Dept. Physics
Heriot-Watt University

Optoelectronic Neural Network Demonstrator:

Program and Results

Yves Randle
July 2002

Contents:

Introduction:

The present document is a report of my work on the optoelectronic neural network demonstrator, which I did during a six-month project in the *Optoelectronic Neural Network* team at the *Physics Department* of *Heriot-Watt University*. My project lasted from February 2002 to July 2002.

When I arrived, the demonstrator was already built but had never been run for neural network computation and scheduling. It is important for the understanding of this report to read the following:

- "Optically Interconnected Computing systems": thesis written by *Keith Symington* in November 2001. Keith, who constructed the demonstrator, explains its functioning as well as more general material about optical solutions for interconnections and neural networks for switching.
- "NOSC Electronic components": describes the electronic components used in the demonstrator.
- "VCSEL Array and Digital Driver": describes the theory and design behind a digital VCSEL (vertical cavity surface emitting laser) driver unit including the profile of an associated VCSEL array.
- "DSP Analogue to Digital Prototype Board – V2.00": describes the electronic boards that interface with the four DSP (digital signal processor) starter kits and carry out the analogue to digital conversion.
- "DSKController V2.00 Module": describes the C++ module used to control the DSPs using the computer.
- "TMS320C5x User's Guide": User's guide of the DSPs, with the assembly language tools.
- "TMS320C5x DSP Starter Kit User's Guide": User's guide of the DSP starter kits used in the demonstrator.

My work was divided into three main parts:

- Write a program in assembly language to run the DSPs in the demonstrator.
- Write a program in Borland C++ to run the demonstrator from the computer.
- Run the demonstrator and evaluate its performance for scheduling.

All the files relative to the programs needed to run the demonstrator can be found in the directory: *C:\Yves\NNetDem V1.00*

I) DSP program (nnetwork.asm):

This part first explains parts of the code in the DSP assembly language, and then describes the different parts of the program needed to run the DSPs, called "nnetwork.asm". The code is enclosed in appendix 1.

1) Assembly language:

These are a few useful things to know about the TMS320C5x DSP starter kit to understand the DSP program:

a) Use of the accumulator:

The memory of the DSPs is composed of words. Each word is composed of 2 bytes (16 bits). The accumulator is longer: it has the size of two words (32 bits). Both the higher part (most significant half) and the lower part (least significant half) of the accumulator can be stored independently in memory spaces. A word can also be left-shifted when loaded into the accumulator. This allows to do two things:

- **Divide by 2^n,** with n an integer smaller than 16. For example to divide a value by 16, the value is first loaded from the memory into the accumulator and in the same time shifted 12 bits to the left. This means that the value is multiplied by $2^{12} = 4096$. The most significant half of the accumulator is then stored back into memory. Not considering the lower part of the accumulator is equivalent to dividing by $2^{16} = 65536$. So the final value has been divided by 65536 / 4096 = 16.
  For example:

```
LAC    *, 12, AR0    ; Load accumulator with the word shifted.
SACH   *, 0, AR0     ; Store the higher part of the accumulator.
```

- **Separate two bytes contained in a single word.** The word is first loaded from the memory into the accumulator and in the same time shifted 8 bits to the left. Each byte is then in a different half of the accumulator and they can be stored independently in two different memory spaces. One of them then has to be shifted 8 bits to the right, which is equivalent to dividing by 256 as explained before.
  For example:

```
LAC    *, 8, AR0     ; Load accumulator with the word shifted.
SACH   *+, 0, AR0    ; Store the higher part of the accumulator.
SACL   *, 0, AR0     ; Store the lower part of the accumulator.
LAC    *, 8, AR0     ; Load and shift the second byte.
SACH   *-, 0, AR0    ; Store the second byte again.
```

b) Initialise a matrix:

An n×m matrix of values is represented n*m values, one after the other. To initialise such a matrix, the value n*m is loaded into the accumulator. An auxiliary register

(ARx) is loaded with the address of the first value of the matrix. The current matrix value (as indicated by the auxiliary register) is initialised and in the same time the auxiliary register is incremented. The value in the accumulator is decremented. If the value in the accumulator is greater than 0, the program branches the initialisation of the current matrix value again, and so on.
For example:

```
        LAR     AR0, #matrix  ; Load AR0 with matrix location.
        LAC     #00010h       ; Load accumulator with value 16.
init:   SPLK    #0, *+, AR0   ; Store 0 in current matrix value and
                                increment AR0.
        SUB     #1            ; Subtract 1 from accumulator.
        BGZ     init          ; Branch label "init" if accumulator
                                is greater than 0.
```

c) Compare two values.

Subtract one value from the other and then compare the result to zero using a conditional branch.
For example:

```
LAC     value1      ; Load accumulator with first value.
SUB     value2      ; Subtract second value.
BLZ     next        ; Branch label "next" if the result is lower
                      than 0, otherwise the next instruction is
                      executed.
```

d) Remarks:

- It is preferable not to use the instruction "SAMM" to address a memory-mapped register like ARx. It involves problems because of the time needed by the DSP to address the memory-mapped registers.
- All the input instructions "IN" had to be written before the address 0x0b00 in the program memory in order to work as expected.

2) DSP initialisation:

The code for DSK board initialisation has to be written at the beginning of each part of the program.

3) Switch off the VCSELs:

This is the first part of the program. When run, it sends a word whose value is 0 to the output port. This switches the VCSELs off.

4) Calibration:

The calibration is an averaged measure of the background noise for each one of the 64 detectors, and an averaged measure of the zeroth order for each detector when the corresponding VCSEL is on. The calibration part of the program will be carried out

for each one of the 4 DSPs, one at a time. The calibration does 8 times the same thing: one time for each of the 8 channels of the ADCs. When one ADC channel is called, two ADCs are called simultaneously and the measures from two detectors are stored within a single word in the DSP memory.

For each value, 256 measures are done to give an accurate average. For each channel of the ADCs, the averaging is done 3 times: the first time with no VCSEL on to give the background noise value for the two detectors, the second time with the VCSEL corresponding to the first detector on and the third time with the VCSEL corresponding to the second detector on.

To know which VCSEL should be switched on when, we use the "selection words" that give a relation between the position of the VCSELs in the VCSEL array and the mapping of the artificial neurons in the DSP memory.

Here is a representation of the VCSEL array; a number is given to each VCSEL:

	1	2	3	4	5	6	7	8
1	1	2	3	4	5	6	7	8
2	9	10	11	12	13	14	15	16
3	17	18	19	20	21	22	23	24
4	25	26	27	28	29	30	31	32
5	33	34	35	36	37	38	39	40
6	41	42	43	44	45	46	47	48
7	49	50	51	52	53	54	55	56
8	57	58	59	60	61	62	63	64

The same representation can be used for the corresponding neural network or detector array. Indeed, each neuron has a corresponding VCSEL and a corresponding detector.

Each DSP controls two rows in the VCSEL array. For example the first DSP controls the following VCSELs:

	1	2	3	4	5	6	7	8
1	1	2	3	4	5	6	7	8
2	9	10	11	12	13	14	15	16

Each DSP controls 16 VCSELs thanks to an output word. Each bit in the output word controls a VCSEL. The most significant bit controls the 16th VCSEL and the least significant bit controls the 1st VCSEL. For example the hexadecimal output word 0x1248 will send the following command to the VCSELs:

	1	2	3	4	5	6	7	8
1	0	0	0	1	0	0	1	0
2	0	1	0	0	1	0	0	0

A VCSEL that receives a "1" is switched on; a VCSEL that receives a "0" is switched off. The selection words represent the 16 possible outputs with only one VCSEL on.

Besides, the values from the detectors are always read in the same order for each DSP. This order gives the mapping of the artificial neurons in the DSP memory. This order is as follows:

	1	2	3	4	5	6	7	8
1	2	4	6	8	10	12	14	16
2	1	3	5	7	9	11	13	15

This gives the order in which the selection words are defined. Here is the actual initialisation of the selection words:

```
sel:    .word 00100h, 00001h, 00200h, 00002h
        .word 00400h, 00004h, 00800h, 00008h
        .word 01000h, 00010h, 02000h, 00020h
        .word 04000h, 00040h, 08000h, 00080h
```

5) Synchronisation:

The synchronisation takes place at the beginning of the neural network optimisation. It consists in synchronising the four DSPs, which are run one after the other, so that the output for each iteration takes place at the same moment to allow correct interconnection between the neurons. It can be done either using the optical system as an interconnection between neurons handled by different DSPs or using a hardware link between the DSPs. Both synchronisations are done so that the same code can be loaded into the four DSPs. A word called “syntyp” is reserved in the DSP memory: it is either equal to 0, which indicates that the program will carry out a hardware synchronisation, or 1, for an optical synchronisation.

a) Optical synchronisation:

Each DSP switches one VCSEL on and measures the optical signal received not on the corresponding detector, but on the detector just beneath it. Indeed there is an interconnection between aligned neurons in both cases of crossbar and Banyan DOEs. The VCSELs that are switched on are number 4, 20, 36 and 52. The detectors used for the synchronisation are number 12, 28, 44 and 60.

Because of the noise level, the input has to be averaged over 256 values in a rotating memory. As soon as the averaged input for each DSP reaches a predefined threshold (called “synth”), it means that all four VCSELs are on and that the program can continue and start the neural network optimisation.

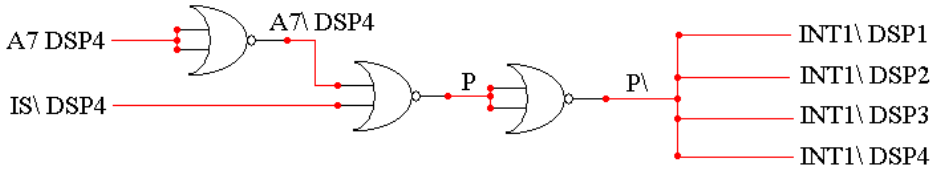
The problem with the optical synchronisation is that because of the necessary input averaging, a delay is induced between the DSPs. The maximum value for this delay in order to allow efficient synchronisation (called “delay” in the DSP memory) has to be defined before starting the program. The outputs are synchronised in each iteration only if the program waits for a time equivalent to the maximum delay before and after changing the output.

When using the optical synchronisation, the time needed for each iteration is about **2.62 ms**. This is much longer than what is actually needed for the neural network calculation, so it doesn't allow fast decision-making. But on the other hand it was very interesting to prove that the optical system could provide all the interconnection needed between the neurons and the DSPs to do the full neural network optimisation.

b) Hardware synchronisation:

For the hardware synchronisation, the DSPs are linked electrically: one DSP (the last to be run, DSP4) sends a hardware interruption to the other three DSPs and to itself. For each DSP, as soon as the hardware interruption is received, the neural network calculation is branched and the neural network optimisation begins.

The interruptions occur on the pin INT1\ of each DSP. The interruption takes place when a "0" is received on INT1\. The address line A7 of DSP4 combined with IS\ of DSP4 will be used to send the interruption signal, which has to be low when A7=1 and IS\=0 (meaning that an I/O port is used). To send the interruption signal, write a value equal to zero to I/O space address 0x0080. The following figure shows the required circuitry using NOR gates and the table shows the associated logic.



IS\ DSP4	A7 DSP4	A7\ DSP4	P	INT1\
0	0	1	0	1
0	1	0	1	0
1	0	1	0	1
1	1	0	0	1

The component used to create the interruption signal is of the following type: National MM74HC02 Quad 2-Input NOR Gates. This component has four NOR gates. Here are the colours of the wires that connect that component to the DSPs:

- Black for the four wires connected to the four INT1\ pins.
- Red for the wire connected to the A7 pin of DSP4.
- Blue for the wire connected to the IS\ pin of DSP4.

The interruption can only happen if bit 9 of ST0 is cleared and if bit 0 of IMR is set. Bit 9 of ST0 is called INTM or interruption mask, IMR is the interruption mask register and bit 0 of IMR corresponds to a mask for interruptions on INT1\.

An interruption on INT1\ then branches a fixed memory address in the DSK: 0x0802, called the interruption vector location for INT1\. So this memory location and the following have to be initialised with the code that indicates that the following part of the program should be branched. For example, if the program should branch the

program memory address 0x2020, the memory addresses 0x0802 and 0x0803 should be initialised with the values 0x7980 and 0x2020:

Memory address:	0x0802	0x0803
Mnemonic:	B	02020h
Corresponding code:	0x7980	0x2020

In the case of hardware synchronisation, the value of "delay" can be equal to 0 and the time needed for one iteration is **0.11 ms**. This time is a lot shorter than that using the optical synchronisation: in this case the iteration is about 24 times faster.

6) Neural network calculation:

Each one of the 64 neurons corresponds to a connection between an input and an output of a switch. Any connection may be requested, and this translates to request values being given to the neurons. The neurons are connected through the optical system. Each neuron's input is the optical signal received on its detector and its output is the optical signal sent by its VCSEL, which is either on or off.

During a neural calculation, the neuron first multiplies the input by –A: A is the weight and the minus sign shows that the input inhibits the neuron. It then adds a bias called B. This gives the value x:

$$x = -A*input + B$$

The neuron has a memory value called "mem" that is updated at each iteration and is equal to:

$$mem = previous\ value\ of\ mem + x$$

It then determines its response based on the activation function f(mem). Here the activation function is a sigmoid function:

$$f(mem) = \frac{1}{1 + e^{-\beta * mem}}$$

The neuron then multiplies f(mem) by the request, which can take any value between 0 and 256 with a precision equal to 2^{-8}. The output is 1 if the result is greater or equal to a threshold value (called "th", usually 0.5), and 0 otherwise:

$$output = 1\ if\ request*f(\text{mem}) \geq threshold$$
$$output = 0\ if\ request*f(\text{mem}) < threshold$$

The output is just one bit per neuron and each DSP handles 16 neurons. So the output is a word (16 bits) for each DSP.

The memory word for each neuron has a large range of integer values: from –32768 to 32767. The sigmoid function takes on 514 values, depending on the neuron memory value mem:

- $f(mem) = 0\ if\ mem < -1024$
- $f(mem) = \frac{1}{1 + e^{-\beta * mem}}\ if\ -1024 \leq mem < 1024$
- $f(mem) = 1\ if\ mem \geq 1024$

A, B and β are called the neural network parameters. The neural calculation takes place a predefined number of times. This number is called "iter"; it represents the number of iterations that have to be carried out for the optimisation to get to a valid result. The result is composed of the final output words of the 4 DSPs.

Number of requests:
The number of requests that are downloaded to the DSPs is stored in the DSP memory word called "number". The following memory word contains the number of results that have currently been obtained. This allows checking whether the program has had time to solve all the requests.

Initialisation:
At the beginning of the program and between the optimisations of two different requests, the neuron memory matrix and the output word are initialised; all the bits are given the value 0 (label "next" in the DSP program"). The number of the current iteration is also initialised. Obviously these initialisations do not take place between two iterations of the same optimisation.
However it can be interesting not to do the initialisation at the beginning of the program in the case of a single request when the neural network should continue the previous optimisation. It is the purpose of the variable called "oneit". If "oneit" equals 1, the neuron memory matrix and the output word are not initialised. If it equals 0, the initialisation takes place normally. This allows the computer to run the iterations of an optimisation one by one, so it can record the evolution of the neurons' memory values.

7) Organisation of the DSP memory:

The user's space in the DSK memory starts at the address 0980h and finishes at the address 2BFFh. The program "nnetwork.asm" is divided into three main parts:

- The program that starts at the address 0A00h and for which a length of 0300h is reserved.
- The uninitialised data from address 0D00h to 20FFh (length 1400h). It is divided into four parts:
  - "sigm" from 0D00h to 0EFFh (length 0200h): the values of the sigmoid function.
  - "aver" from 0F00h to 0FFFh (length 0100h): the rotating memory that is used for averaging the input in the optical synchronisation.
  - "req" from 1000h to 1FFFh (length 1000h): the request matrices: up to 256 request matrices with 16 values.
  - "res" from 2000h to 20FFh (length 0100h): the result words: up to 256 results.
- The initialised data that starts at the address 2100h and for which a length of 0100h is reserved. It contains the list of constants and variables needed for the calibration, synchronisation and neural network calculation.

a) Sigmoid function values:

The sigmoid function values consist in a look-up table containing 512 values. The active range of neuron memory values, from –1024 to 1024, contains 2048 values. To find the corresponding sigmoid function value, the neuron memory value is first divided by 4, then the result found serves as an address to find the sigmoid function value. For example, if the memory value is equal to 0xfd0b (-757):

- The memory value is divided by 4: the result is 0xff43 (-189, only keep the integer part).
- 0x0e00 is added to the result to find the address: 0xff43 + 0x0e00 = 0x0d43.
- The corresponding sigmoid function is located at the address 0x0d43.

