New Ideas for Nonlinear Distributed Control
Part 1 : Numerical Tools

R. Le Riche* and I. Kroo**
*CNRS, Institut de Mécanique de Rouen,
Le Madrillet, 76300 Saint Etienne du Rouvray, France.
** Dept. of Aeronautics and Astronautics, Stanford Univ.,
Stanford, CA 94035, USA.

December 20, 1997

1 Introduction

Neural networks and stochastic optimization methods have opened the way to systematic approaches for tackling nonlinear, chaotic control problems [3]. Neural networks have been applied to control as a result of their properties of "universal approximation" [1] and "parsimony" [2]. Those properties suggest that very general multidimensional mappings can be performed by neural networks. Neural networks are therefore an algorithmic tool for generating any type of controller. This powerful potential of neural networks has been widely acknowledged in the last decade, as is illustrated by the amount of litterature generated on the subject.

Versatility of neural networks hides the accompanying drawback of complexity in devising the right neural controller. Various structures of networks are being used (feedforward "ridge" networks and radial basis functions being the two most common nonrecurrent forms). Once a structure of network has been chosen, tuning its parameters represents a high dimension, nonlinear optimization problem. Recent developpements in the field of stochastic optimization, especially evolutionary methods ([4], [5]), and increasing flexibility and availability of computers offer nevertheless promising solutions to the neural network determination problem (see [6] for a review of combined evolutionary algorithms and neural networks).

The present project will investitage the use of neural networks for controlling aircrafts in turbulent atmosphere. Contributions of neural networks and evolutionary calculation methods to aircraft control might be determinant in the presence of wind gusts. First, neural networks have a good ability to handle noise in inputs because they classify input signals. Second, evolutionary optimizers, because they are probabilistic methods that work on a set of controllers are more resistant to noise in the evaluation of the objective function than gradient based methods. Third, deriving gradients of the error over a given time range is a costly procedure that can be avoided by evolutionary method.

This report is a second update on the work already performed. So far, the work has been concerned with programming necessary numerical tools. Three modules have been implemented: a neural network generator, a collection of optimization programs, and a longitudinal flight simulator. All programs have been written in C++. The reason for going through the tedious programming of widespread numerical techniques such as neural networks (see for example the MatLab toolbox, or the SNNS [9]) is that, for futur research developments, a fully dynamic, general, and open computer program will be necessary.

We first describe the main features of the neural network generator, next show some of the tests performed to validate it, and give some practical results for training neural networks. Finally, we describe the flight simulator and the atmospheric disturbance models implemented.

2 Neural Network Generator

The neural network implements "generalized neurons". Generalized neurons are any single input single output transfer function. They can be connected to any other neuron, including themself.

The C++ description of the class of generalized neurons, `GNEURON`, is given in Figure 1. Predecessors to a given neuron have their address stored in the array of pointers[1] to neurons `predecessors`. Each neuron also has a list f pointers to its successors, `successors`, which is usefull for recurring derivative calculation. The outputs of the neuron are represented by the array `out`. All arrays and lists are dynamically sized during execution, making the number of inputs and outputs to any neuron free to change. Apart from connections to predecessors and output arrays, neurons have methods to load their characteristics (`load`), evaluate their output (`evaluate`), and write themselves out (`write`). All specific neurons derive from the abstract `GNEURON` class. Three instances of neurons have been implemented, namely `input` neurons, `sigmoid` neurons, and `linear` neurons. It is nevertheless very easy to add other neurons to the simulator, since it does not involve changing any of the existing code. All what is required are load, write and evaluate functions, and a class interface such as that of Figure 2. To account for potential new neurons, use was made of the dynamic inheritance features of C++. The function `link` of the `GNEURAL_NET` class (see Figure 3) is in charge of connecting the neurons by building `ARRAY<GNEURON*> predecessors` and `LIST<GNEURON*> successors`. Before `link` is called, each neuron only

