedures for the hp and alliant computers.*

The performance gain is given for several problem sizes. Note that a performance gain of 80% means that the new version of plato is 5 times as fast as the old version. It is interesting to observe that with increasing problem size, also the performance gain increases. This is caused mainly by the fact that the amount of time relative to total execution time that is spent in procedure *ludec_upd* increases quadratically

with the number of neurons (this is not the case for the leafcell routines). The leafcell routines, together with routine *ludec_upd* take up a large part of total execution time. It will be clear that if the number of neurons is larger, then the relative amount of time that is spent in *ludec_upd* increases. The new version of this routine itself is about 4 times as quick as its old version. It will be clear that if there is more time spent in this procedure, which is true for increasing problem, then the gain obtained for this procedure will have a more pronounced effect on the gain in total simulation time. Also the overhead needed for eg. calling routines and initializing variables relatively decreases with increasing problem size. This also contributes to an increase performance gain for larger problem sizes.

Furthermore it is interesting that while the hp computer originally has a large advantage over the alliant, for large problem sizes the alliant performs even better that the hp. This is caused by the much larger working memory of alliant, which is about 120 Mbytes, as compared to 16 Mbytes for the hp. The amount of memory that is needed for a simulation of 100 neurons is about 10 Mbytes. Accesses to this memory result on the hp in a lot of swapping of memory segments from main memory to disk and vice versa. The alliant is less hampered by this swapping activity because it has a larger working memory.

5.2 Final performance

Here the performance of the alliant plato version is evaluated. In this final version, all modifications are implemented that are discussed in the previous chapter. Again various instances of the tsp problem were used as testcases. The result of the simulation is abstracted in Fig. 5.2.

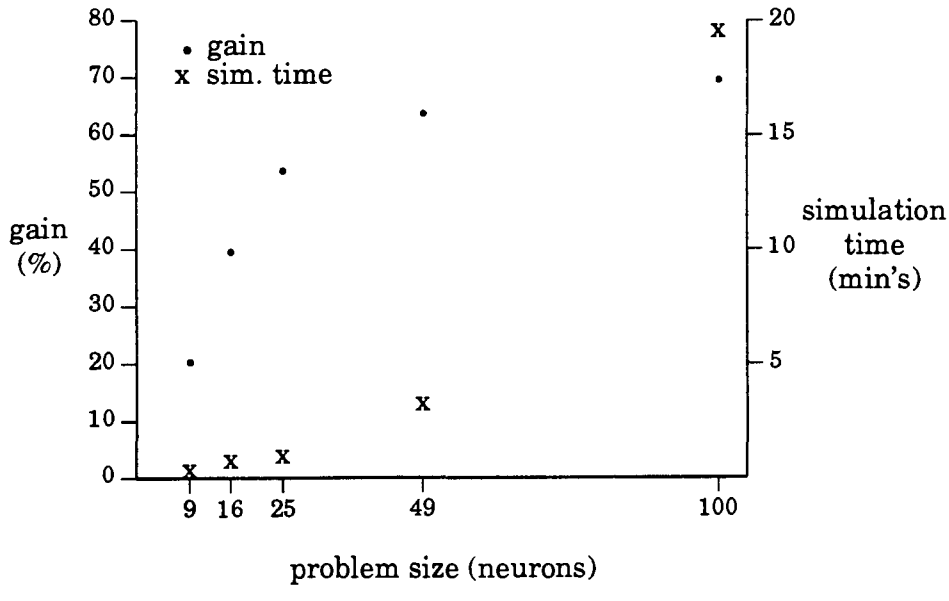

Figure 5.2. *Performance of the final version of plato on the alliant computer.*

In this figure are shown for the alliant computer the gain and the simulation time for various instances of the tsp problem. Again the obtained performance gain increases with problem size, the reasons for this effect were discussed in the previous section. The total simulation time increases dramatically with problem size. This is easily explained, when we consider the connectivity of the neural net that is used to simulate the tsp problem. In section 2.3 the terms of the connection matrix were deduced. From constant C of the energy function result connections from each neuron to all other neurons, making the total amount of connections $O(n^2)$ with n the number of neurons, or $O(m^4)$ with m the number of cities. The number of connections determines the number of entries in the system matrix and the amount of work that will have to be done in the routines that use this matrix. In Table 5.1 are listed the 10 most expensive procedures of the final plato version.