Of course the sigmoid function values, which depend on β, have to be initialised before the program is run.

b) Direct memory addresses:

The DSP program sometimes calls direct memory addresses that contain constants or variables in the initialised data. This only calls addresses starting with 21xxh if the data memory page pointer (DP) bits are correctly initialised. Indeed, these bits contain the first 9 bits of any direct memory address. For an address starting with 21xxh, the first 9 bits are: 0010 0001 0, which makes the binary value 1000010, which is equal to 66. The DP bits are thus initialised using the following instruction:

```
        LDP     #66
```

II) The optoelectronic neural network demonstrator program (NNetDem):

The windows that correspond to the program described here are shown in appendix 2.

1) The optoelectronic neural network demonstrator control module (NNetCtrl):

This module allows to start up and run the DSPs of the demonstrator. The neural network parameters are defined in it.
The module is composed of the following eleven files:
NNetCtrl.cpp, NNetCtrl.h, NNetCtrl.dfm, NNetVCSELs.cpp, NNetVCSELs.h, NNetVCSELs.dfm, DSKCtrl.cpp, DSKCtrl.h, DSKCtrl.dfm, Ports.cpp and Ports.h.

a) Software interface:

* Variables:

A, B, beta (β), it (number of iterations), th (threshold), synth (synchronisation threshold), syndelay (synchronisation delay):

```
double NNetCtrlForm->A
double NNetCtrlForm->B
double NNetCtrlForm->beta
unsigned int NNetCtrlForm->it
double NNetCtrlForm->th
unsigned int NNetCtrlForm->synth
unsigned int NNetCtrlForm->syndelay
```

These variables are the neural network parameters and the parameters needed to run the neural network optimisation. They are described before, in the part "DSP program".

vcsel[8][8], vcselssix, vcselseight:

```
bool NNetCtrlForm->vcsel[8][8]
int NNetCtrlForm->vcselssix
int NNetCtrlForm->vcselseight
```

These are altered when using the "Broken VCSELs" button and the "VCSELs" form. The matrix of booleans `vcsel[8][8]` represents each one of the 64 VCSELs. It is `true` if the corresponding VCSEL works, `false` if it doesn't. The number of VCSELs that work is called `vcselseight`. The number of VCSELs that work among the central 6×6 array is called `vcselssix`.

active:

```
bool NNetCtrlForm->active
```

This Boolean describes the state of the "NNetCtrlForm" window. If `active` is `true`, the parameters shown in the window can be changed. If it is `false`, the parameters can only be read.

* Functions:

StartUpNetwork:

```
bool NNetCtrlForm->StartUpNetwork(bool speed)
```

This procedure starts up the demonstrator. It first initialises the four DSKs. It then loads the file "nnetwork.out" into the DSPs. The procedure then runs the part of the DSP program that switches the VCSELs off and it runs the calibration. It finally transmits `A`, `B`, `it`, `th`, `synth`, `syndelay` and the sigmoid function values based on `beta` to the four DSPs. The procedure returns `true` if no errors occur in the DSK Controller Module or `false` otherwise.
speed: Should a DSK calibration be performed or not. Can be `true` or `false`.

RunNetwork:

```
bool NNetCtrlForm->RunNetwork(double ***request, int reqnumber,
unsigned int ***result, int *resnumber, int synchronisation)
```

This procedure runs the optimisation of a certain number of requests. It first transmits the request matrices to the DSPs, then runs the DSPs and receives the result matrices. The procedure returns `true` if no errors occur in the DSK Controller Module or `false` otherwise.
`***request`: The request matrices.
`reqnumber`: The number of requests to be solved.
`***result`: The result matrices.
`*resnumber`: The number of results the program has had time to optimise.
`synchronisation`: The type of synchronisation used. Can be `OPTICS` or `HARDWARE`.

RunSingleRequestNetwork:

```
bool NNetCtrlForm->RunSingleRequestNetwork(double **request, unsigned
int **out, int **in, int **mem, int *resnumber, int synchronisation)
```

This procedure runs a certain number of iterations of the current optimisation, without resetting the values of the output matrix or the neurons' memory matrix. This can be done only if there is only one request to be solved. The procedure transmits `singleit=1` to indicate that a single request process is required. It then transmits the request matrices to the DSPs, runs them and receives the output word (converted into the output matrix), the input matrix and the neurons' memory matrix. The procedure returns `true` if no errors occur in the DSK Controller Module or `false` otherwise.
`**request`: The request matrix.
`**out`: The output matrix.
`**in`: The input matrix.
`**mem`: The neurons' memory matrix.
`*resnumber`: The number of results the program had time to optimise. Can be `0` or `1`.
`synchronisation`: The type of synchronisation used. Can be `OPTICS` or `HARDWARE`.

The next four procedures are very similar. They respectively transmit new values for `A`, `B`, `it` and the sigmoid function values based on `beta` to the four DSPs. They return `true` if no errors occur in the DSK Controller Module or `false` otherwise.

ChangeANetwork:

```
bool NNetCtrlForm->ChangeANetwork()
```

ChangeBNetwork:

```
bool NNetCtrlForm->ChangeBNetwork()
```

ChangeBetaNetwork:

```
bool NNetCtrlForm->ChangeBetaNetwork()
```

ChangeItNetwork:

```
bool NNetCtrlForm->ChangeItNetwork()
```

ResetIteration:

```
bool NNetCtrlForm->ResetIteration(unsigned int **out, int **mem)
```

This procedure transmits an output matrix (converted into an output word) and a neurons' memory matrix to the DSPs. This is used during single requests to reset an iteration when an error occurred in the previous iterations, which lead to invalid values in the neuron's memory matrix and in the output word. Indeed the neurons' memory matrix and the output word are the only evolving variables that are kept from one iteration to the next. The procedure returns `true` if no errors occur in the DSK Controller Module or `false` otherwise.
`**out`: The output matrix.
`**mem`: The neurons' memory matrix.

GetNoise:

```
bool NNetCtrlForm->GetNoise(unsigned int **noise, unsigned int
**oorder)
```

This procedure receives the values of the noise matrix and the zeroth order matrix from the DSPs. The procedure returns `true` if no errors occur in the DSK Controller Module or `false` otherwise.
`**noise`: The noise matrix.
`**oorder`: The zeroth order matrix.

b) User interface:

See figure 1 of appendix 2.

* Neural network parameters:

The group box called "Neural network parameters" allows changing the values of `A`, `B`, `beta`, `it`, `th`, `synth` and `syndelay`, if `active` is `true`.

* Neural network controls:

DSK Control:
When this button is clicked, the "DSKCtrl" form is displayed modally.

Broken VCSELs:
When this button is clicked, the "VCSELs" form is displayed modally (see the next paragraph, "VCSELs form").

Switch VCSELs Off:

When this button is clicked, the VCSELs are switched off. The computer first initialises the four DSKs, and then loads the file "nnetwork.out" into the DSPs. It then runs the part of the DSP program that switches the VCSELs off. If the "High speed" check box is checked, a DSK calibration is performed or a pre-calibrated value is used.

* VCSELs form:

See figure 2 of appendix 2. The neurons that correspond to VCSELs that do not work can be deselected using the "VCSELs" form. This alters the values of `vcsel[8][8]`, `vicselssix` and `vcselseight`.

2) The optoelectronic neural network demonstrator main program (NNetDem):

This program allows the user to:

- Run a predefined number of full optimisations on the demonstrator, using randomly generated request matrices.
- Read the result matrices obtained and have some information about the results in term of validity, optimality…
- Run a number of tests on the different parameters of the demonstrator and get tables that can be used to plot the evolution of the validity, optimality…

All the request matrices generated in the main program only use the request values 0 and 1, which means that the neurons are either requested or not.

a) Variables and functions:

size, type, synchronisation:

```
int Main->type
int Main->size
int Main->synchronisation
```

These variables specify the type of interconnection, the size of the neuron array, and the type of synchronisation.

`type` can be either `CROSSBAR` or `BANYAN`.

`size` can be either `8` (for the 8×8 neuron array) or `6` (for the 6×6 central neuron array). If `type` is `BANYAN`, then `size` can only be `8`.

`synchronisation` can be either `OPTICS` or `HARDWARE`.

load:

```
int Main->load
```

This variable specifies the load: the number of neurons that will be requested in the request matrices. It can take any integer value between `0` and the number of VCSELs that work (either `NNetCtrlForm->vicselseight` or `NNetCtrlForm->vicselssix`).

requestnumber, resultnumber:

```
int Main->requestnumber
int Main->resultnumber
```

These variables respectively specify the number of request matrices that will be sent to the neural network and the number of result matrices that have been received from the neural network. They can take any integer value between `0` and `256`.

*****request, ***result:**

```
double Main->***request;
unsigned int Main->***result;
```

These two matrices are three-dimensional and represent respectively the request matrices and the result matrices. The first index (k in `request[k][i][j]` or `result[k][i][j] respectively`) indicates the index of the matrix and varies from `0` to `requestnumber-1` or `resultnumber-1 respectively`.
`request[k]` is an 8×8 matrix of request values, which are real numbers.
`result[k]` is an 8×8 result matrix, with values equal to `0` or `1`.

opt:

```
int Main->opt
```

This variable represents the optimum number of neurons that can be on, depending on the neuron array size and the load. If `load` is greater than `size`, then `opt` equals `size`. If `load` is lower than `size`, then `opt` equals `load`.

validnumber, optimumnumber, suboptimumnumber, shouldbeonnumber, neuronson, missingneurons:

```
int Main->validnumber
double Main->neuronson
int Main->optimumnumber
int Main->suboptimumnumber
int Main->shouldbeonnumber
double Main->missingneurons
```

These variables are used to evaluate the results that are given by the demonstrator.
`validnumber` is the number of results that are valid.
`neuronson` is the average number of neurons on over all the results received, the number of which is `resultnumber`.
`optimumnumber` is the number of results that have optimum results: a number of neurons on equal to `opt`. This variable is only used for the crossbar interconnection.
`suboptimumnumber` is the number of results that have results with a number of neurons on equal to `opt-1`. This variable is only used for the crossbar interconnection.
`shouldbeonnumber` is the number of results that have `opt -1` neurons on when they could have `opt` neurons on: one neuron did not turn on when it should have. This variable is only used for the crossbar interconnection.
`missingneurons` is the average number of neurons that did not turn on when they could have. This variable is only used for the banyan interconnection.

reqindex:

```
int Main->reqindex
```

This variable indicates the index of the request or result matrix that is currently displayed in the "Run the optoelectronic neural network" group box.

RandRequestFill:

```
void __fastcall Main->RandRequestFill()
```

This function randomly loads the request matrices: the request matrices are filled with values equal to `0` and `1`, the number of values equal to `1` corresponds to the value of `load`. The neurons with a request value equal to `1` are the ones that are requested. If `size` equals `6`, then no neuron is requested in the two outer rows and the two outer columns.

EvaluateTime:

```
bool __fastcall Main->EvaluateTime(int time)
```

Based on `time`, which corresponds to a number of seconds, this function evaluates the time in hours and minutes needed to run a test. It then displays a dialogue box hat indicates to the user the time needed. The user can then choose to run the test or not.

CheckValidBanyan:

```
bool __fastcall Main->CheckValidBanyan(int i, int j, int reqindex)
```

This function checks the validity of one value of a result matrix in the case of the banyan interconnection.
`i, j`: the two indices of the value to be checked within the result matrix.
`reqindex`: the index of the result matrix that is being checked for validity.

InterpretResult:

```
void __fastcall Main->InterpretResult()
```

This function interprets the result matrices that have been received from the demonstrator. The following steps are obviously done differently for the crossbar interconnection and the banyan interconnection.
The function first checks whether the results are valid or not and the number of valid results is stored in `validnumber`. It evaluates the average number of neurons on and stores the value in `neuronson`.
For a crossbar interconnection, it updates the values of `optimumnumber`, `suboptimumnumber` and `shouldbeonnumber`.
For a banyan interconnection, it updates the value of `missingneurons`.

InterpretOneResult:

```
void __fastcall Main->InterpretOneResult()
```

This function interprets the result matrix with the index equal to `reqindex`. It indicates whether the matrix is valid or not. If it is valid, it indicates the number of neurons on and the number of neurons that did not turn on when they could have. For the crossbar interconnection, the number of missing neurons can only be 1 and only when the number of neurons on is equal to `opt-1`.

b) User interface:

See figure 3 of appendix 2.

* Main controls:

DOE type, Array size, Synchronisation:

The type of interconnection (crossbar or banyan), the size of the neuron array (6×6 or 8×8, only for the crossbar interconnection), and the type of synchronisation (optical or hardware) can be chosen using these three radio boxes.

Load:
This specifies the load; it alters the value of `load`. The default value is 50 %: 32 neurons out of 64 for the 8×8 neuron array, 18 out of 36 for the 6×6 neuron array.
If the load is equal to 2, a check box with the caption "Same row or column" appears. The user can then choose to have the two requested neurons in the same row or the same column in all the request matrices.

Number of requests:
This is the number of request matrices that will be sent to the demonstrator. The default value is 100 and the maximum value is 256.

Reset:
When this button is clicked, all the variables that are used to evaluate the results are erased. The results displayed in the group box called "Run the optoelectronic neural network" are also erased. Any open disk file is closed. The buttons used to run and test the demonstrator are enabled.

Neural Network Control:
When this button is clicked, the "NNetCtrl" form is displayed modally (see the paragraph called "The optoelectronic neural network demonstrator control module").

Custom Request:
When this button is clicked, the "CustReq" form is displayed modally (see the paragraph called "The custom request window").

* Run the optoelectronic neural network:

Run Random:
When this button is clicked:

- First the time needed to run the system is evaluated and the function `EvaluateTime` is called.
- Then the function `NNetCtrlForm->StartUpNetwork` is called to start up the demonstrator.
- Depending on the value of `reqnumber`, memory is allocated to the matrices `***request` and `***result`.
- The function `RandRequestFill` is called to randomly create the request matrices, depending on `load`.
- The function `NNetCtrlForm->RunNetwork` is called to run the demonstrator with the request matrices randomly created and write the results in `result[k][i][j]`.
- The results received are interpreted using the function `InterpretResult`.
- The interpretation is displayed in the memo with the label "Results".

If an error occurs while controlling the DSPs, an error dialogue box appears, the function is stopped and the "Reset" button is enabled.

Run Triangle:
Clicking this button almost does the same thing as clicking "Run Random". The only difference is that all the request matrices are the same: the neurons requested are those that are in the lower part of the array, under the diagonal. This is a representation of this matrix:

	1	2	3	4	5	6	7	8
1	1	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0
3	1	1	1	0	0	0	0	0
4	1	1	1	1	0	0	0	0
5	1	1	1	1	1	0	0	0
6	1	1	1	1	1	1	0	0
7	1	1	1	1	1	1	1	0
8	1	1	1	1	1	1	1	1

The triangular request matrix is very useful to test the system because it only has one optimum solution that is well known: a result matrix with all the neurons in the diagonal on.

Results:
In this memo is displayed the interpretation of the results that have been received when using "Run Triangle" or "Run Random".
For the crossbar interconnection, it indicates:

- The number of results received (`resultnumber`).
- The number of valid results (`validnumber`).
- The number of optimum results (`optimumnumber`).
- The number of results with `opt-1` neurons on (`suboptimumnumber`).
- The number of results with either `opt` neurons on or `opt-1` neurons on (`optimumnumber+suboptimumnumber`).
- The number of results with `opt-1` neurons on that could have `opt` neurons on (`shouldbeonnumber`).
- The average number of neurons on (`neuronson`).

For the banyan interconnection, it indicates:

- The number of results received (`resultnumber`).
- The number of valid results (`validnumber`).
- The average number of neurons on (`neuronson`).
- The average number of neurons that could be on and are not (`missingneurons`).