[1] use has been made of nonstandard C++ tools such as arrays, lists, vectors, matrices, tensors, selfdestructing pointers, strings, ascii files, etc., that are part of the library developped in [10].

```
class GNEURON {

  protected :
    STRING name;
    CARRAY<CONNECTION> inputs;
    ARRAY<GNEURON*> predecessors;
    LIST<GNEURON*> successors;
    ARRAY<GNEURON_OUTPUT> out;
    VECTOR tf_param;
    VECTOR dnetout_dout;
    MATRIX dnetout_dw;
    MATRIX dnetout_dparam;

  public :

    GNEURON();
    GNEURON(int);
    virtual ~GNEURON();
    GNEURON(const GNEURON&);

    virtual void load(ASCII_FILE&);
    virtual void evaluate()=0;
    virtual void calc_sensit();
    virtual void write(ostream&);
    virtual void reset_out_value();

};
```

Figure 1: Generalized Neurons Class

knows its predecessors through their connection names (stored during `load` function execution in `CARRAY<CONNECTION> inputs`).

Once a topology of neurons is specified (it does not have to be layers, althought it can be), all what is needed to calculate the neural network output is an evaluation order. This is specified through ordered "list of neurons". The neurons within the lists, and the lists, are evaluated according to the entry order. An example of neural network and input file is given in Figure 4 and 5, respectively. The network has three layers, and is recurrent because the output neuron points to itself. The commands used in the input file start with one or more "*", according to their level of generality. They are assumed to be sufficiently self-explanatory so that no other explanation is given here.

The gradient of the error with respect to the weights and neuron transfer function parameters (`tf_param`) can be recurcively calculated, in the reverse order of the lists of neurons (see [7]). Sensitivity analysis is performed if the

```
class SIGMOID_NEURON : public GNEURON {
   public :
      double bias;
      SIGMOID_NEURON();
      virtual ~SIGMOID_NEURON();
      virtual void evaluate();
      virtual void load(ASCII_FILE&);
      virtual void write(ostream&);
};
```

Figure 2: Sigmoid Neurons Class

```
class GNEURAL_NET {

   protected :
     int nb_output;
     STRING name;
     PLIST<GNEURON> list;
     PLIST<LIST_OF_NEURONS> masterlist;

   public :
     GNEURAL_NET();
     virtual ~GNEURAL_NET();
     void evaluate( VECTOR & , LIST<VECTOR> & );
     void calc_sensit(LIST<VECTOR> & grad_net_resp);
     ofstream netout;
     const STRING & get_name() const;
     void link();

     friend  GNEURAL_NET* make_neural_net(const char * name);
     friend GNEURAL_NET* make_neural_net(SIMULATION_ENV & env, const VECTOR & x);

};
```

Figure 3: Neural Network Class

**sensitivity_calc command is issued (cf. Figure 5). The cost of sensitivity analysis for the complete network is about the same as the cost of an evaluation. Derivatives of the network output with respect to neuron input weights and transfer function parameters are stored within each neuron in the matrices dnetout_dw and dnetout_dparam. The sensitivity of network outputs to the neuron output is stored in dnetout_dout. Gradient calculation for reccurent networks (such as backpropagation through time [7] or forward propagation [8]) has not yet been implemented.

Further details about this neural network generator (what it can and cannot do) are given in Appendix 8. About 7000 evaluations of a 10 neurons network are performed in 1 CPU second on a Sun4m workstation (cougar). If sensitivity analysis is performed, this figure doubles.