TABLE 5.1. *The 10 most expensive procedures after plato optimization. Problem size is 100 neurons, integration method is TR and accuracy is* 10^{-3}. *A 3-segment model is used.*

% time	calls	Total (ms/call)	procedure name
38.5	6338	58.22	ns_ludec_upd
16.2	7183	21.01	ns_non_sparse_solve
15.3	7183	21.96	ns_nonzero_list
3.9	437321	0.08	cvec_i_calc_pwlder
3.8	455459	0.08	i_calc_udotbar
2.4	164423	0.14	YYlook
2.3	527800	0.04	output_column_variable
2.0	19143	2.15	i_upd_xvec
1.6	455537	0.03	i_upd_module_vars
1.2	6248	1.84	ns_i_rank1_upd

The additional optimizations to other routines increased the time spent in *ns_ludec_upd* to 38.5%. Next comes routine *ns_non_sparse_solve* which takes 16.2% of the total execution time. Thus 54.7% of the time is spent in routines which take a time quadratically in the number of cities. Therefore it is logical that the simulation time rises very hard with problem size, this time is $O(n^2)$. It is concluded that if a completely connected network is simulated, the problem size above which simulation times become too large to be practical is about 100 neurons.

Note that the top five of Table 5.1 is formed by routines that are not further optimizable. Since these routines take up 77.7% of the total simulation time, it is concluded that plato cannot be optimized further by changing individual routines. Rather it becomes necessary to evaluate the simulation process itself, perhaps optimizing it for neural nets.

5.3 Additional tsp test runs

Apart from the tests used to check the performance gain obtained by modifying plato, tests were made to evaluate the effects of changes in the number of pwl segments, the method of integration and the integration accuracy.

5.3.1 Changing the number of pwl segments

As shown in section 3.2, the number of linear segments in the used pwl model is 3. To deduce if and how the simulation time is influenced by the number of linear segments, another model was constructed that approximates the neuron nonlinearity with 5 segments. The idea is that when the nonlinearity is approximated with greater precision, the integration step can become larger, resulting in a more time-efficient simulation.

It showed out that the simulation time is not very dependent of the number of segments in the pwl model. Apparently the 3-segment model approximates the nonlinearity to such a great precision that the simulation step does not depend very much on the deviations inherent to this model.

5.3.2 Changing the integration method

Plato offers the possibility to use various methods of integration. Some tests were made to evaluate the effect of the method of integration on total simulation time.

In table 5.2 the results are abstracted.

TABLE 5.2. *Simulation time for the 4-city tsp problem, a 3-segment pwl model for various integration methods.*

Simulation time (secs)				
TR	BE	BDF	ACT	FE
24.5	24.8	29.1	31.7	80.3

Tabulated are the simulation times for the integration methods trapezium, backward Euler, backward difference, acontractive and forward Euler. It is seen that the simulation time is greatly affected by the choice of integration method. Forward Euler gives very bad results, as is to be expected from such an unstable integration method. If the used integration method is so poor, plato has to take many small steps to reach a stable solution, resulting in a very time consuming simulation process. The more stable the integration method, the bigger the time step and the shorter the simulation time.

5.3.3 Changing the integration accuracy

Besides the method of integration, also the accuracy of integration may be adjusted. In Fig 5.3 is shown the dependency of simulation time on integration accuracy. Note that the maximum signal value (maximum output value of any neuron) is 1.

As was expected for neural nets, it is not necessary to use a great precision. According to Fig. 5.3, precision may even be as low as one tenth of the actual maximum signal value. For neural nets the result of the simulation is just a set of high or low values.