Request or result number:
This specifies the index of the request matrix or result matrix that will be displayed. It alters the value of `reqindex`.

See request:
When this button is clicked, the request matrix with the index equal to reqindex is displayed. The corresponding result is interpreted by the function

`InterpretOneResult`, and the evaluation of the result is displayed in the memo with the label "result".

See result:
When this button is clicked, the result matrix with the index equal to reqindex is displayed. The result matrix is interpreted by the function `InterpretOneResult`, and the evaluation of the result is displayed in the memo with the label "result".

Result:
The interpretation of the result that is currently displayed or which corresponds to the request that is currently displayed (given by `InterpretOneResult`) is indicated int this memo.

* Test the optoelectronic neural network:

A Test, B Test, Beta Test, It Test, Load Test, Stability Test:
Clicking these buttons almost does the same thing. The difference is the parameter on which the test is carried out. For example when the "A test" button is clicked:

- First the "Test starting point", the "Test finishing point" and the "Test step" are read in the three edit boxes.
- Then the time needed for the test is evaluated and the function `EvaluateTime` is called.
- It opens a file to write the results. The name of the file is specified in the edit box with the label "Write results in file".
- Then the function `NNetCtrlForm->StartUpNetwork` is called to start up the demonstrator.
- Depending on the value of `reqnumber`, memory is allocated to the matrices `***request` and `***result`.
- The loop begins at this point. The new value of A is sent to the DSPs using the function `NNetCtrlForm->ChangeANetwork`.
- The function `RandRequestFill` is called to randomly create the request matrices, depending on `load`.
- The function `NNetCtrlForm->RunNetwork` is called to run the demonstrator with the request matrices randomly created and write the results in `result[k][i][j]`.
- The results received are interpreted using the function `InterpretResult`.
- The interpretation is written in the file.
- If the test finishing point has not been reached, the loop continues with the next value of A.
- The file is finally closed.

If an error occurs while controlling the DSPs, an error dialogue box appears, the function is stopped and the "Reset" button is enabled.

A and B Test:
Clicking this button does almost the same thing as clicking the other test buttons. The only difference is that both A and B vary in this test, according to the values in the six edit boxes: "A starting point", "A finishing point", "A step", "B starting point", "B finishing point", "B step".

Write results in file:
This specifies the name of the file in which the interpretation of the test results is written. The file first indicates the type of interconnection. If the type is crossbar, it indicates the size of the neuron array.
For the crossbar interconnection, the file gives the following information every time the demonstrator is run (for each one of the test values):

- The values of `NNetCtrlForm->A`, `NNetCtrlForm->B`, `NNetCtrlForm->beta`, `NNetCtrlForm->it`, `requestnumber`, `load`, `resultnumber`.
- The values of `resultnumber` (called "result"), `validnumber` (called "valid"), `optimumnumber` (called "8 ON" for example if `opt` equals 8), `suboptimumnumber` (called "7 ON"), `optimumnumber+suboptimumnumber` (called "7 or 8 ON"), `shouldbeonnumber` (called "7 not 8"), `neuronson` (called "neurons ON").

For the banyan interconnection, the file gives the following information every time the demonstrator is run (for each one of the test values):

- The values of `NNetCtrlForm->A`, `NNetCtrlForm->B`, `NNetCtrlForm->beta`, `NNetCtrlForm->it`, `requestnumber`, `load`, `resultnumber`.
- The values of `resultnumber` (called "result"), `validnumber` (called "valid"), `neuronson` (called "neurons ON"), `missingneurons` (called "missing").

3) The custom request window (NNetCustReq):

This window allows the user to:

- Write his own request matrix with any real request values.
- Run a certain number of iterations of the current optimisation.
- Read the different matrices from the DSP memories.

Only one request matrix is sent to the DSPs at a time here.

a) Variables and functions:

size, type, synchronisation, load:
These variables are redefined here but play the same role as in the main program.

value:

```
double CustReq->value
```

This variable indicates the request value: the value that will be used in the request matrix. Its default value is `1.0`.

****request, **evol, **out, **noise, **oorder, **in, **mem:**

```
double CustReq->**request
unsigned int CustReq->**out
int CustReq->**in
int CustReq->**mem
unsigned int CustReq->**noise
unsigned int CustReq->**oorder
double CustReq->**evol
```

These are all two-dimensional matrices that can be displayed in the state grid.
`**request` is the request matrix that will be sent to the DSPs.

`**out`, `**in`, `**mem`, `**noise`, `**oorder` are respectively the output matrix, the input matrix, the neurons' memory matrix, the noise matrix and the zeroth order matrix that will be read from the DSPs.
`**evol` is the evolution matrix which represents f(mem) (see chapter I called "DSP program").

display:

```
int CustReq->display
```

This variable indicates the matrix that is currently displayed. It can be `REQUEST_PLANE`, `OUT_PLANE`, `IN_PLANE`, `MEM_PLANE`, `EVOL_PLANE`, `NOISE_PLANE`, `OORDER_PLANE` or `INNOISE_PLANE`. The default value is `REQUEST_PLANE`. `INNOISE_PLANE` corresponds to a matrix that is the sum of `**in` and `**noise`.

iterations:

```
int CustReq->iterations
```

This variable indicates the number of iterations that have been run so far in the current optimisation.

resnumber:

```
int CustReq->resnumber
```

This variable indicates the number of results that have been received from the DSPs. If the demonstrator works as expected, its value is 1, as only one matrix request is sent when using the custom request window.

valid:

```
bool CustReq->valid
```

This Boolean indicates whether the current output matrix is a valid result or not.

activeneurons:

```
int CustReq->activeneurons
```

This variable indicates the current number of neurons on.

InterpretResult:

```
void __fastcall CustReq->InterpretResult()
```

This function interprets the current output matrix: it updates the values of `valid` and `activeneurons`.

UpdateDisplay:

```
void __fastcall CustReq ->UpdateDisplay()
```

This function updates the matrix that is displayed, depending on the value of `display`.

b) User interface:

See figure 4 of appendix 2.

* Main controls:

DOE type, Array size, Synchronisation:

The type of interconnection (crossbar or banyan), the size of the neuron array (6×6 or 8×8, only for the crossbar interconnection), and the type of synchronisation (optical or hardware) can be chosen using these three radio boxes.

Request value:
This specifies the request value. It alters the value of `value`. To change a value of the request matrix, double-click the corresponding cell when the request matrix is displayed. This writes the request value in the cell. If the value in the cell is not 0, double-clicking writes a 0 in the cell.

Load:
This specifies the load: the number of neurons that will be requested in the request matrices. It alters the value of `load`. The default value is 50 %: 32 neurons out of 64 for the 8×8 neuron array, 18 out of 36 for the 6×6 neuron array.

Number of iterations to run:
This specifies the number of iterations that should be performed by the neural network. It alters the value of `NNetCtrlForm->it`.

Random load:
Clicking this button randomly loads the request matrix with values equal to 1. The number of neurons requested is equal to `load`. If `size` equals `6`, then no neuron is requested in the two outer rows and the two outer columns.

Neural Network Control:
When this button is clicked, the "NNetCtrl" form is displayed modally (see the paragraph called "The optoelectronic neural network demonstrator control module").

Start Up:
When this button is clicked, the function `NNetCtrlForm->StartUpNetwork` is called to start up the demonstrator. The function `NNetCtrlForm->GetNoise` is then called to get the values of the noise matrix and the zeroth order matrix that have been measured during the calibration. Finally the "Neuron Evolution" button and the "Run" button are enabled.

Reset:
The values of `iterations`, `activeneurons`, `resnumber`, `**request`, `**evol`, `**out`, `**noise`, `**oorder`, `**in` and `**mem` are reset. The request matrix is selected. Any open disk file is closed. The "Start Up" button is enabled; the "Neuron Evolution" button and the "Run" button are disabled.

* Run the optoelectronic neural network:

When the "Run" button is clicked, a certain number of iterations (equal to `NNetCtrlForm->it`) of the current optimisation are run, using the current request matrix. These are the steps followed:

- First the value of `iterations` is updated by adding `NNetCtrlForm->it` to the current value of `iterations`.

- Then the function `NNetCtrlForm->ChangeItNetwork` is called to send the new value of `NNetCtrlForm->it` to the DSPs.
- The function `NNetCtrlForm->RunSingleRequestNetwork` is called to send the request matrix, run the demonstrator and update the other matrices.
- If the number of results received is different from 1, it means that there has been a problem while running the demonstrator, for example the synchronisation did not work as expected. In this case the previous output matrix and the previous neurons' memory matrix are sent back to the DSPs using the function `NNetCtrlForm->ResetIteration`. Then the demonstrator is run again using the function `NNetCtrlForm->RunSingleRequestNetwork` until the number of results received is equal to 1.
- The function `InterpretResult` is called to interpret the output matrix received.
- The interpretation of the result is written in the memo with the label "Result". It indicates whether the matrix is valid or not (`valid`). If it is valid, it indicates the number of neurons on (`activeneurons`) and the number of iterations run so far (`iterations`).
- The function `UpdateDisplay` is called to update the matrix that is currently displayed.

If an error occurs while controlling the DSPs, an error dialogue box appears, the function is stopped and the "Reset" button is enabled.

* Select the matrix to be displayed:

The buttons in this group box are used to choose the matrix that is displayed. They alter the value of `display` and call the function `UpdateDisplay`.

* Neuron evolution:

This group box is used to have an idea of the evolution of the neurons during an optimisation. The iterations are run one by one and the evolution matrix is written in a file at the end of each iteration. These are the steps followed:

- First a file is opened with the name specified in the edit box with the label "Write neuron evolution to file".
- Then `NNetCtrlForm->it is given the value 1.` The function `NNetCtrlForm->ChangeItNetwork` is called to send the new value of `NNetCtrlForm->it` to the DSPs.
- The demonstrator is run as when clicking "Run". This is done a number of times that corresponds to the value in the edit box with the label "Number of iterations to run". Each time, the evolution matrix is written in the file as a single line to give a table of the evolution of the 64 neurons.
- The function `InterpretResult` is called and the interpretation of the result is written in the file.
- The file is closed and `NNetCtrlForm->it is given its original value back.`

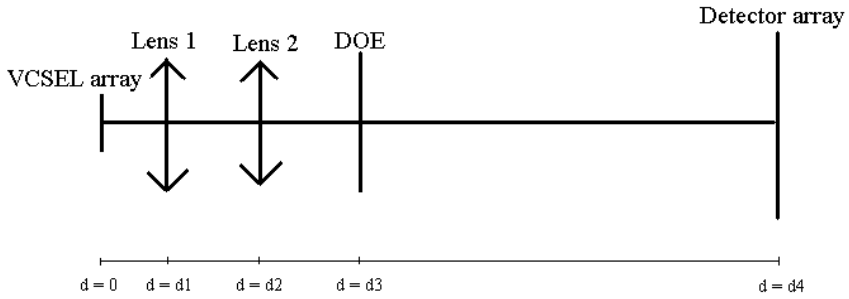
III) Results and discussion:

1) Optical system:

a) Description:

The optical system is composed of the following elements, as shown in the following figure (not to scale):

- The VCSEL array, placed at d = 0.
- Lens 1, focal length f_1 = 150 mm, placed at d = d_1.
- Lens 2, focal length f_2 = 24 mm, placed at d = d_2.
- The DOE (diffractive optic element), placed at d = d_3.
- The detector array, placed at d = d_4 = 180 mm.



The magnification γ between the VCSELs and the detectors has to be γ = 6.
Just for memory, here are the values of d_4 and γ as functions of f_1, f_2, d_1 and d_2:

$$d_4 = \frac{d_2^2(d_1 - f_1) - d_1^2(d_2 + f_2)}{f_1 f_2 + d_1 d_2 - d_1 f_2 - d_2 f_1 - d_1^2}$$

$$\gamma = \frac{f_1 f_2}{f_1 f_2 + d_1 d_2 - d_1 f_2 - d_2 f_1 - d_1^2}$$

The two conditions d_4 = 180 mm and γ = 6 are respected for the following values:

d_1 = 15.5 mm
d_2 = 26.675 mm

b) Zeroth order:

The zeroth order is the optical signal that is not diffracted by the DOE. The zeroth order of the DOEs is not negligible, so some of the signal sent by a VCSEL gets to the detector of the same neuron, which acts as an auto-inhibition. And since the neurons that are on are not supposed to inhibit themselves, the zeroth order has to be compensated for.

This is done during the calibration, by measuring the signal received by each detector when the corresponding VCSEL is on. The value measured is then subtracted from every neuron's input when its output is equal to 1.

c) Signal and noise levels:

When no VCSEL is on, the signal is, as read by the DSPs:

$$signal = 0 \pm 6$$

For the crossbar DOE, the signal received when one VCSEL (in the same row or same column) is on is, as read by the DSPs:

$$signal = 16 \pm 4$$

For the banyan DOE, the signal received when one VCSEL (in the same row or same column) is on is, as read by the DSPs:

$$signal = 9 \pm 4$$

For both DOEs, when only one VCSEL is on, the signal received on a detector that is in an adjacent row or column is, as read by the DSPs:

$$signal = 4 \pm 4$$

This gives a measure of the cross talk from one row (or column) to the next.

d) Problems:

The problems with the optical system as it is now are the following:

* Weak optical signals:

From the optical power that is emitted by a VCSEL, only a small part reaches the detectors that are aimed at. In theory, the DOEs have an efficiency of about 50 % (1/2) and the DOE divides the power into a given number of spots.
For the crossbar DOE, the number of spots is 28, so the original optical power is divided by $2 * 28 = 56$.
For the banyan DOE, the number of spots is 48, so the original optical power is divided by $2 * 48 = 96$.
This explains why about 40 % of the signal is lost when changing the DOE from crossbar to banyan. This induces a worse signal to noise ratio.

What could be done to improve the signal to noise ratio is increase the VCSEL output power, use DOEs with better efficiencies or decrease the noise. But the VCSELs used can't be driven with a higher current, as it would probably damage them quickly. Besides the optical system is so small that the working distances of the DOEs and the DOE gratings are very small, which doesn't allow much greater values for the efficiency. Not much can be done about the noise either, as it mainly comes from the electronics.

* Cross talk:

As can be seen from the signal and noise levels, when only one VCSEL is on, some signal is received on the detectors that are in adjacent rows or columns. This is what is called cross talk: the neurons inhibit other neurons that they should not inhibit. This is a problem when 3 or 4 neurons are on, as the signal received by some neurons that should not be inhibited is equivalent to the signal received by neurons that should be

inhibited. This means that some neurons do not turn on when they could, which leads to sub-optimal solutions.

* Broken VCSELs:

Two VCSELs do not work. These are the VCSELs number 27 and 48. In the test programs, they're never requested, in case they could induce invalid results. Because of this the load cannot exceed 96.875 % (62 out of 64) for the 8×8 neuron array. It cannot exceed 97.22 % (35 out of 36) for the central 6×6 neuron array.

2) Crossbar switch results:

The figures described in this part refer to appendix 3. All the results were obtained with the following parameters:

$$\beta = 0.02$$
$$number\ of\ iterations = 300$$
$$threshold = 0.5$$

The first two parameters that must be optimised in order to have good results are the weight A and the bias B. Figure 1 shows the evolution of the validity when A and B vary, with a load equal to 50 %. We can see that the validity can be equal to about 100 %, but that it drops for low values of A or big values of B. Figure 2 shows the evolution of the average number of neurons on. We can see that the number of neurons on decreases when A increases or when B decreases, the maximum number of neurons on being 8. The best results correspond to a validity of about 100 % and a maximum of neurons on. This is obtained for:

$$A = 1.05$$
$$B = 16$$

The results are about the same when A increases or when B decreases. Figure 3 shows the evolution of the results as a function of B and with A = 1.05. This figure shows that the validity drops when B is too high. It shows that the optimality increases with B, and that it naturally decreases with the validity. The best result is for B = 16, the optimality has its highest value with a good validity. When B<16, some results only have 7 neurons on when they could have 8. This is due to cross talk, which induces sub-optimal results. But this can be compensated for with a higher value for the bias.