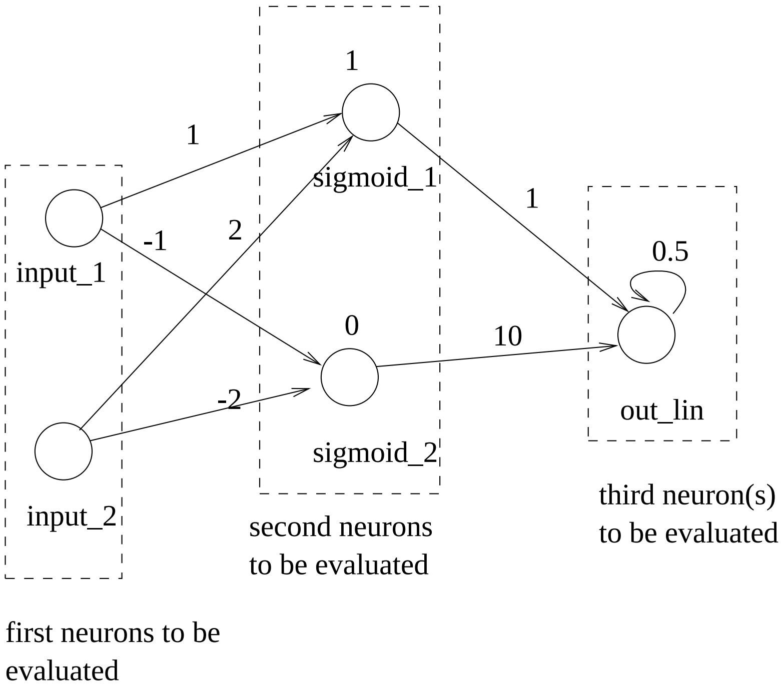

Figure 4: Example of Network

```
**gneuron input
  *name input_1
  *predecessors  0
  *sequence  4
  0.1  0.2  0.3  0.4  0.5
**gneuron input_2
  *name input_2
  *predecessors  0
  *sequence  4
  -0.7  -0.6  -0.5  -0.4
**gneuron sigmoid
  *name sigmoid_1
  *predecessors  2  input_1  1.  input_2  2.
  *bias  2.2
**gneuron sigmoid
  *name sigmoid_2
  *predecessors  2  input_1  -1.  input_2  -2.
  *bias  2.2
**gneuron linear
  *name out_lin
  *predecessors  3  sigmoid_1  1.  sigmoid_2  10.
     out_lin  0.5
  *bias  2.2

**list_of_neurons
  *name inputlayer
  *list 2 input_1  input_2
**list_of_neurons
  *name middlelayer
  *list 2 sigmoid_1  sigmoid_2
**list_of_neurons
  *name outputlayer
  *list 1 out_lin
  **sensitivity_calc
**end
```

Figure 5: Example of Neural Network through its Input File

3 Optimization Tools

3.1 Optimization Methods

The optimization module contains three optimizers for problems with continuous design variables[2], an SQP [11], a simplex [12], and an evolutionary algorithm [13]. This panel of methods encompasses most aspects of optimization algorithms (local versus global, constrained versus unconstrained optimization). On the one hand, there are 2 local, rapidly converging methods, namely SQP and the simplex algorithms. The evolutionary algorithm is, on the other hand, a stochastic method global in scope, i.e., it has a chance of escaping local optima. As such, it offers additional robustness when dealing with nonconvex, multimodal optimization problems.

Among the local optimizers, simplex is a 0 order method (it does not require gradient calculation). It is a robust optimizer, especially for low slope functions. SQP, which stands for sequential quadratic programming, works by sequentially approximating the optimization problem by a quadratic programming problem, which is then solved by ad-hoc methods [14]. It is the only method among those proposed here to explicitly handle optimization constraints [3]. In the version included in the module (CFSQP [11]), quasi-feasible iterates only are generated.

The foundation of the evolutionary algorithm of the module is a standard steady state algorithm, with real numbers encoding, linear crossover operator and gaussian mutation [13]. It presents three additional specificities:

1. The evolutionary optimizer has a pointer to a CFSQP optimizer. Use of the local CFSQP optimizer is possible either through a "CFSQP mutation" operator, or through a final CFSQP refinement of the best point found by the evolutionary search. First of all, CFSQP can be called through a special mutation operator which, instead of randomly perturbing a design point, performs CFSQP iterations on it. The CFSQP mutation operator has two functions: i) it improves the mutated point through SQP type of local optimization, ii) for constrained optimization problems, CFSQP starts by generating a feasible point (the "F" stands for feasible iterates). The CFSQP mutation can therefore be seen as a projection (or repair) operator into the feasible domain[4].

[2]The description is restricted here to continuous design variables eventhough evolutionary optimizers can handle noncontinuous design variables. The reason is that noncontinuous problems require specific implementations, which do not fit the level of generality of this text.

[3]Explicitely means by a strategy other than penalization: if the optimization problem is formulated as $\min_{x \in S} f(x)$ such that $g(x) \leq 0.$, penalization would replace this original constrained problem by an unconstrained problem of the form $\min_{x \in S} f(x) + H(r,x)$, where $H(r,x) = 0$ when $g(x) \leq 0$ and $H(r,x) > 0$ otherwise. $H(r,x)$ is an increasing function of r, $r > 0$.