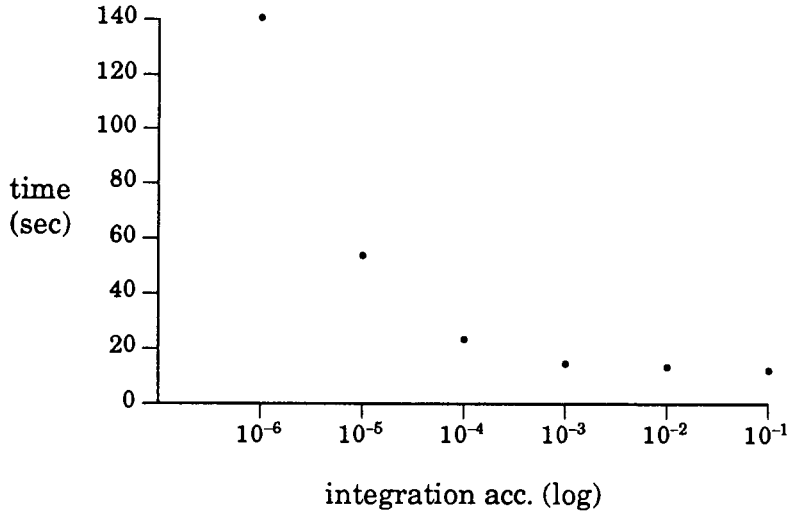

Figure 5.3. *Simulation time for the 4-city tsp problem, a 3-segment pwl model, TR integration rule as a function of integration accuracy.*

It is noted that for larger problem sizes, the relationship between simulation time and integration accuracy is not so beautiful as perhaps is suggested by Fig. 5.3. This becomes clear when Fig. 5.4 is evaluated.

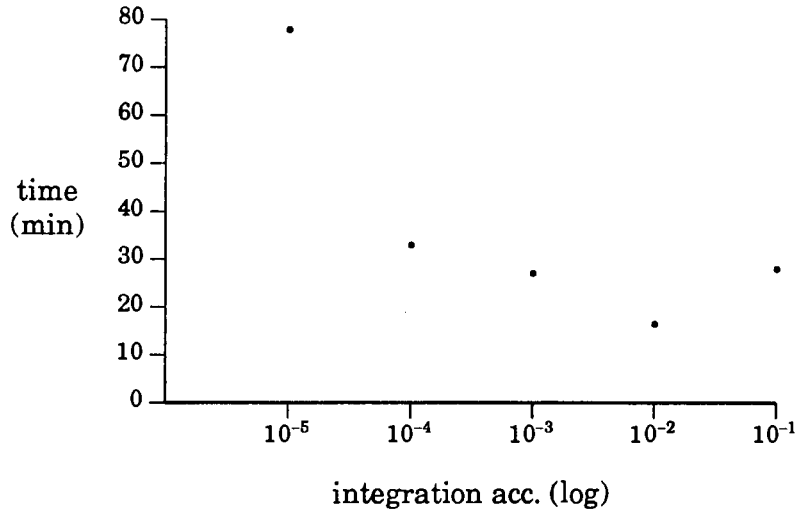

Figure 5.4. *Simulation time for the 10-city tsp problem, a 3-segment pwl model, TR integration rule as a function of integration accuracy.*

Basically the simulation time decreases with decreasing integration accuracy, but when this accuracy becomes too small the simulation time increases again. For this accuracy, due to the great number of connections with a 100 neuron net, the simulation does not converge to a correct solution.

It can be concluded that an accuracy of 10^{-2} suits best for most problem sizes.

5.4 Conclusions

In section 5.1 is shown that for smaller problem sizes, the plato version without vector/concurrency optimizations is faster, although this advantage decreases with larger problem sizes, and eventually turns into a slight disadvantage for a problem size of 100 neurons. From this test the importance can be deduced of the large main memory of the alliant computer. It appears that for a large problem size the time spent in memory swapping is as important as the time spent in the actual simulation process.

Tests of the final performance show that performance gain rises with problem size, being about 70% for a 100 neuron problem. But, logically, also the simulation time rises with problem size, as is explained in section 5.2. For a 100 neuron tsp problem, the simulation time is about 20 minutes. Because (Fig. 5.2) the simulation time rises sharply if problem size exceeds 100 neurons, it is considered to be impractical to simulate with plato tsp problems larger than about 100 neurons. Therefore the maximum achievable gain is 70% for these problems.

Also shown in section 5.2 is that in the final plato version for a 100 neuron tsp problem 77.7% of total simulation time is spent in 5 procedures that cannot easily be further improved. It is evident that large performance improvements must be found in modifying the simulation process itself, in stead of optimizing individual routines.

Also discussed is the dependence of the simulation efficiency on other factors. For instance, the chosen integration method influences the simulation speed. This is logical, because if the simulation method has poor convergence properties (which is the case with forward Euler), then the simulator will choose its timestep very small, resulting in a lot of steps and thus a lengthy simulation process. The best integration method is the trapezium method.