With A = 1.05 and B = 16, for the 8×8 neuron array and with a load equal to 50 %, we got on average:

$$validity = 99.9\ \%$$
$$average\ number\ of\ neurons\ on = 7.67$$
$$optimality = 67\ \%$$
$$number\ of\ results\ with\ 7\ neurons\ on\ instead\ of\ 8 = 1.5\ \%$$

The most interesting results are those given when the load varies from 0 % to 100 %. Figure 4 shows the evolution of the results as a function of the load with A = 1.05 and B = 16. We can see that the optimality increases with the load and that the results always have 7 or 8 neurons on when the load is greater than 40 %. The validity keeps

a value greater than 95 % when the load is greater than 15 %. The number of results with 7 neurons on instead of 8 always stays below than 10 %.

Figure 5 shows the average number of neurons on as a function of the load. The shape of the graph corresponds to the results that Dr Keith Symington got with the software simulation of the optoelectronic neural network.

Although quite satisfying, the results shown on figures 4 and 5 are too instable and there are too many invalid results. We realised that this was due to unexpected input values in the first column of the neuron array (see "remarkable results" a bit further on). So we did the same experiment using only an array of 6×6 neurons, the central ones. The next results for the crossbar DOE all correspond to the central 6×6 neuron array. Figure 6 thus shows the evolution of the results as a function of the load. In this case the validity remains equal to 100 % when the load is greater than 25 % and the number of results with 7 neurons on when they could have 8 remains smaller than 1 %.

With A = 1.05 and B = 16, for the central 6×6 neuron array and with a load equal to 50 %, we got on average:

validity = 100 %
average number of neurons on = 5.65
optimality = 65.4 %
number of results with 7 neurons on instead of 8 = 0 %

The same results were obtained with a smaller number of iterations. For the 6×6 neuron array, 100 iterations were enough to get these results, whereas 300 iterations were always needed for the 8×8 neuron array.

Although the last figure is very satisfying, there is a problem: the validity drops for small loads (inferior to 15 %). The problem is that for small loads, the value of the bias B should be smaller to get good results. For example with a small load, when only two neurons are requested and they are in the same row or column, a value of B that is too high doesn't allow to solve the problem: one neuron alone will not be enough to inhibit the other, so both will turn on and the result will be invalid. Figure 7 shows the evolution of the results when only two neurons are requested and they are in the same row or column. The figure shows that the problem is solved every time with B<12.5.

In order to get good values of the validity even for small loads, the experiment shown in figure 6 is done again with B<12.5. Figure 8 thus shows the evolution of the results as a function of the load with B = 12. In this case the validity remains equal to 100 % whatever the load. The number of results with 7 or 8 neurons on is the same as previously. What changes is the optimality, which is lower, and the number of results that should have 8 neurons on instead of 7, which increases with the load. Indeed a lower value of B induces this sort of problem, as seen before.

Figure 9 corresponds to the same experiment as figure 8. It shows the average number of neurons on and the average number of neurons that could be on as a function of the

load. In this case the actual results are not optimal but all the results are valid, which is more important.

3) Banyan switch results:

The figures described in this part refer to appendix 4. All the following results were obtained with the following parameters:

$$\beta = 0.02$$
$$number\ of\ iterations = 300$$
$$threshold = 0.5$$

Figure 1 shows the evolution of the validity when A and B vary, with a load equal to 50 %. We can see that the validity can be equal to about 100 %, but that it drops for low values of A or big values of B. Figure 2 shows the evolution of the average number of neurons on. We can see that the number of neurons on decreases when A increases or when B decreases. The best results are obtained for:

$$A = 1.05$$
$$B = 9$$

The results are about the same when A increases or when B decreases. Figure 3 shows the evolution of the results as a function of B and with A = 1.05. This figure shows that the validity drops when B is too high. It shows that the number of neurons on increases with B. The best result is for B = 9, the number of neurons on is the highest with a good validity. The number of missing neurons, due to cross talk, decreases as B increases. Indeed a higher bias tends to force the neurons to switch on.

With A = 1.05, B = 9 and a load equal to 50 %, we got on average:

$$validity = 99.9\ \%$$
$$average\ number\ of\ neurons\ on = 5.11$$
$$average\ number\ of\ missing\ neurons = 1.57$$

Figure 4 shows the evolution of the results as a function of the number of iterations, with A = 1.05, B = 9 and load = 50 %. The results are very good for the region:

$$150 \leq number\ of\ iterations \leq 300$$

This means that 150 iterations are sufficient to obtain the expected results.

The most interesting results are those given when the load varies. Figure 5 shows the evolution of the results as a function of the load with A = 1.05 and B = 9. We can see that the validity keeps a value greater than 97 % when the load is greater than 15 %. The number of neurons on increases with the load, but never exceeds 5.4. The number of missing neurons also increases with the load, almost linearly. The number of neurons that should be on is the sum of the number of neurons on and the number of missing neurons. It is the curve that we would expect to get for the number of neurons on if the system were perfect. Although the results are valid they are not optimal, and that it because in the case of the banyan DOE the signal to noise ratio is too low and we can't compensate for this with a higher value of B without losing the validity.

There is another problem that appears in figure 5: the validity drops for small loads (inferior to 15 %). The problem is the same as that encountered with the crossbar interconnection; the value of B is too high. Figure 6 shows the evolution of the results as a function of B when only two neurons are requested and they are in the same row or column. The figure shows that the problem is solved every time with B≤6.

In order to get good values of the validity even for small loads, the experiment shown in figure 5 is done again with B≤6. Figure 7 thus shows the evolution of the results as a function of the load with B = 6. In this case the validity remains equal to 100 % whatever the load. The problem is that there are even less neurons on, and the average number of neurons on cannot exceed 3.8. Indeed a lower value of B induces this sort of problem, as seen before. The signal to noise ratio would have to be improved and the cross talk reduced to be able to get better results, i.e. a curve for the number of neurons on that approaches the theoretical one and goes up to 8 when the load is equal to 100 %.

4) Discussion: the neural network parameters

First of all, all the results I got for the demonstrator correspond to request matrices with only the values 0 and 1. This means that the neurons were either requested or not. In real situations, it could be interesting to test the system with different request values, which has not been done here. But it could be done by slightly altering the C++ program. The DSP program and the optoelectronic neural network demonstrator control module allow different real request values as they are now.

The most important results to test the optoelectronic neural network demonstrator for scheduling are the validity and the optimality. The optimality corresponds to having as many neurons on as possible, as it means that the throughput of the switch will be as high as possible. But the most important is that the validity should always be 100 %, even if it means that the results should not be optimal. Indeed invalid results are very difficult to deal with. A switch will not allow invalid interconnection, so another device would be needed to get rid of the invalid results. But that can't be done, as it would mean losing the speed of scheduling made possible by the neural network. So all the results really have to be valid.

The neural network parameters A and B play a very important role in the optimisation. Their values are critical to get valid and (or) optimal results. As has been said before, the roles of A and B are very similar. An increase in A is equivalent to a decrease in B. This is because of the definition of the neural network calculation formula for the neuron's memory value "mem":

$$mem = previous\ value\ of\ mem - A*input + B$$

An increase in the weight A gives more importance to the input, which decreases "mem" and so inhibits the neurons. This is the same as a decrease in the bias B, which gives more importance to the input and the inhibition. Let i_1 be the value of the input of a neuron when just one other neuron communicates with it. To have good results, A and B must be such as:

$$A * i_1 \geq B$$

The best results are obtained for:

$$A * i_1 \approx B$$

This leaves an infinite number of solutions with A/B constant. In the following paragraphs the effects of a change in B are explained. The same would be true with a change in A.

If the value of B is too high, there are invalid results because not enough importance is given to the neurons' inhibitory inputs. On the other hand, if B is too low, "mem" can never increase because the noise in the input is enough to inhibit the neurons. So the neurons never switch on and the results are not optimal. A compromise between validity and optimality has to be done to find the best value of B. If the signal to noise ratio was improved, it would mean that the results would be optimal even for smaller values of B. It would increase the range of values of B for which the results have both satisfying validity and optimality (the stability range). Indeed we can see that the stability range is larger in the case of the crossbar interconnection, where the signal to noise ratio is about 4, than in the case of the banyan interconnection, where the signal to noise ratio is about 2 (see figure 3 of appendix 3 and figure 3 of appendix 4). The interesting thing to notice is that we still managed to get results with a low signal to noise ratio, which is very encouraging for the promotion of optoelectronic neural networks.

More than the noise, it is the cross talk that represents the worst problem with the system as it is now, because it leads to sub-optimal solutions. But what is interesting is to see that the effect of the cross talk could be partially compensated for by increasing the value of B within the validity range (see figure 3 of appendix 3).

As far as the experiments I carried out are concerned, a change in β has no influence on the results. But the value of beta would probably play a role if the request values were different from just 0 or 1.

The number of iterations needed for the optimisation to get to a stable solution is smaller than that foreseen with the simulation. This is because the value of B is relatively bigger than that found with the simulation. Indeed a high value of the bias B leads to a stable solution with less iterations. The danger of allowing too few iterations is the possibility of having invalid results.

5) Neuron evolution during optimisation:

Figure 1 of appendix 5 shows the evolution of the 64 neurons as a function of the number of iterations during one optimisation. What is represented here is the evolution of f(mem) for each neuron (see the first part, "DSP program"). The output is defined as:

$$output = 1 \text{ if } f(mem) \geq 0.5$$
$$output = 0 \text{ if } f(mem) < 0.5$$

We can see that at the end 8 neurons are on and the others are off, a stable solution has been reached after about 150 iterations. The request and result matrices are enclosed as tables 1 and 2 of appendix 5.

6) Remarkable results:

a) Input averaging:

During optimisation, the neuron's input only consists in one measure. But although the input is very noisy, the results are not at all changed when the neuron input is averaged over 16 or 256 values during optimisation. This can be explained by saying that the important number of iterations done during the optimisation induces a sort of natural averaging of the input.

b) Lights on:

The neural optimisation was usually done in the dark, but it appears that the results are not different when the lights are on, so with more ambient light. What is important is that the ambient light conditions remain the same during calibration, when the ambient light will be recorded as noise, and during optimisation.

c) Unexpected inputs in the first channel of each ADC:

As was explained earlier, a majority of invalid results come from the fact that two neurons from the first column will be on at the same time, which is valid neither for the crossbar switch nor for the Banyan switch. This comes from the fact that the inputs from the detectors in the first column, so from the first channel of each ADC, sometimes have unexpected values. On average, one time out of two the input will be correct and the other time it will be unexpectedly low. So two neurons from the first column sometimes can't inhibit one another effectively.
I have not been able to find where the problem comes from, as the optical system doesn't seem to be cutting off one part of the optical signal in particular.

d) Broken VCSELs:

The neurons corresponding to broken VCSELs can be requested, as long as these neurons can be on together, so they don't lead to invalid results. Indeed these neurons cannot inhibit any other neurons but they can be inhibited. They'll never win over other requested neurons but they will switch on if no neuron inhibits them. In any case, they'll never lead to invalid results.

Conclusion:

Although very satisfying, the results that are exposed here could be improved by working on the signal to noise ratio in the system and by lowering the cross talk. But it was interesting to show that these problems could be partially compensated for by carefully choosing the values of the different neural network parameters.

The best speed that we could get from the system was 0.11 ms for one iteration using the hardware synchronisation and about 150 iterations to make a decision. This means that as it is now the system needs about 16.5 ms for each optimisation, so that it can make about 60 decisions per second.

Theses results are very important in the sense that they prove that good results can be obtained using an optoelectronic neural network for scheduling. To our knowledge, it was the first time anyone got such results. It is very promising, as it leads the way to new generations of optoelectronic neural networks used for very fast and efficient scheduling of packet switching.