[4]Many authors have discussed the advantage of blending local and evolutionary optimizations: L. Davis stated [15], "give me your best expert algorithm, and I will improve your best known solution by mixing it with a genetic algorithm", E. Souza de Curci has proposed a stochastic gradient, M. Schoenauer uses local optimizers to fix the values of continuous

```
****optimize optimizer name (sqp, evolution, simplex)
  ***values
      name init. value min max
  ***constraint function of the unknowns
  ***convergence
      parameters of the optimizer
****return
```

Figure 6: Optimizer Interface

2. Constraints are handled through a dynamically adjusting penalty function (r, the vector of penalty parameters, changes according to $dr \propto max(g(\bar{x}), 0.)$ where $\bar{x}$ is the best solution so far in terms of the penalized objective function).

3. The set of the best n_f feasible and n_i infeasible points distant of at least Δx (user specified) are collected during the search This set is a collection of alternative good solutions, different from the final population.

3.2 Interfacing and Programming strategy

The module is based on a generic (pure abstract) class, verb!OPTIMIZER!, which provides methods and interfaces common to all the optimizers, like read, verification and error handling functions. Communication with the optimizers is performed through an input file such as shown in Figure 6. The program can be used to optimize any type of function with any constraints. Objective function and constraints have a fixed prototype. They are given by the user and linked to the rest of the code.

The program is fully integrated with the neural network generator. The user can choose either to run a neural network optimization, through the "-o" option, or a neural network simulation (no tuning of the network coefficients), through the "-s" option. In the event that the "-s" simulation option is taken, no optimizer is loaded in memory (the `OPTIMIZER*` pointer is not affected).

4 Results for Function Interpolation

A series of tests has been performed as a validation for the tools described earlier and as a way of acquiring expertise on teaching the most widespread type of

---

variables in genetic programming, ... A few tests performed for optimizing the weights of feedforward neural networks have not shown a clear advantage at using CFSQP mutation operator, but a thorough study on specially designed function landscapes should be performed for real insight. Fine tuning evolutionary solutions by local optimizers, if not a new idea, makes a lot of sense in practice. A good discussion of repair operators, and more generally constraints handling techniques in evolutionary optimization, can be found in [5]. Note that Orvosh proposed to use repair on about 5.0% of the points generated during evolutionary searches.

networks, feedforward neural networks. The optimizers have been validated on analytical functions. Those tests are not reported here, as they have no other significance than programming validation. Tests on neural networks are listed by increasing difficulty. Gradients of the neural networks were obtained by finite differences in Test 1 to Test 5. Analytical gradient calculated by chain rule derivation is used from Test 6 on.

4.1 Test 1 : weights and biases determination in a highly saturated network

Test 1 is an easy problem where the weights and biases of a FFNN (feedforward neural network) with two sigmoid neurons in the middle layer and one sigmoid neuron in the output layer are sought. What makes the problem easy is that there are few design variables (8), the target response function was obtained using the network with a certain set of weights (so that there is at least one perfect solution to the problem), they were only 20 points to interpolate from, and most of the sigmoid neurons were saturated in the optimal network. Because many neurons were saturated, they did not play any role in the function other than passing a constant to their neighbouring neurons. Many sets of weights and biases cause the network to be saturated in the same way, so many equivalent solutions exist for problem 1. Figure 7 shows the targeted and best neural

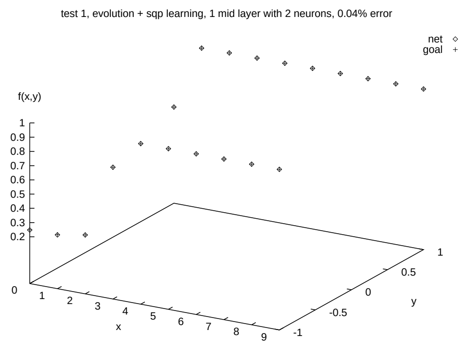

Figure 7: Targeted and simulated response surface for Test 1

response surface obtained. The fit is perfect. Learning is performed using, first, an evolutionary search (population size = 50, crossover prob. = 0.7, mutation prob. = 0.1). Next, the best solution of the evolutionary search is refined using SQP (see Figures 8 and 9 for convergence plots). Eventhough only a test is reported here, other runs have given the same quality of fit. The solutions do not reproduce the optimal set of weights, i.e. equivalent optimal point are located.