The simulation process can be speeded up by choosing the integration accuracy as small as possible. This is possible because for neural nets not the actual form of the output signals is important, but only the end result matters, being a collection of high or low values. It appears that the integration accuracy which for most problem sizes suits our purposes is 10^{-2}.

The simulation speed is hardly afflicted by a change from a 3-segment pwl model to a 5-segment pwl model. Apparently the 3-segment model is precise enough that the time step can be chosen rather large. This time step then does not change for a 5-segment model.

6. Final conclusions and recommendations

The most important conclusion to be drawn is that for neural nets a performance gain of maximal 70% can be obtained, and that it is not easy to improve this gain much further. The gain is highest for large problem sizes, but since also the simulation time increases very fast (about quadratically) with problem size, the largest problem that can be simulated in reasonable time (in the order of a couple of tens of minutes) is 100 neurons. During the simulation of this problem, about 77% of the total simulation time is spent in five procedures that already were optimized to a great extent. Therefore it is concluded that a further significant performance gain is not easily obtainable from further enhancing the performance of individual routines. Rather the simulation process itself should be considered, although the writer is a bit skeptical about this, having experienced that the original plato already is optimized in various ways.

Of course not only the efficiency of the program code is important for the simulation speed, also some user-adjustable parameters have much influence. Most important are the integration method and integration accuracy. Since the solution that is determined with a neural net depends solely on which neurons in the final state are active and which are not, the timestep may be chosen so great that actual information about the precise time evolution of the individual signals is lost to some extend. It was determined that the accuracy of integration may be as low as 10^{-2}. Also important is the stability of the used integration method. From 5 different methods, the trapezium method proved to result in the most time-efficient simulation. Not surprisingly, forward Euler proved to be by far the worst. The number of segments of the piecewise linear model has only to be 3. A model with 5 segments proved not to result in a faster simulation, apparently the integration step chosen by plato for the 3-segment model is not much smaller than the time steps chosen with the 5-segment model.

If it is decided that plato is going to be used to simulate neural nets, then some thought must be given about the conversion of a functional neural net description to a suitable ndml file. It even may prove to be useful to entirely omit the ndml intermediate stage and convert a neural net description directly into a datastructure (consisting of leafcells and a system matrix) that can be used by plato. Already no ndml compiler is needed because the plato input is generated by a custom-written program. If plato is modified as described above also the part can be omitted that transforms the input into the data structures, resulting in a decrease in total overhead. The functional input to plato could be stated in a simple special purpose programming language, that enables the user to enter the neural net description and parameters, as well as the parameters that have to be passed to plato, perhaps also eliminating the need for a task file.

Also the user interface at the other end could perhaps be adjusted especially for neural net simulation. While Superplog is a nice program, it just offers a bit too much information, taking much time in the process. A way could be found to reduce the size of the large plato signal output and present the results to the user in a more abbreviated, handsome way. For instance it would be nice to be able to deduce quickly the final values of the neuron outputs, in stead of having to walk through

the entire range of signals using Superplog.

A final observation can be made about the fx/c compiler with which the plato source code was optimized to take advantage of the vector/concurrent capabilities of the available alliant fx/8 computer. It turned out that it is not always very easy to change the code into a form that can be digested by the compiler into a vectorized and/or concurrent form. The actual form of the code, and the use of special; optimization pragmas is a very delicate business. For example, if a loop is vectorizable, changing the order of the iterations causes the compiler to conclude that the loop is not optimizable any more.

References

[1] Bokhoven, W.M.G. van , "Piecewise-Linear Modelling and Analysis," Ph.D. Thesis, Eindhoven, The Netherlands, May 1981.

[2] Hopfield, J.J. and D.W. Tank, "Neural Computation of Decisions in Optimization Problems," *Biological Cybernetics*, vol. 52, pp. 141-152, 1985.

[3] Garey, M.R. and Johnson, D.S., *Computers and intractability: A guide to the theory of NP-Completeness,* pp. 18-23, W.H. Freeman and company, New York, 1979.

[4] Janssen, G.L.J.M., "Network Description and Modeling Language - NDML," in *The Integrated Circuit Design Book*, ed. P. Dewilde, Delft University Press, 1986.