Appendix 1: The DSP program (nnetwork.asm):

```
************************************************************************
*             Optoelectronic Neural Network Demonstrator               *
*                     DSP program: nnetwork.asm                        *
************************************************************************

************************************************************************
* Initialisation of data.                                              *
************************************************************************

; Include memory-mapped registers.
        .mmregs

; Reserve DSP memory space (uninitialised data).
; Starting at adress 0d00h.
        .bss  sigm, 512     ; Sigmoid function values.
        .bss  aver, 256     ; Array used for averaging input values.
        .bss  req, 4096     ; Request matrices (up to 256).
        .bss  res, 256      ; Result words (up to 256).

; Initialised data. Starting at adress 2100h.
        .data
val:    .word 00000h, 00000h  ; Values of A and B.
th:     .word 00000h          ; Value of the threshold.
iter:   .word 00000h, 00000h  ; Number of iterations for each request.
synth:  .word 00000h          ; Synchronisation threshold.
delay:  .word 00000h          ; Delay in the synchronisation.
number: .word 00000h, 00000h  ; Number of requests and results.
oneit:  .word 00000h          ; Indicates single or multiple requests.
out:    .word 00000h          ; Output word.
syntyp: .word 00000h          ; Type of synchronisation used.
in:     .word 00000h, 00000h, 00000h, 00000h  ;Input matrix.
        .word 00000h, 00000h, 00000h, 00000h
        .word 00000h, 00000h, 00000h, 00000h
        .word 00000h, 00000h, 00000h, 00000h
mem:    .word 00000h, 00000h, 00000h, 00000h  ; Memory matrix.
        .word 00000h, 00000h, 00000h, 00000h
        .word 00000h, 00000h, 00000h, 00000h
        .word 00000h, 00000h, 00000h, 00000h
sel:    .word 00100h, 00001h, 00200h, 00002h  ; Selection words.
        .word 00400h, 00004h, 00800h, 00008h
        .word 01000h, 00010h, 02000h, 00020h
        .word 04000h, 00040h, 08000h, 00080h
store:  .word 00000h          ; Word used to store values.
reqloc: .word 00000h          ; Location of the current request value.
resloc: .word 00000h          ; Location of the current result value.
itnb:   .word 00100h, 00000h  ; Gives 256 iterations for averaging.
syn:    .word 00008h          ; VCSEL used for the synchronisation.
temp:   .word 00000h, 00000h  ; To store input values temporarily.
av:     .word 00000h, 00000h  ; To store values for averaging.
noise:  .word 00000h, 00000h, 00000h, 00000h  ; Noise matrix.
        .word 00000h, 00000h, 00000h, 00000h
        .word 00000h, 00000h, 00000h, 00000h
        .word 00000h, 00000h, 00000h, 00000h
oorder: .word 00000h, 00000h, 00000h, 00000h  ; Zeroth order matrix.
        .word 00000h, 00000h, 00000h, 00000h
        .word 00000h, 00000h, 00000h, 00000h
        .word 00000h, 00000h, 00000h, 00000h
zero:   .word 00000h          ; Value equal to 0.
one:    .word 01000h          ; Value equal to 1 (Q12).
three:  .word 00003h, 00000h  ; Value equal to 3 (and storing word).
four:   .word 00004h          ; Value equal to 4.
sixtn:  .word 00010h          ; Value equal to 16.

; Program. Starting at adress 0a00h.
         .text

************************************************************************
* Input subtroutines for each channel.                                 *
************************************************************************

; Subroutines used to select an ADC, store an input value and start an
```

```
; analogue to digital conversion.
ch1:    IN      *+, 00200h, AR0   ; Store an input and select an ADC.
        IN      *-, 00100h, AR0   ; Start the conversion.
        RET                       ; Return from sub-routine.
ch2:    IN      *+, 02600h, AR0
        IN      *-, 00100h, AR0
        RET
ch3:    IN      *+, 04a00h, AR0
        IN      *-, 00100h, AR0
        RET
ch4:    IN      *+, 06e00h, AR0
        IN      *-, 00100h, AR0
        RET
ch5:    IN      *+, 09200h, AR0
        IN      *-, 00100h, AR0
        RET
ch6:    IN      *+, 0b600h, AR0
        IN      *-, 00100h, AR0
        RET
ch7:    IN      *+, 0da00h, AR0
        IN      *-, 00100h, AR0
        RET
ch8:    IN      *+, 0fe00h, AR0
        IN      *-, 00100h, AR0
        RET



*                   ****************************
*                   *     Switch VCSELs off.    *
*                   * Starting at adress 0a28h. *
*                   ****************************

*********************************************************************
* DSK board initialisation.                                         *
* See TI document SPRA253.pdf for further information.              *
*********************************************************************

; Initialise TMS320C50 On-chip timer.
; Load PRD register for period of 100nsec TDDR=0
        SPLK    #01h, PRD
; Re-load and begin timer.
        SPLK    #20h, TCR

; Serial port initialisation.
; FSM=1, XRST and RRST=00
        SPLK    #08h, SPC
; FSM=1, XRST and RRST=11
        SPLK    #0c8h, SPC

; AIC Initialisation.
; Init 8000h-FFFFh as global memory.
        LACC    #80h
; Store to global memory alloc register.
        SACL    GREG
; Use AR0 to point to location FFFFh.
        LAR     AR0, #0FFFFh
; Access global memory 10,000 times to drive pin
; low for duration.
        RPT     #10000
        LACC    *, 0, AR0
; Restore GREG to 0000.
        SACH    GREG

*********************************************************************
* Zero output (switch the VCSELs off).                              *
*********************************************************************

        LAR     AR0, #zero        ; Load zero value location (AR0).
        MAR     *, AR0            ; Select auxiliary register AR0.
        OUT     *, 00000h, AR0    ; Send zero to output (VCSELs).
end1:   B       end1              ; Infinite loop.
```

```
*                   ****************************
*                   *        Calibration.      *
*                   * Starting at adress 0a40h. *
*                   ****************************

************************************************************************
* DSK board initialisation.                                            *
* See TI document SPRA253.pdf for further information.                 *
************************************************************************

; Initialise TMS320C50 On-chip timer.
; Load PRD register for period of 100nsec TDDR=0
        SPLK    #01h, PRD
; Re-load and begin timer.
        SPLK    #20h, TCR

; Serial port initialisation.
; FSM=1, XRST and RRST=00
        SPLK    #08h, SPC
; FSM=1, XRST and RRST=11
        SPLK    #0c8h, SPC

; AIC Initialisation.
; Init 8000h-FFFFh as global memory.
        LACC    #80h
; Store to global memory alloc register.
        SACL    GREG
; Use AR0 to point to location FFFFh.
        LAR     AR0, #0FFFFh
; Access global memory 10,000 times to drive pin
; low for duration.
        RPT     #10000
        LACC    *, 0, AR0
; Restore GREG to 0000.
        SACH    GREG

************************************************************************
* Calibration.                                                         *
************************************************************************

        LAR     AR0, #temp      ; Load location of temporary input (AR0).
        LAR     AR1, #av        ; Location of word used for averaging (AR1).
        LAR     AR2, #noise     ; Location of noise matrix (AR2).
        LAR     AR4, #oorder    ; Location of zero order matrix (AR4).
        LAR     AR6, #sel       ; Location of selection words (AR6).
        LAR     AR7, #zero      ; Location of zero value (AR7).
        MAR     *, AR0          ; Select AR0 register.

; Noise and zeroth order values for the first channel (2 neurons):
        CALL    ch1             ; Call input subroutine for channel 1.
        CALL    sub0            ; Call subroutine 0.
ini1:   CALL    sub1            ; Call subroutine 1.
cha1:   CALL    ch1             ; Call input subroutine for channel 1.
        CALL    sub2            ; Call subroutine 2.
        BGZ     cha1            ; Branch cha1 if accumulator>0.
        CALL    sub3            ; Call subroutine 3.
        BGZ     ini1            ; Branch ini1 if accumulator>0.

; The same for the seven other channels (remaining 14 neurons):
        CALL    ch2
        CALL    sub0
ini2:   CALL    sub1
cha2:   CALL    ch2
        CALL    sub2
        BGZ     cha2
        CALL    sub3
        BGZ     ini2

        CALL    ch3
        CALL    sub0
ini3:   CALL    sub1
cha3:   CALL    ch3
        CALL    sub2
```

```
        BGZ     cha3
        CALL    sub3
        BGZ     ini3

        CALL    ch4
        CALL    sub0
ini4:   CALL    sub1
cha4:   CALL    ch4
        CALL    sub2
        BGZ     cha4
        CALL    sub3
        BGZ     ini4

        CALL    ch5
        CALL    sub0
ini5:   CALL    sub1
cha5:   CALL    ch5
        CALL    sub2
        BGZ     cha5
        CALL    sub3
        BGZ     ini5

        CALL    ch6
        CALL    sub0
ini6:   CALL    sub1
cha6:   CALL    ch6
        CALL    sub2
        BGZ     cha6
        CALL    sub3
        BGZ     ini6

        CALL    ch7
        CALL    sub0
ini7:   CALL    sub1
cha7:   CALL    ch7
        CALL    sub2
        BGZ     cha7
        CALL    sub3
        BGZ     ini7

        CALL    ch8
        CALL    sub0
ini8:   CALL    sub1
cha8:   CALL    ch8
        CALL    sub2
        BGZ     cha8
        CALL    sub3
        BGZ     ini8

end2:   B       end2            ; Infinite loop.

*********************************************************************
* Calibration. Subroutines.                                         *
*********************************************************************

; Subroutine 0: initialisation of the storing word following "three"
; in memory. It indicates one of the three stages in the calibration:
sub0:   MAR     *, AR5          ; Select AR5.
        LAR     AR5, #three     ; Load location of the value three (AR5).
        LAC     *+, 0, AR5      ; Load accumulator with this value.
        SACL    *, 0, AR0       ; Store this value in the next word.
        RET                     ; Return from subroutine.

; Subroutine 1: initialisation of the averaging words and the storing
; word following "itnb". It indicaties the current iteration:
sub1:   MAR     *, AR1          ; Select AR1 (averaging word).
        RPT     #100            ; Repeat 100 times NOP.
        NOP                     ; No operation: wait for conversion.
        LAC     #00000h         ; Load accumulator with the value 0.
        SACL    *+, 0, AR1      ; Store 0 in first averaging word.
        SACL    *-, 0, AR3      ; Store 0 in second averaging word.
        LAR     AR3, #itnb      ; Location of number of iterations (AR3).
        LAC     *+, 0, AR3      ; Load accu with this value (256).
        SACL    *, 0, AR0       ; Store this value in the next word.
```

```
        RET                     ; Return from subroutine.

; Subroutine 2: Separate the two input values (one word) and add them
; to the previous inputs for averaging. This is done 256 times (number
; of iterations = 256):
sub2:   LAC     *, 8, AR0       ; Load input value from ADCs.
        SACH    *+, 0, AR0      ; Store first input value.
        SACL    *, 0, AR0       ; Store second input value (shifted).
        LAC     *, 8, AR0       ; Load and shift second value.
        SACH    *-, 0, AR1      ; Store second input value again.
        LAC     *, 0, AR0       ; Load the first averaging word.
        ADD     *+, 0, AR1      ; Add the first input value.
        SACL    *+, 0, AR1      ; Store in the first averaging word.
        LAC     *, 0, AR0       ; Load the second averaging word.
        ADD     *-, 0, AR1      ; Add the second input value.
        SACL    *-, 0, AR3      ; Store in the second averaging word.
        RPT     #100            ; Repeat 100 times NOP.
        NOP                     ; No operation: wait for conversion.
        LAC     *, 0, AR3       ; Load the remaining number of
        SUB     #1              ; iterations and subtract 1.
        SACL    *, 0, AR0       ; Store the value.
        RET                     ; Return from subroutine.

; Subroutine 3: According to the calibration stage, store average
; values in the noise or zeroth order matrices:
sub3:   MAR     *, AR5          ; Select AR5 (calibration stage).
        LAC     *, 0, AR1       ; Load number of calibration stage.
        SUB     #1              ; Subtract 1.
        BZ      un              ; If the value = 0, branch "un".
        SUB     #1              ; Subtract 1.
        BZ      deux            ; If the value = 0, branch "deux".
; If the calibration stage is number 3 (noise value):
        LAC     *+, 8, AR2      ; Load, shift first averaging value.
        SACH    *+, 0, AR1      ; Store in the first noise location.
        LAC     *-, 8, AR2      ; Load, shift second averaging value.
        SACH    *-, 0, AR6      ; Store in the second noise location.
        OUT     *+, 00000h, AR5 ; Send selection word (one VCSEL on).
        B       endsub          ; Branch end of subroutine 3.
; If the calibration stage is number 2 (first zeroth order value):
deux:   LAC     *, 8, AR4       ; Load, shift first averaging value.
        SACH    *, 0, AR4       ; Store: first zeroth order location.
        LAC     *, 0, AR2       ; Load the value again.
        SUB     *+, 0, AR4      ; Subtract the corresponding noise.
        SACL    *+, 0, AR6      ; Store the value again.
        OUT     *+, 00000h, AR5 ; Send selection word (one VCSEL on).
        B       endsub          ; Branch end of subroutine 3.
; If the calibration stage is number 1 (second zeroth order value):
un:     MAR     *+, AR1         ; Select second averaging word.
        LAC     *-, 8, AR4      ; Load, shift second averaging value.
        SACH    *, 0, AR4       ; Store: second zeroth order location.
        LAC     *, 0, AR2       ; Load the value again.
        SUB     *+, 0, AR4      ; Subtract the corresponding noise.
        SACL    *+, 0, AR7      ; Store the value again.
        OUT     *, 00000h, AR5  ; Send zero value (all VCSELs off).
endsub: LAC     *, 0, AR5       ; Load the number of the calibration
        SUB     #1              ; stage and subtract 1.
        SACL    *, 0, AR0       ; Store the value.
        RET                     ; Return from subroutine.



*                 ******************************
*                 * Neural network calculation. *
*                 * Starting at adress 0b28h.   *
*                 ******************************

*********************************************************************
* DSK board initialisation.                                         *
* See TI document SPRA253.pdf for further information.              *
*********************************************************************

; Initialise TMS320C50 On-chip timer.
; Load PRD register for period of 100nsec TDDR=0
        SPLK    #01h, PRD
```

```
; Re-load and begin timer.
        SPLK    #20h, TCR

; Serial port initialisation.
; FSM=1, XRST and RRST=00
        SPLK    #08h, SPC
; FSM=1, XRST and RRST=11
        SPLK    #0c8h, SPC

; AIC Initialisation.
; Init 8000h-FFFFh as global memory.
        LACC    #80h
; Store to global memory alloc register.
        SACL    GREG
; Use AR0 to point to location FFFFh.
        LAR     AR0, #0FFFFh
; Access global memory 10,000 times to drive pin
; low for duration.
        RPT     #10000
        LACC    *, 0, AR0
; Restore GREG to 0000.
        SACH    GREG

*********************************************************************
* Hardware synchronisation (interruption).                          *
*********************************************************************

        LAR     AR0, #syntyp      ; Load location of type of synchronisation.
        LAC     *, 0, AR0         ; Load type of synchronisation.
        BGZ     optics            ; If the type of synchronisation is different from
0, branch "optics".
dum:    B       dum               ; Infinite loop.
        CLRC    INTM              ; The C++ program branches this point. The
interruption mask bit is cleared.
        LAC     IMR               ; Load the interruption mask register.
        OR      #00001h           ; Set the bit that coreesponds to INT1\.
        SACL    IMR               ; Store the interruption mask register back.
        LAR     AR0, #zero        ; Load location of value 0.
        OUT     *, 00080h, AR0    ; Send 0 to output and use the address that is used
for the interruption.
wait:   B       wait              ; Infinite loop.

*********************************************************************
* Optical synchronisation.                                          *
* Initialisation of the averaging table "aver".                     *
*********************************************************************

optics: LAR     AR0, #aver        ; Load location of averaging table (AR0).
        LAR     AR5, #itnb        ; Location of number of iterations (AR5).
        MAR     *, AR5            ; Select AR5.
        LAC     *, 0, AR0         ; Load number of iterations (256).
a:      SPLK    #0, *+, AR0       ; Store 0 in averaging table.
        SUB     #1                ; Subtract 1 to accumulator.
        BGZ     a                 ; Branch "a" until accu = 0.

*********************************************************************
* Optical synchronisation.                                          *
*********************************************************************

        LAR     AR0, #syn         ; Load location of word indicating the VCSEL used
for the synchronisation (AR0).
        OUT     *, 00000h, AR0    ; Send word to output.

        LAR     AR0, #temp        ; Load location of temporary values (AR0).
        LAR     AR1, #av          ; Load location of averaging word (AR1).
        LAR     AR2, #synth       ; Load location of synchronisation threshold (AR2).
        LAR     AR4, #noise+6     ; Load location of noise value corresponding to the
detector used for the synchronisation (AR4).

        CALL    ch4               ; Call input subroutine for channel 4.
        RPT     #100              ; Repeat "NOP" 100 times: wait for the ADC
conversion.
        NOP
        MAR     *, AR1            ; Select auxiliary register AR1.
```

```
        LAC     #00000h         ; Load value 0 in accumulator.
        SACL    *, 0, AR5       ; Store value 0 in first averaging word
(initialisation).
loop:   LAR     AR3, #aver      ; Load location of beginning of averaging matrix
(AR3).
        LAR     AR5, #itnb      ; Load location of number of iterations for
averaging: 256 (AR5).
        LAC     *+, 0, AR5      ; Load accumulator with number of iterations.
        SACL    *, 0, AR0       ; Store number of iterations in following memory
word (initialisation of current itearation).
c:      CALL    ch4             ; Call input subroutine for channel 4.
        LAC     *, 8, AR0       ; Load and shift input values from the ADC to keep
the second one.
        SACH    *, 0, AR0       ; Store second input value (in the most significant
part of the accumulator).
        LAC     *, 0, AR1       ; Load the input value again (not shifted).
        ADD     *, 0, AR3       ; Add the current averaging value.
        SUB     *, 0, AR1       ; Subtract the oldest value in the averaging
matrix.
        SACL    *, 0, AR0       ; Store the averaging value obtained.
        LAC     *, 0, AR3       ; Load the input value again.
        SACL    *+, 0, AR1      ; Store the value in the averaging matrix.
        LAC     *+, 8, AR1      ; Load the averaging value calculated earlier and
shift it (to divide it by 256).
        SACH    *, 0, AR1       ; Store the higher part of the accumulator in the
second averaging word.
        LAC     *-, 0, AR4      ; Load that value again.
        SUB     *, 0, AR2       ; Subtract the noise corresponding to the detector
used.
        SUB     *, 0, AR5       ; Subtract the synchronisation threshold.
        BGEZ    then            ; If the result is greater or equal to 0, branch
"then": following of the program.
        RPT     #100            ; Repeat "NOP" 100 times: wait for the ADC
conversion.
        NOP
        LAC     *, 0, AR5       ; Load the number of the current iteration (out of
256).
        SUB     #1              ; Subtract 1.
        SACL    *, 0, AR0       ; Store the value.
        BGZ     c               ; Branch "c" if the result is greater than 0.
        MAR     *, AR5          ; Select AR5.
        B       loop            ; Branch "loop" unconditionnally when this point is
reached.

*********************************************************************
* Neural network calculation. Initialisation and output.            *
*********************************************************************

then:   LDP     #66             ; Load data memory page pointer with the value 66
which indicates adresses 21xxh.
        LAR     AR0, #number+1  ; Load location of the word following the number of
requests and results (AR0).
        MAR     *, AR0
        SPLK    #0, *, AR0      ; Initialisation of this word (value 0): it
indicates the number of the current request.
        LAR     AR0, #reqloc    ; Load location of the word containing the current
request matrix location (AR0).
        SPLK    #01000h, *, AR0 ; Initialisation of this word (value 1000h,
beginning of the request matrices).
        LAR     AR0, #resloc    ; Load location of the word containing the current
result word location (AR0).
        SPLK    #02000h, *, AR0 ; Initialisation of this word (value 2000h,
beginning of the result words).

; Beginning of the loop for a new request matrix.
next:   LAR     AR0, #oneit     ; Load location of the word indicating if the
request are multiple or not (AR0).
        LAC     *, 0, AR0       ; Load accumulator with that value (1 if single
request, 0 if multiple)
        BGZ     sinit           ; Branch "sinit" if the value is greater than 0.
        LAR     AR0, #mem       ; Load location of the memory matrix (AR0).
        LAC     #00010h         ; Load accumulator with the value 16.
imem:   SPLK    #0, *+, AR0     ; Initialisation of the memory matrix values (value
0).
```

```
        SUB     #1              ; Subtract 1 to accumulator (will be done 16
times).
        BGZ     imem            ; Branch "imem" if the result is greater than 0.
        LAR     AR0, #out       ; Load location of output word (AR0).
        SPLK    #0, *, AR0      ; Initialise output word (value 0).
sinit:  LAR     AR0, #iter      ; Load location of number of iterations (AR0).
        LAC     *+, 0, AR0      ; Load accumulator with number of iterations.
        SACL    *, 0, AR0       ; Store the value in the following memory word:
indicates the current iteration.

begin:  LAR     AR0, #delay     ; Load location of the synchronisation delay word
(AR0).
        LAC     *, 0, AR4       ; Load the synchronisation delay value (n).
count1: SUB     #1              ; Subtract 1: this will be done n times.
        BGZ     count1          ; Branch "count1" if the result is greater than 0.
        LAR     AR4, #out       ; Load output word location (AR4).
        OUT     *, 00000h, AR0  ; Send output word to output (VCSELs are switched
on or off).
        LAC     *, 0, AR0       ; Load the synchronisation delay value (n).
count2: SUB     #1              ; Subtract 1: this will be done n times.
        BGZ     count2          ; Branch "count2" if the result is greater than 0.


*********************************************************************
* Neural network calculation. Input.                                 *
*********************************************************************

        LAR     AR0, #in        ; Load input matrix location (AR0).
        CALL    ch1             ; Call input subroutine for channel 1.
        RPT     #100            ; Repeat "NOP" 100 times: wait for the ADC
conversion.
        NOP
        LAR     AR1, #sel       ; Load selection values location (AR1).
        LAR     AR2, #oorder    ; Load oorder matrix location (AR2).
        LAR     AR3, reqloc     ; Load location of the current request value,
contained in memory word "reqloc" (AR3).
        LAR     AR6, #mem       ; Load memory matrix location (AR6).
        LAR     AR7, #noise     ; Load noise matrix location (AR7).
        CALL    ch2             ; Call input subroutine for channel 2.
        CALL    calc            ; Call subroutine "calc".
        CALL    ch3             ; Call input subroutine for channel 3.
        CALL    calc            ; Call subroutine "calc".
        CALL    ch4             ; Call input subroutine for channel 4.
        CALL    calc            ; Call subroutine "calc".
        CALL    ch5             ; Call input subroutine for channel 5.
        CALL    calc            ; Call subroutine "calc".
        CALL    ch6             ; Call input subroutine for channel 6.
        CALL    calc            ; Call subroutine "calc".
        CALL    ch7             ; Call input subroutine for channel 7.
        CALL    calc            ; Call subroutine "calc".
        CALL    ch8             ; Call input subroutine for channel 8.
        CALL    calc            ; Call subroutine "calc".
        CALL    ch1             ; Call input subroutine for channel 1.
        CALL    calc            ; Call subroutine "calc".

        LAR     AR4, #iter+1    ; Load location of current iteration number (AR4).
        MAR     *, AR4          ; Select register AR4.
        LAC     *, 0, AR4       ; Load accumulator with current iteration number.
        SUB     #1              ; Decrement iteration number.
        SACL    *, 0, AR4       ; Store iteration number.
        BGZ     begin           ; Branch beginning of loop for the next iteration if
the iteration number is greater than 0.

        LAR     AR0, #out       ; Load output word location (AR0).
        LAR     AR1, resloc     ; Load location of current result word (AR1).
        LAR     AR2, #resloc    ; Load location of word containing the address of the
current result word (AR2).
        MAR     *, AR0          ; Select register AR0.
        LAC     *, 0, AR1       ; Load accumulator with output word.
        SACL    *, 0, AR2       ; Store the word at the current result location.
        LAC     *, 0, AR2       ; Load accumulator with result location word.
        ADD     #1              ; Add 1: increment the result location word, move on
to the next result.
```

```
        SACL    *, 0, AR0       ; Store the value obtained in the word containing the
address of the current result word.

        LAR     AR0, #reqloc    ; Load location of word containing the address of the
current request matrix (AR0).
        LAC     *, 0, AR0       ; Load acumulator with request location word.
        ADD     #16             ; Add 16: increment the request matrix location, move
on to the next request.
        SACL    *, 0, AR0       ; Store the value obtained.

        LAR     AR0, #number+1  ; Load location of the current request (AR0).
        LAC     *, 0, AR0       ; Load accumulator with the current request number.
        ADD     #1              ; Add 1: increment the current request number.
        SACL    *-, 0, AR0      ; Store the value back.
        SUB     *, 0, AR0       ; Subtract the total number of requests (word that is
just before).
        BLZ     next            ; Branch beginning of calculation for the next
request matrix if needed.

end3:   B       end3            ; Inifinite loop.

*********************************************************************
* Neural network calculation (subroutine "calc").                   *
*********************************************************************

calc:   LAC     *, 8, AR0       ; Load input values from ADCs. The two values form a
single word (two bytes) and will be separated.
        SACH    *+, 0, AR0      ; Store first input value.
        SACL    *, 0, AR0       ; Store second input value (shifted).
        LAC     *, 8, AR0       ; Load and shift second value.
        SACH    *-, 0, AR0      ; Store second input value again.
        LAC     *, 0, AR7       ; Load first input value.
        SUB     *+, 0, AR0      ; Subtract the corresponding noise value.
        SACL    *+, 0, AR0      ; Store the first input value back.
        LAC     *, 0, AR7       ; Load second input value.
        SUB     *+, 0, AR0      ; Subtract the corresponding noise value.
        SACL    *-, 0, AR4      ; Store the second input value back.

        LAC     *, 0, AR1       ; Load the current output word.
        AND     *+, AR0         ; Logical "and" with a selection word: selects the
bit corresponding to the neuron concerned in the output word.
        BZ      noton1          ; If the result equals 0, branch "noton1", the VCSEL
is not on.
        LAC     *, 0, AR2       ; If the VCSEL is on, load the first input value
again.
        SUB     *, 0, AR0       ; Subtract the corresponding zeroh order value.
        SACL    *, 0, AR0       ; Store the first input value back.
noton1: MAR     *+, AR2         ; Increment AR0 and select AR2.
        MAR     *+, AR4         ; Increment AR2 and select AR4.
        LAC     *, 0, AR1       ; Load output word.
        AND     *-, AR0         ; Logical "and" with the following selection word.
        BZ      noton2          ; If the result equals 0, branch "noton2", the VCSEL
is not on.
        LAC     *, 0, AR2       ; If the VCSEL is on, load the second input value
again.
        SUB     *, 0, AR0       ; Subtract the corresponding zeroth order value.
        SACL    *, 0, AR0       ; Store the second input value back.
noton2: MAR     *-, AR2         ; Decrement AR0 and select AR2.
        MAR     *+, AR0         ; Increment AR2 and select AR0.

; Calculate the first memory value, using the first input value.
        LAR     AR5, #store       ; Load storing word location (AR5).
        LT      0                 ; Load the T register with the value of A.
        MPY     *+, AR6           ; Multiply by the first input value.
        PAC                       ; Load accumulator with result.
        NEG                       ; Opposite value of the accumulator.
        SACB                      ; Store accumulator in accumulator B.
        LAC     *, 12, AR6        ; Load accumulator with memory value.
        ADDB                      ; Add both accumulators.
        ADD     1, 4              ; Add the value of B.
        SACH    *, 4, AR6         ; Store the memory value obtained.
; Calculate address and read value of corresponding sigmoid function.
        LAC     *, 0, AR6         ; Load accumulator with the memory value.
```

```
        SUB     #08000h         ; Subtract 8000h: a value lower than 8000h is
positive and a value greater or equal to 8000h is negative.
        BGEZ    a1              ; Branch "a1" if the value is negative.

; If the memory value is positive:
        LAC     *, 14, AR6      ; Load and shift the memory value.
        SACH    temp            ; Store the memory value divided by 4 in the
temporary location "temp".
        LAC     *, 0, AR6       ; Load the memory value again.
        SUB     #003ffh         ; Subtract 3ffh: if the value is greater than 3ffh,
the sigmoid function value is 1.
        BLEZ    b1              ; Branch "b1" if the result is lower or equal to 0.
        LAR     AR5, #one       ; Load location of the word containing the value 1
(format Q12) (AR5).
        MAR     *+, AR5         ; Increment AR6 and select AR5.
        LT      *, AR3          ; Load the T register with the value 1 (format
Q12).
        B       c1              ; Branch "c1" unconditionnally.

; If the memory value is negative:
a1:     LAC     *, 14, AR6      ; Load and shift the memory value.
        SACH    temp            ; Store the memory value divided by 4 in the
temporary location "temp".
        LAC     temp            ; Load the value of "temp".
        ADD     #0c000h         ; Add c000h
        SACL    temp            ; Store the value obtained in "temp".
        LAC     *, 0, AR6       ; Load the memory value again.
        SUB     #0fc00h         ; Subtract fc00h: if the value is lower or equal to
fc00h, the sigmoid function value is 0.
        BGEZ    b1              ; Branch "b1" if the result is greater or equal to
0.
        LAR     AR5, #zero      ; Load location of the word containing the value 0
(AR5).
        MAR     *+, AR5         ; Increment AR6 and select AR5.
        LT      *, AR3          ; Load the T register with the value 0.
        B       c1              ; Branch "c1" unconditionnally.

b1:     MAR     *+, AR5         ; Increment AR6 and select AR5.
        LAC     temp            ; Load the value of "temp".
        ADD     #00e00h         ; Add 0e00 to have the locvation of the sigmoid
function value.
        SACL    *, 0, AR5       ; Store the value at the location "store".
        LAR     AR5, store      ; Load AR5 with location of the sigmoid function
value, contained in the word "store".
        LT      *, AR3          ; Load the T register with the sigmoid function
value.
; Multiply by request value.
c1:     MPY     *+, AR1         ; Multiply by the first request value.
        PAC                     ; Load the accumulator with the result.
; Compare with threshold value (format Q20).
        SUB     2, 4
        BGEZ    on1             ; Branch "on1" if result <0 or =0.
; Switch off corresponding VCSEL if result >0.
off1:   LAC     *+, 0, AR4      ; Load accumulator with selection word.
        XOR     #0ffffh         ; Complement of selection word.
        AND     *, AR4          ; Switch bit to 0 in output.
        SACL    *, 0, AR0       ; Store output word.
        B       then1           ; Branch second part of subroutine.
; Switch on corresponding VCSEL.
on1:    LAC     *+, 0, AR4      ; Load accumulator with selection word.
        OR      *, AR4          ; Switch bit to 1 in output.
        SACL    *, 0, AR0       ; Store output word.

; Move on to the second memory value, using the second input value.
; The program is the same as for the irst memory value.
then1:  LAR     AR5, #store
        LT      0
        MPY     *+, AR6
        PAC
        NEG
        SACB
        LAC     *, 12, AR6
        ADDB
        ADD     1, 4
```

```
        SACH    *, 4, AR6
        LAC     *, 0, AR6
        SUB     #08000h
        BGEZ    a2

        LAC     *, 14, AR6
        SACH    temp
        LAC     *, 0, AR6
        SUB     #003ffh
        BLEZ    b2
        LAR     AR5, #one
        MAR     *+, AR5
        LT      *, AR3
        B       c2

a2:     LAC     *, 14, AR6
        SACH    temp
        LAC     temp
        ADD     #0c000h
        SACL    temp
        LAC     *, 0, AR6
        SUB     #0fc00h
        BGEZ    b2
        LAR     AR5, #zero
        MAR     *+, AR5
        LT      *, AR3
        B       c2

b2:     MAR     *+, AR5
        LAC     temp
        ADD     #00e00h
        SACL    *, 0, AR5
        LAR     AR5, store
        LT      *, AR3
c2:     MPY     *+, AR1
        PAC
        SUB     2, 4
        BGEZ    on2
off2:   LAC     *+, 0, AR4
        XOR     #0ffffh
        AND     *, AR4
        SACL    *, 0, AR0
        B       then2
on2:    LAC     *+, 0, AR4
        OR      *, AR4
        SACL    *, 0, AR0
then2:  RET                     ; Return from subroutine.

        .end                    ; End of program.
```