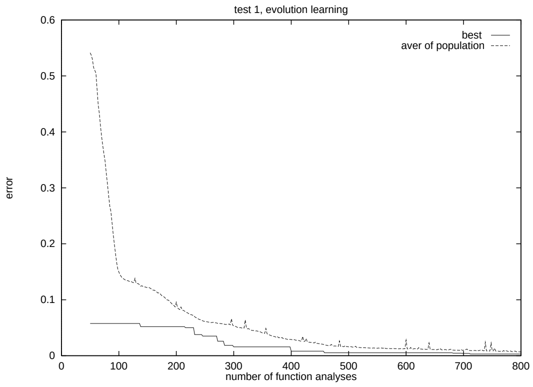

Figure 8: Test 1, evolution convergence

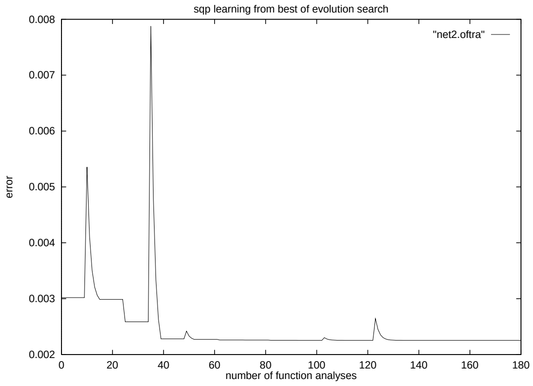

Figure 9: Test 1, SQP convergence on the best of previous evolution search

4.2 Test 2 : weights and biases determination

Test 2 is the same as Test 1, but the weights of the optimal neural network to be found have been changed in order not to have too many neurons saturated (the targeted surface is not as flat as in Test 1). As a result, Test 2 is more difficult than Test 1. In particular, there are local optima on which local optimizers stall. A practical way of deleting most of the local optima is found to be adding interpolation points. After going from 20 to 400 interpolation points, the problem could be solved rapidly (a few thoudands analyses) using SQP.

4.3 Test 3 : interpolation of a 3rd degree polynomial

The purpose of the test is to interpolate the 3rd degree polynomial,

$$f(x, y) \; = \; 5x^3 - 2y^3 + 1.3x^2y - 0.4xy^2 - 7x^2 - y^2 + 4xy + 3x - 9y + 12,$$

for x and $y \in [0, 10]$. Initial experiments have shown that normalization of design variables and function greatly enhanced the reliability of SQP optimization. This should be linked to better conditionning of the problem and more accurate finite differences. A plot of the normalized target surface is given in Figure 10. In addition, normalization enables tighter bounds on the design variables, which significantly reduces the size of the design space for evolutionary sampling.

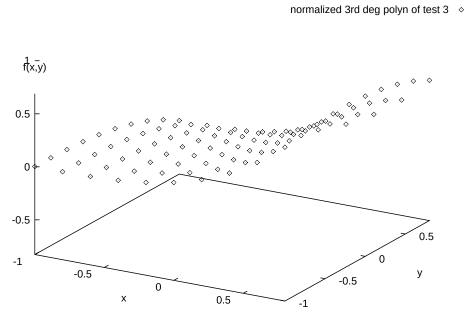

Figure 10: Test 3, normalized target surface

Tests were performed on a 1 hidden layer FFNN with 5 hidden neurons. Learning the FFNN is an optimization problems with 21 unknowns. 100 uniformly distributed points were given to sample the function. Three SQP runs were started from 3 different points. All produced network approximations below 5.0% error (1.0%, 2.3 % and 4.6%, respectively) within 2000 evaluations of the objective function (which represents 100 neural network simulations since there are 100 sample points). A typical SQP convergence is shown in Figure 11. Comparison between the target surface and the best FFNN approximation is given in Figure 12.