[5] Jess, J.A.G., "Piecewise Linear Models for Nonlinear Dynamic Systems," *Frequenz*, vol. 42, no. 2/3, pp. 71-78.

[6] Shackleford, J.B., "Neural Data Structures: Programming with Neurons," *Hewlett-Packard Journal*, vol. 40, no. 3, pp. 69-78, June 1989.

[7] Stiphout, M.T. van, "A piecewise Linear Analysis Tool for Mixed-Level Circuit Simulation," Ph.D. Thesis, Eindhoven, the Netherlands, 1990.

Appendix 1: example tsp description

Here the city input and output for the 3-city tsp problem is listed. This way the reader can see how this simple input is converted into the ndml description of this problem. The file that serves as city input was named *map3*, but could have had any name. It is listed below. The c-like comment is added for clarity but may not appear in the file that serves as the actual city input!

```
nn              /* compound name */
9               /* number of neurons */
1.1             /* A */
1               /* B */
0.33            /* C */
0.707           /* D */
0.1, 0.1        /* coordinates of first city */
0.5, 0.5        /* coordinates of second city */
0.1, 0.9        /* coordinates of third city */
2               /* neuron offset value offset */
0.2             /* delay Δ */
0.01            /* neuron gain G */
0.1111111111    /* initial output value init_out */
0.001           /* maximum random noise on initial value */
```