Appendix 2: The optoelectronic neural network demonstrator program (NNetDem) windows:

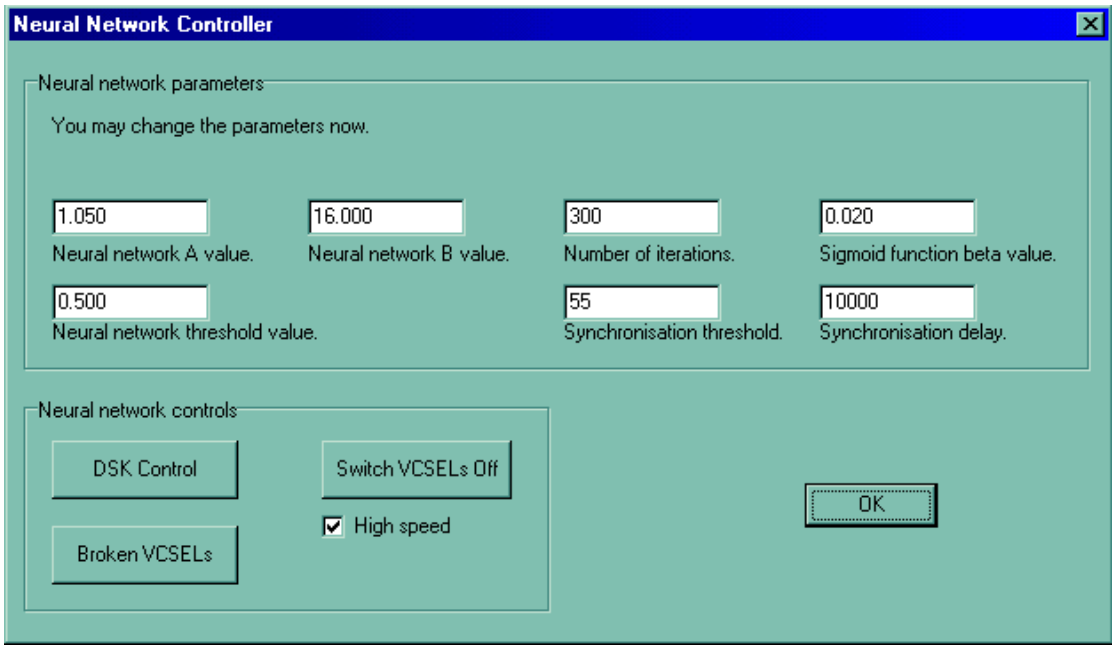


Figure 1: optoelectronic neural network demonstrator control module window.

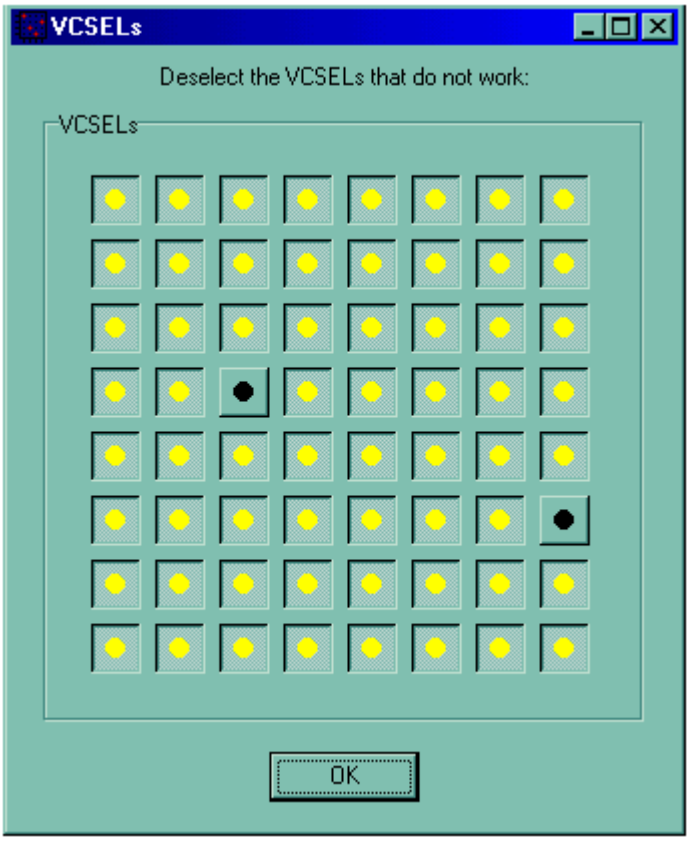


Figure 2: VCSELs form window.