The ndml file was generated by the command

city map3

to which city responds with

<city>: plato input is saved in file nn.ndml

Note that city took from its input file the compound name and added to it the suffix *.ndml*. Furthermore, it added comment that resembles the terminology in which the ndml language is defined. City also produces simple error messages when the number of arguments is not 1 or if a file cannot be found.

```
(* compound_system: *)
compound nn__1();

(* instance_declaration_part: *)
instance
cell1 : adder9__1;
cell2 : adder9__2;
cell3 : adder9__3;
cell4 : adder9__4;
cell5 : adder9__5;
cell6 : adder9__6;
cell7 : adder9__7;
cell8 : adder9__8;
cell9 : adder9__9;

(* net_declaration_part: *)
net
out[0],out[1],out[2],out[3],out[4],out[5],out[6],out[7],out[8] : signal;

(* compound_body: *)
begin
cell1(out[0],out[1],out[2],out[3],out[4],out[5],out[6],out[7],out[8],out[0]);
cell2(out[0],out[1],out[2],out[3],out[4],out[5],out[6],out[7],out[8],out[1]);
cell3(out[0],out[1],out[2],out[3],out[4],out[5],out[6],out[7],out[8],out[2]);
cell4(out[0],out[1],out[2],out[3],out[4],out[5],out[6],out[7],out[8],out[3]);
cell5(out[0],out[1],out[2],out[3],out[4],out[5],out[6],out[7],out[8],out[4]);
cell6(out[0],out[1],out[2],out[3],out[4],out[5],out[6],out[7],out[8],out[5]);
cell7(out[0],out[1],out[2],out[3],out[4],out[5],out[6],out[7],out[8],out[6]);
cell8(out[0],out[1],out[2],out[3],out[4],out[5],out[6],out[7],out[8],out[7]);
cell9(out[0],out[1],out[2],out[3],out[4],out[5],out[6],out[7],out[8],out[8]);
end;

(* leafcell_definitions: *)

(* leafcell #9 (of 9): *)
leafcell adder9__9(in[0],in[1],in[2],in[3],in[4],in[5],in[6],in[7],in[8],out
begin
var(pl.3,1);
pl.3=-1,2.000000000;
[pl.3,in[0]]:=-0.895600021;
[pl.3,in[1]]:=-0.895600021;
[pl.3,in[2]]:=-1.330000043;
[pl.3,in[3]]:=-0.729939580;
[pl.3,in[4]]:=-0.729939580;
[pl.3,in[5]]:=-1.330000043;
[pl.3,in[6]]:=-1.430000067;
[pl.3,in[7]]:=-1.430000067;
[pl.3,in[8]]:=-1.430000067;
var(pl.3,out,u.1,pl.1,pl.2,1);
zero.1=,-1,50.000000000,50.000000000,-50.000000000,0.111624964;
```

```
du.1=5.000000000,,-5.000000000,,,;
pl.1=,,1,1,,0.002680070;
pl.2=,,-1,,1,0.017319930;
remove[pl.3,pl.3];
end;

(* leafcell #8 (of 9): *)
leafcell adder9__8(in[0],in[1],in[2],in[3],in[4],in[5],in[6],in[7],in[8],out
begin
var(pl.3,1);
pl.3=-1,2.000000000;
[pl.3,in[0]]:=-0.895600021;
[pl.3,in[1]]:=-1.330000043;
[pl.3,in[2]]:=-0.895600021;
[pl.3,in[3]]:=-0.729939580;
[pl.3,in[4]]:=-1.330000043;
[pl.3,in[5]]:=-0.729939580;
[pl.3,in[6]]:=-1.430000067;
[pl.3,in[7]]:=-1.430000067;
[pl.3,in[8]]:=-1.430000067;
var(pl.3,out,u.1,pl.1,pl.2,1);
zero.1=,-1,50.000000000,50.000000000,-50.000000000,0.111445367;
du.1=5.000000000,,-5.000000000,,,;
pl.1=,,1,1,,0.002584895;
pl.2=,,-1,,1,0.017415105;
remove[pl.3,pl.3];
end;

(* leafcell #7 (of 9): *)
leafcell adder9__7(in[0],in[1],in[2],in[3],in[4],in[5],in[6],in[7],in[8],out
begin
var(pl.3,1);
pl.3=-1,2.000000000;
[pl.3,in[0]]:=-1.330000043;
[pl.3,in[1]]:=-0.895600021;
[pl.3,in[2]]:=-0.895600021;
[pl.3,in[3]]:=-1.330000043;
[pl.3,in[4]]:=-0.729939580;
[pl.3,in[5]]:=-0.729939580;
[pl.3,in[6]]:=-1.430000067;
[pl.3,in[7]]:=-1.430000067;
[pl.3,in[8]]:=-1.430000067;
var(pl.3,out,u.1,pl.1,pl.2,1);
zero.1=,-1,50.000000000,50.000000000,-50.000000000,0.111338496;
du.1=5.000000000,,-5.000000000,,,;
pl.1=,,1,1,,0.003029871;
pl.2=,,-1,,1,0.016970129;
remove[pl.3,pl.3];
end;
```

```
(* leafcell #6 (of 9): *)
leafcell adder9__6(in[0],in[1],in[2],in[3],in[4],in[5],in[6],in[7],in[8],out
begin
var(pl.3,1);
pl.3=-1,2.000000000;
[pl.3,in[0]]:=-0.729939640;
[pl.3,in[1]]:=-0.729939640;
[pl.3,in[2]]:=-1.330000043;
[pl.3,in[3]]:=-1.430000067;
[pl.3,in[4]]:=-1.430000067;
[pl.3,in[5]]:=-1.430000067;
[pl.3,in[6]]:=-0.729939580;
[pl.3,in[7]]:=-0.729939580;
[pl.3,in[8]]:=-1.330000043;
var(pl.3,out,u.1,pl.1,pl.2,1);
zero.1=,-1,50.000000000,50.000000000,-50.000000000,0.111964814;
du.1=5.000000000,,-5.000000000,,,;
pl.1=,,1,1,,0.003080056;
pl.2=,,-1,,1,0.016919944;
remove[pl.3,pl.3];
end;

(* leafcell #5 (of 9): *)
leafcell adder9__5(in[0],in[1],in[2],in[3],in[4],in[5],in[6],in[7],in[8],out
begin
var(pl.3,1);
pl.3=-1,2.000000000;
[pl.3,in[0]]:=-0.729939640;
[pl.3,in[1]]:=-1.330000043;
[pl.3,in[2]]:=-0.729939640;
[pl.3,in[3]]:=-1.430000067;
[pl.3,in[4]]:=-1.430000067;
[pl.3,in[5]]:=-1.430000067;
[pl.3,in[6]]:=-0.729939580;
[pl.3,in[7]]:=-1.330000043;
[pl.3,in[8]]:=-0.729939580;
var(pl.3,out,u.1,pl.1,pl.2,1);
zero.1=,-1,50.000000000,50.000000000,-50.000000000,0.112073392;
du.1=5.000000000,,-5.000000000,,,;
pl.1=,,1,1,,0.002411512;
pl.2=,,-1,,1,0.017588488;
remove[pl.3,pl.3];
end;

(* leafcell #4 (of 9): *)
leafcell adder9__4(in[0],in[1],in[2],in[3],in[4],in[5],in[6],in[7],in[8],out
begin
var(pl.3,1);
pl.3=-1,2.000000000;
[pl.3,in[0]]:=-1.330000043;
```

```
[pl.3,in[1]]:=-0.729939640;
[pl.3,in[2]]:=-0.729939640;
[pl.3,in[3]]:=-1.430000067;
[pl.3,in[4]]:=-1.430000067;
[pl.3,in[5]]:=-1.430000067;
[pl.3,in[6]]:=-1.330000043;
[pl.3,in[7]]:=-0.729939580;
[pl.3,in[8]]:=-0.729939580;
var(pl.3,out,u.1,pl.1,pl.2,1);
zero.1=,-1,50.000000000,50.000000000,-50.000000000,0.111363158;
du.1=5.000000000,,-5.000000000,,,;
pl.1=,,1,1,,0.002232756;
pl.2=,,-1,,1,0.017767243;
remove[pl.3,pl.3];
end;

(* leafcell #3 (of 9): *)
leafcell adder9__3(in[0],in[1],in[2],in[3],in[4],in[5],in[6],in[7],in[8],out
begin
var(pl.3,1);
pl.3=-1,2.000000000;
[pl.3,in[0]]:=-1.430000067;
[pl.3,in[1]]:=-1.430000067;
[pl.3,in[2]]:=-1.430000067;
[pl.3,in[3]]:=-0.729939640;
[pl.3,in[4]]:=-0.729939640;
[pl.3,in[5]]:=-1.330000043;
[pl.3,in[6]]:=-0.895600021;
[pl.3,in[7]]:=-0.895600021;
[pl.3,in[8]]:=-1.330000043;
var(pl.3,out,u.1,pl.1,pl.2,1);
zero.1=,-1,50.000000000,50.000000000,-50.000000000,0.111606687;
du.1=5.000000000,,-5.000000000,,,;
pl.1=,,1,1,,0.002874987;
pl.2=,,-1,,1,0.017125013;
remove[pl.3,pl.3];
end;

(* leafcell #2 (of 9): *)
leafcell adder9__2(in[0],in[1],in[2],in[3],in[4],in[5],in[6],in[7],in[8],out
begin
var(pl.3,1);
pl.3=-1,2.000000000;
[pl.3,in[0]]:=-1.430000067;
[pl.3,in[1]]:=-1.430000067;
[pl.3,in[2]]:=-1.430000067;
[pl.3,in[3]]:=-0.729939640;
[pl.3,in[4]]:=-1.330000043;
[pl.3,in[5]]:=-0.729939640;
[pl.3,in[6]]:=-0.895600021;
```

```
[pl.3,in[7]]:=-1.330000043;
[pl.3,in[8]]:=-0.895600021;
var(pl.3,out,u.1,pl.1,pl.2,1);
zero.1=,-1,50.000000000,50.000000000,-50.000000000,0.111663207;
du.1=5.000000000,,-5.000000000,,,;
pl.1=,,1,1,,0.002552844;
pl.2=,,-1,,1,0.017447155;
remove[pl.3,pl.3];
end;

(* leafcell #1 (of 9): *)
leafcell adder9__1(in[0],in[1],in[2],in[3],in[4],in[5],in[6],in[7],in[8],out
begin
var(pl.3,1);
pl.3=-1,2.000000000;
[pl.3,in[0]]:=-1.430000067;
[pl.3,in[1]]:=-1.430000067;
[pl.3,in[2]]:=-1.430000067;
[pl.3,in[3]]:=-1.330000043;
[pl.3,in[4]]:=-0.729939640;
[pl.3,in[5]]:=-0.729939640;
[pl.3,in[6]]:=-1.330000043;
[pl.3,in[7]]:=-0.895600021;
[pl.3,in[8]]:=-0.895600021;
var(pl.3,out,u.1,pl.1,pl.2,1);
zero.1=,-1,50.000000000,50.000000000,-50.000000000,0.111298367;
du.1=5.000000000,,-5.000000000,,,;
pl.1=,,1,1,,0.002297470;
pl.2=,,-1,,1,0.017702529;
remove[pl.3,pl.3];
end;
```