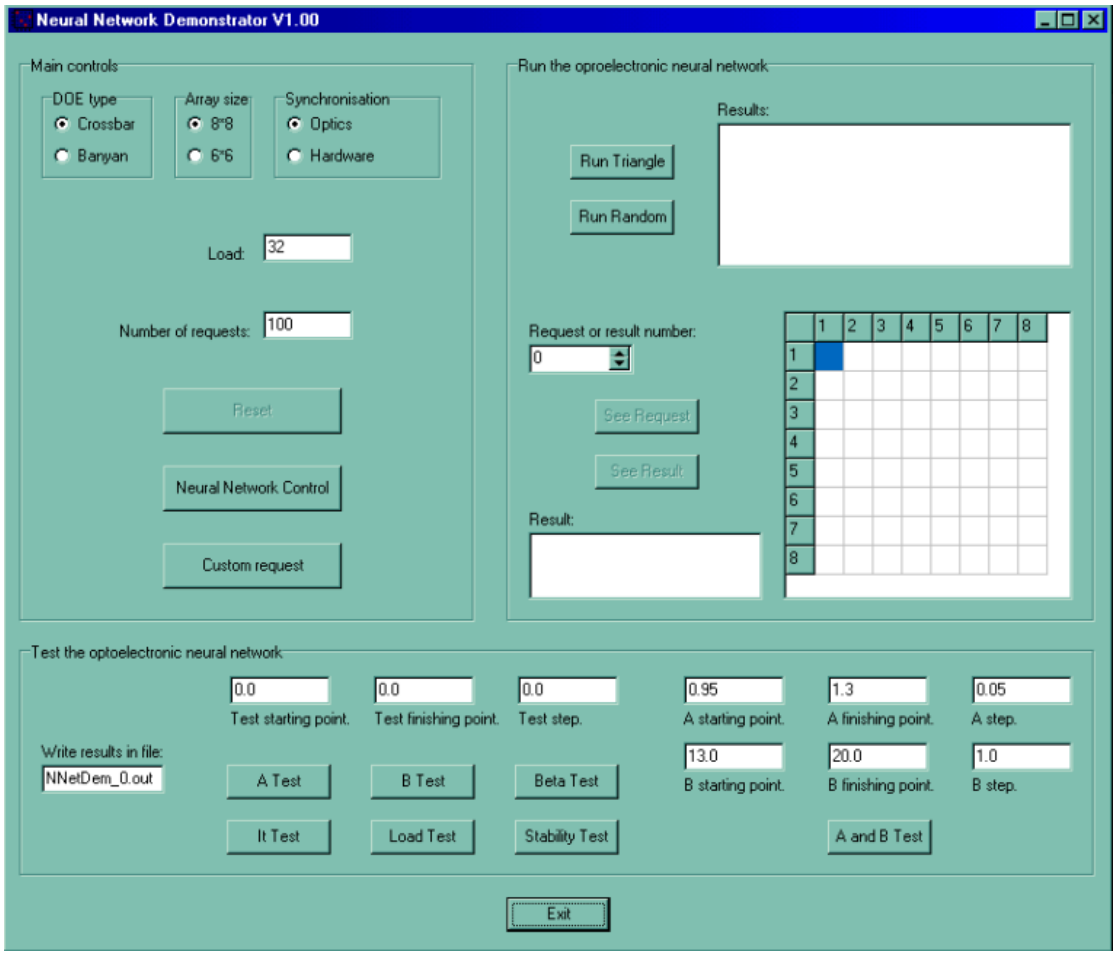


Figure 3: optoelectronic neural network demonstrator main program window.

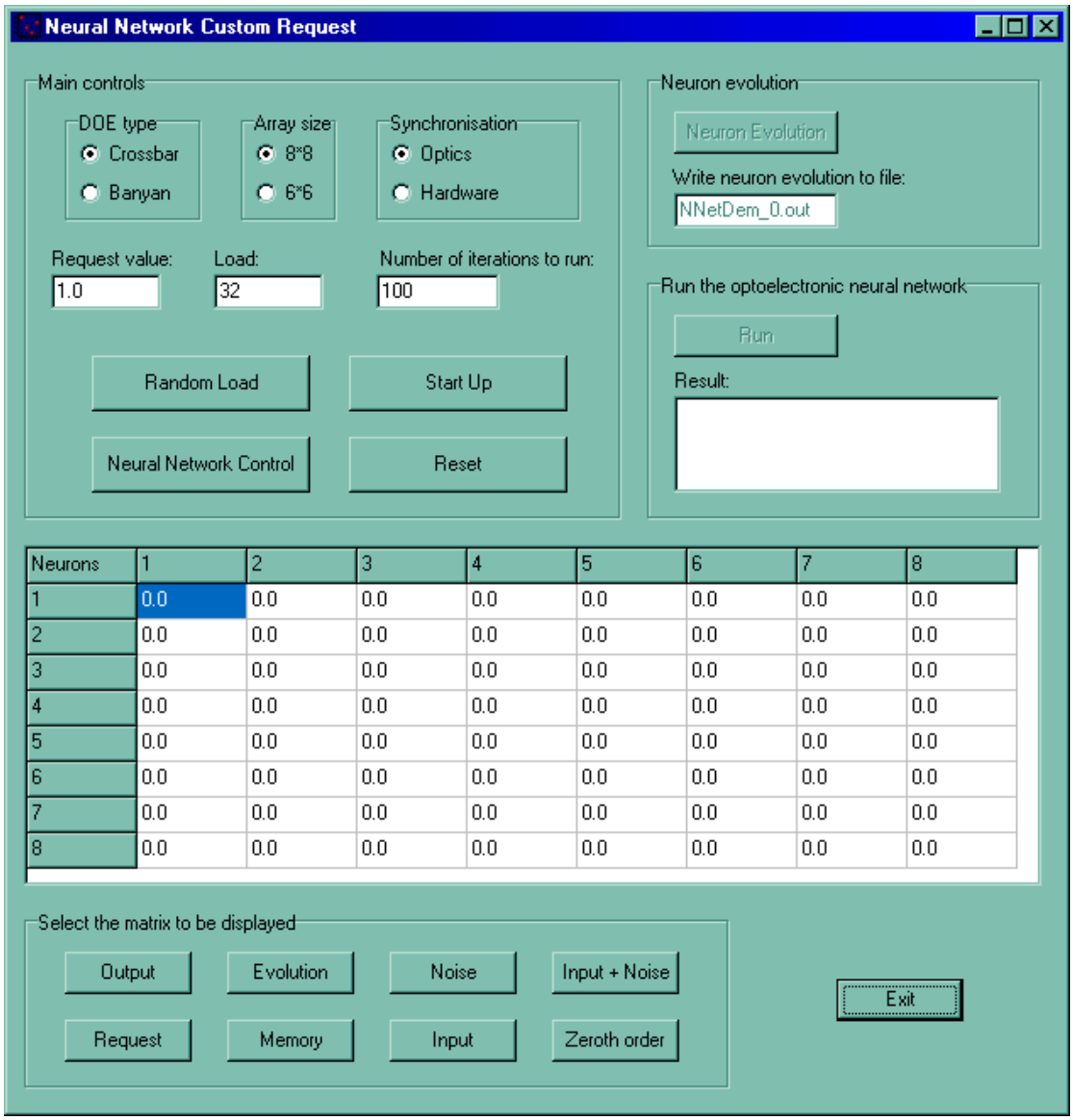


Figure 4: custom request window.

Appendix 3: Crossbar switch results:

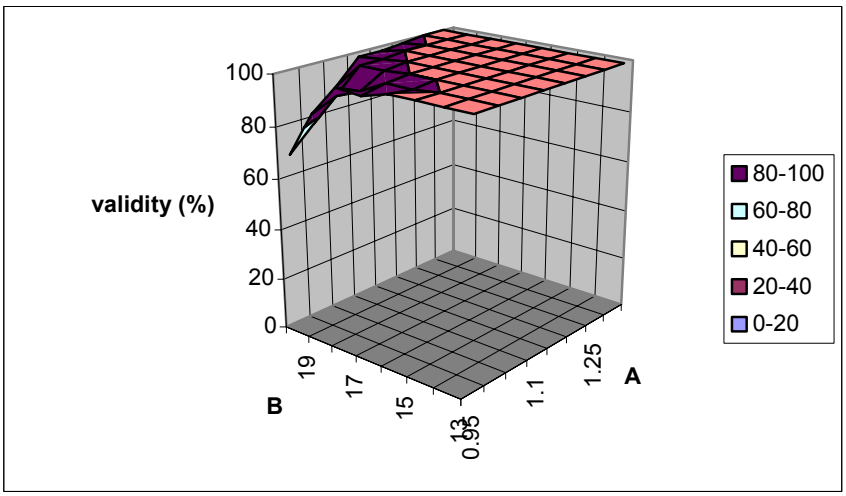


Figure 1: validity as a function of A and B (8×8).

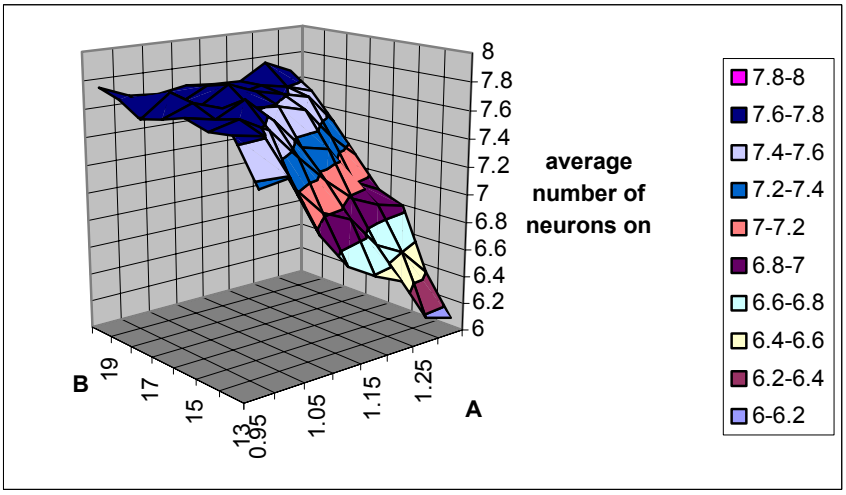


Figure 2: average number of neurons on as a function of A and B (8×8).

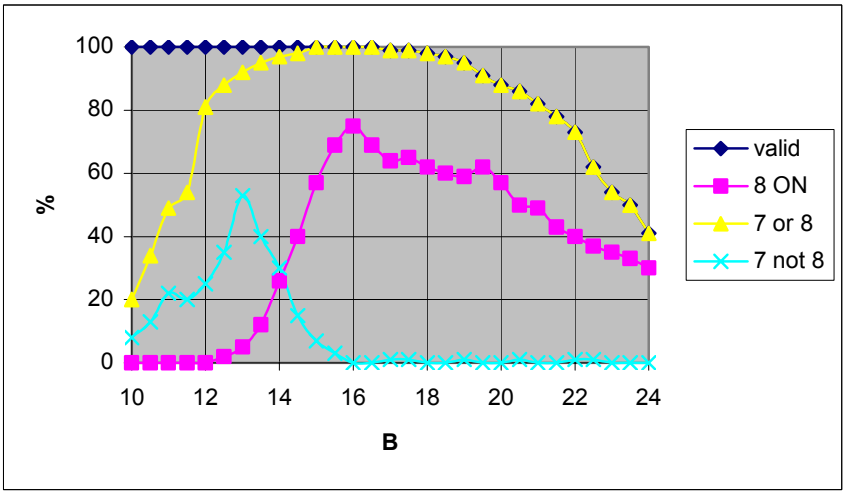


Figure 3: results as functions of B (8×8).

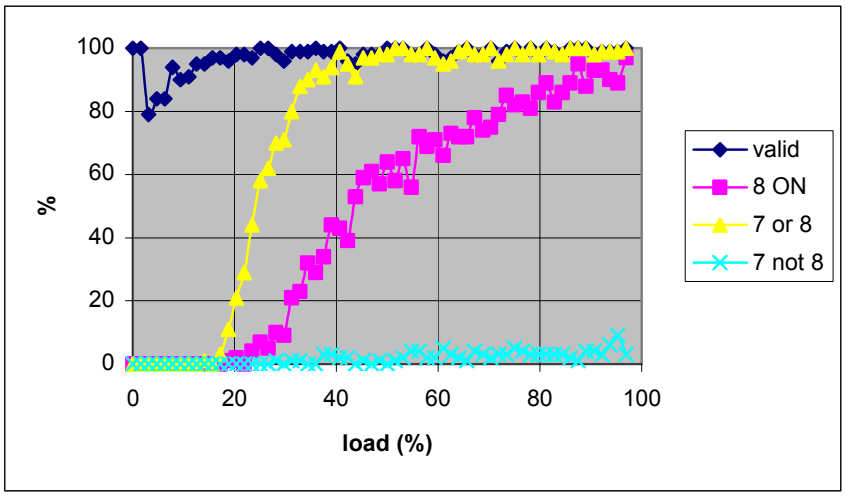


Figure 4: results as functions of the load (8×8, B = 16).

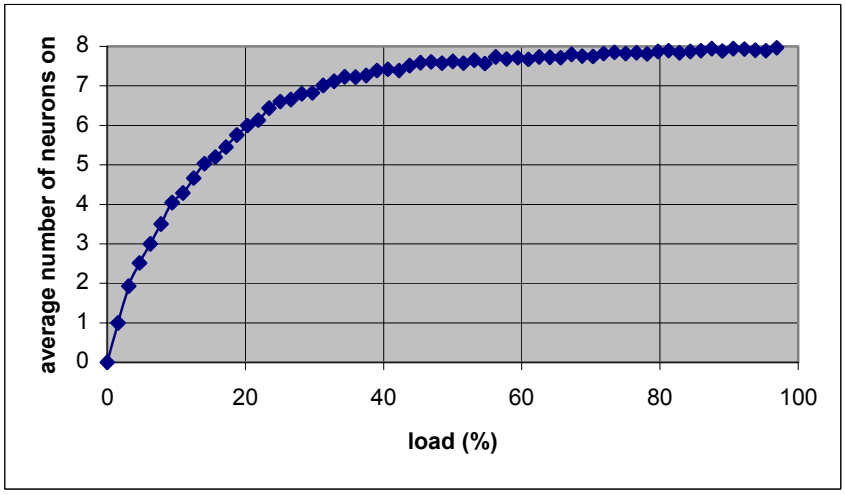


Figure 5: average number of neurons on as a function of the load (8×8, B = 16).

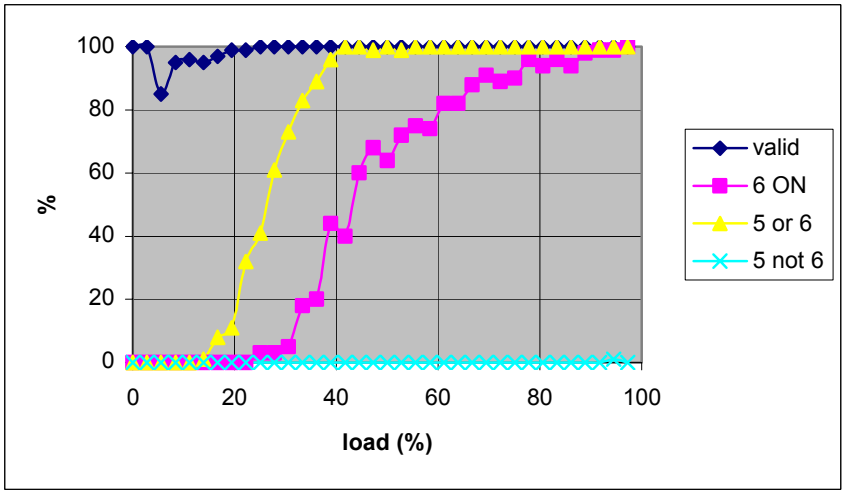


Figure 6: results as functions of the load (6×6, B = 16).

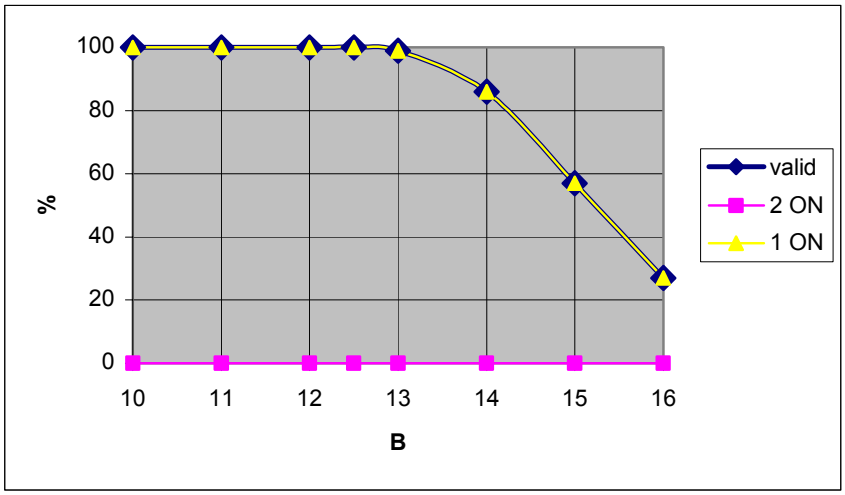


Figure 7: results as functions of B for two neurons requested in the same row/column.

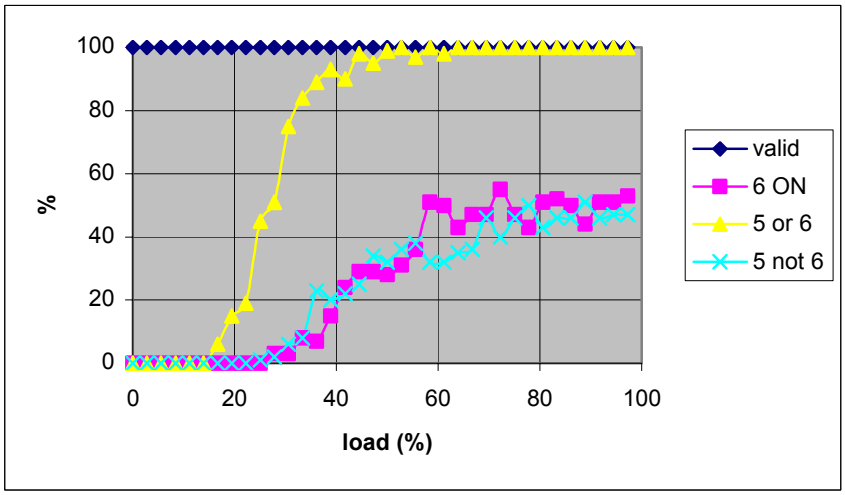


Figure 8: results as functions of the load (6×6, B = 12).

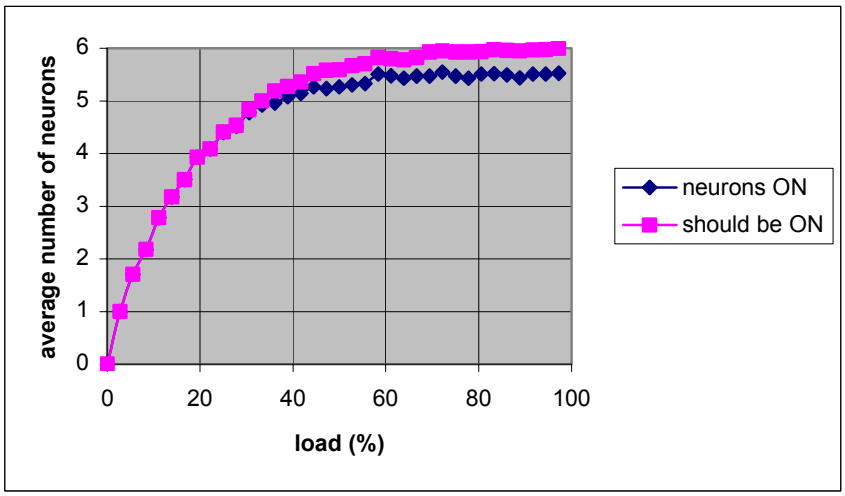


Figure 9: average number of neurons on as a function of the load (6×6, B = 12).

Appendix 4: Banyan switch results:

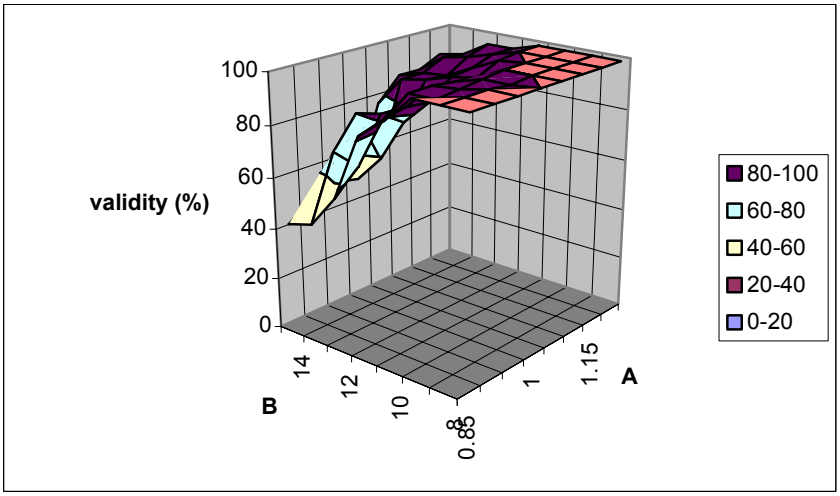


Figure 1: validity as a function of A and B.

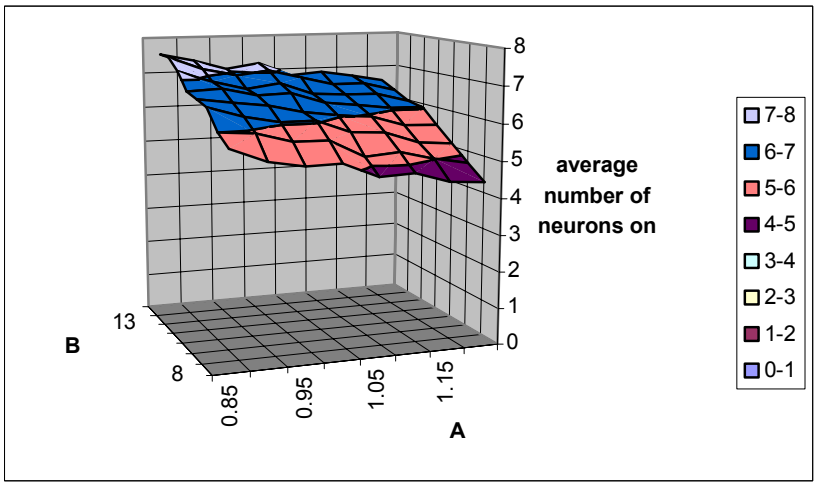


Figure 2: average number of neurons on as a function of A and B.

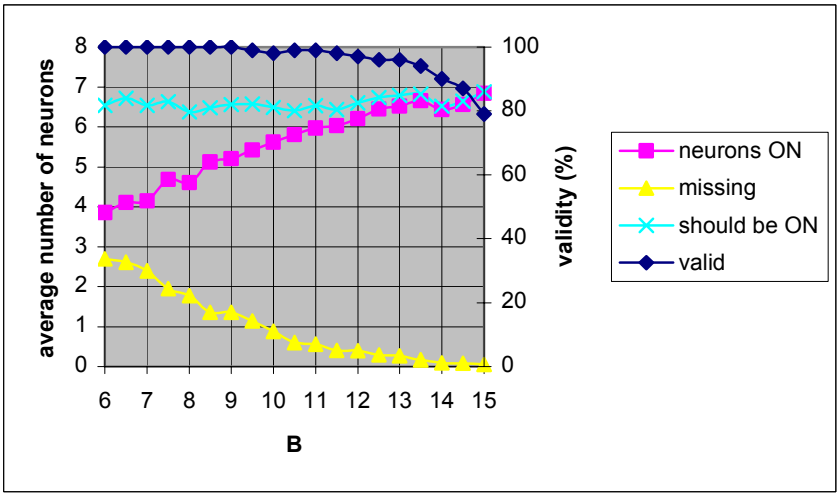


Figure 3: results as functions of B.

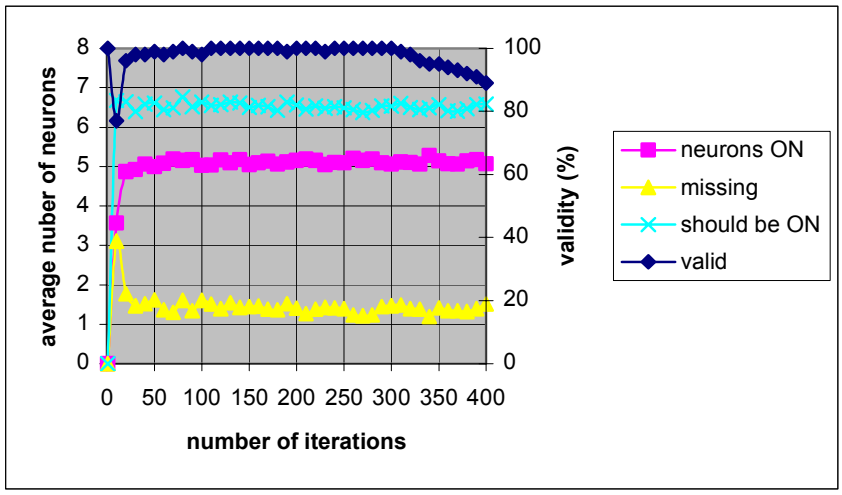


Figure 4: results as functions of the number of iterations.

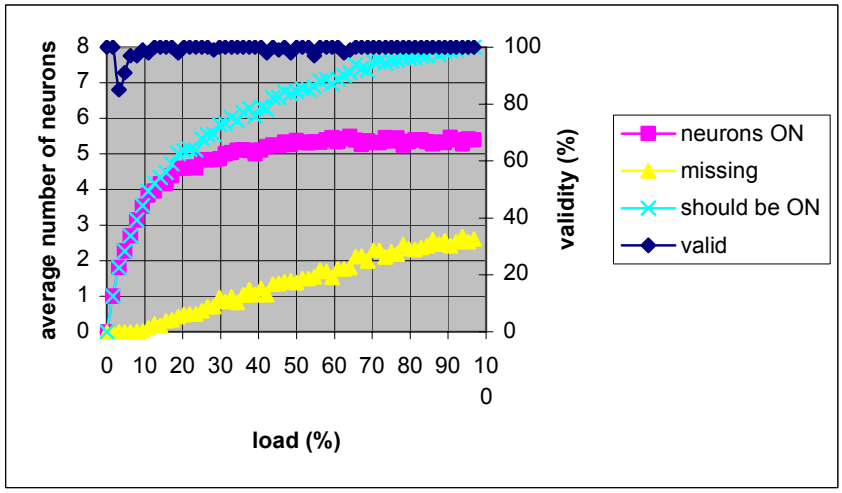


Figure 5: results as functions of the load (B = 9).

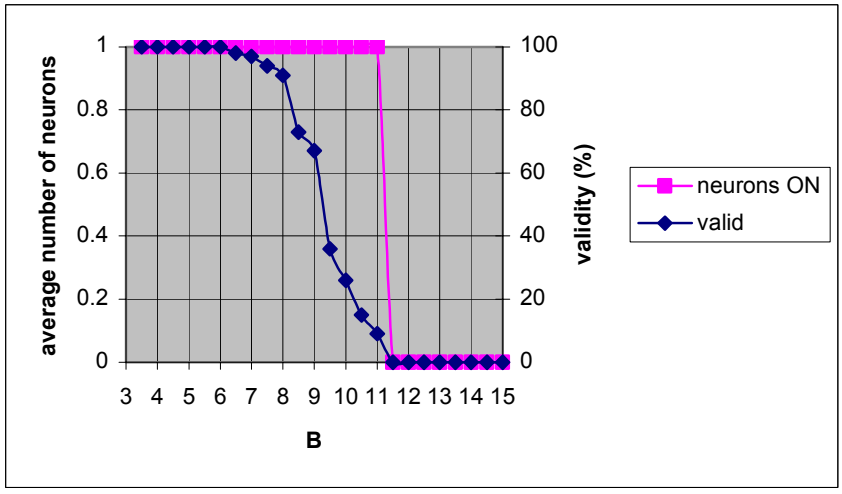


Figure 6: results as functions of B for two neurons requested in the same row/column.

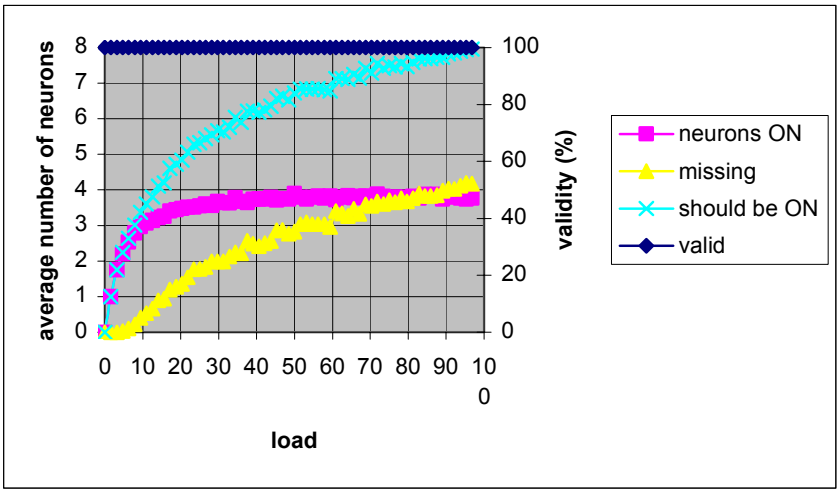


Figure 7: results as functions of the load (B = 6).

Appendix 5: Neuron evolution during optimisation:

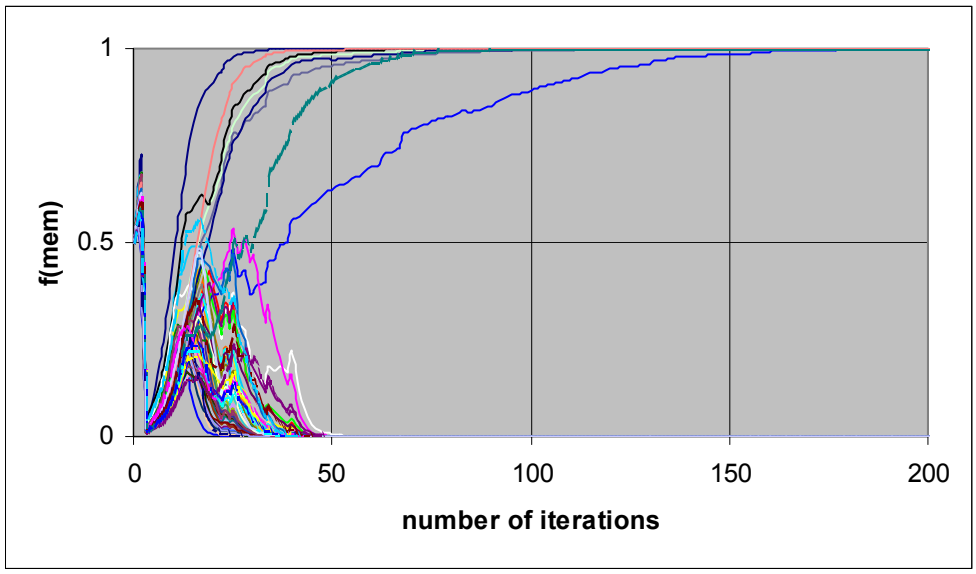


Figure 1: example of the evolution of the neurons during an optimisation.

	1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1
4	1	1	0	1	1	1	1	1
5	1	1	1	1	1	1	1	1
6	1	1	1	1	1	1	1	0
7	1	1	1	1	1	1	1	1
8	1	1	1	1	1	1	1	1

Table 1: the request matrix.

	1	2	3	4	5	6	7	8
1	1	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0
3	0	0	0	0	0	0	1	0
4	0	0	0	0	0	0	0	1
5	0	0	0	1	0	0	0	0
6	0	1	0	0	0	0	0	0
7	0	0	0	0	1	0	0	0
8	0	0	0	0	0	1	0	0

Table 2: the result matrix.