Evolutionary optimizer does not represent a significant advantage over local optimizers for this problem as it finds 5 to 10.0% error in 2000 evaluations. Further evolutionary improvement of the network is extremely slow (only a few percent are typically gained by continuing the search to 200000 analyses).

4.4 Test 4 : interpolation of a function with discontinuous derivatives

Test 4 concerns the interpolation of the function,

$$f(x, y) \; = \; \frac{|5x^3 - 2y^3 + 1.3x^2y - 0.4xy^2 - 7x^2 - y^2 + 4xy + 3x - 9y| \; - \; 15}{15},$$

$x,\, y \in \; [-1, 1]$. Because of the absolute value, $f(x, y)$ has a discontinuous first derivative. 441 points are used to sample the function.

Figure 11: SQP convergence on test 3, 21 design variables

Three FFNN have been compared: two 1 hidden layer networks with 3 and 7 hidden neurons, respectively, and a 2 hidden layers network, with 3 and 4 neurons respectively (for the same total of 7 hidden neurons). Those networks have 13, 29 and 30 variables, respectively.

Three SQP training have been performed for the 3 neurons FFNN. They have resulted in errors of 3.2%, 8.0%, and 6.0%, respectively, within 2000 objective function evaluations. A typical SQP convergence for this problem is shown in Figure 13. A comparison between target surface and best FFNN surface is given in Figures 14 and 15. Figure 15 results from the slicing of Figure 14. It is seen in Figure 15 that most of the error occurs at the jump in first derivative. This is due to the relatively low number of points at the change in derivative (i.e. their low overall weight in the total error function). By increasing the weight of the sample points where the discontinuity in first derivative occurs, one could obtain a better fit in that region, at the expense, however, of a degraded overall approximation. This is no longer true when one increases the number of neurons. An evolutionary optimization (population size 400, crossover prob = 0.7, mutation prob = 0.1, 200000 analyses) produced an 11% error network, which was subsequently improved to a 7% error network after SQP iterations.

Three SQP trainings have been performed on the 1 hidden layer, 7 neurons, FFNN. They have produced 2.3%, 1.9%, and 2.6% error networks for runs of

Figure 12: Test 3, Comparison between the target surface and the best FFNN surface

20000 to 40000 analyses. Comparison between target and 1.9% error FFNN is given in Figure 16. Evolutionary search resulted in a 6.0% error network. The error of this network was decreased to 2.8% after final SQP refinement.

Three SQP learnings with 2 hidden layers neural network resulted in 1.8%, 0.9%, and 5.4% errors. With respect to the 1 hidden layer network with the same number of neurons (7), the two layers network exhibits better performances, but is more difficult to train. Initial convergence below 10.0% error is slower with the 2 layers FFNN than it is with the 1 layer FFNN. Whereas better 2 layers networks have been determined, training of 2 hidden layers networks is more susceptible of stopping at a local optimum than training of 1 hidden layer networks. This is due to the higher linearity of single hidden layer FFNN than two hidden layers FFNN (with the same number of neurons). Single layer networks have more connections to the output neurons. The weights of those connections play a linear (or quadratic, depending on the formulation) role in the total error. Those weights are therefore much easier to adjust than weights between previous layers. This is one of the incentive for using radial basis function networks. A comparison between the target and the best two layers FFNN is shown in Figure 17.

Figure 13: Test 4, SQP convergence with three neurons in one hidden layer

4.5 Test 5 : interpolation of a discontinous function

The last validation test concerned a discontinuous function. The function to interpolate was,

$$x_1 = x_0 + 4\, floor(2x_0),$$

$$f(x_1, x_2) = \frac{5x_1^3 - 8x_2^3}{27}$$

where x_0 and x_2 vary between -1 and 1, and $floor(x)$ returns the largest integer smaller than x. A graph of the function is given in Figure 18. Three tests performed on a 2 hidden layers (3 and 4 neurons in the first and second hidden layers, respectively) network resulted in 3.2%, 3.3% and 4.1% error, respectively. The search length was of the same order as for the Test 4. Comparison between the target function and its neural approximation after learning is shown in Figure 19. This example shows that FFNN have the ability to accurately interpolate discontinuous functions. The jumps in objective function are produced by neurons in the last hidden layer that fire abruptely. This observation implies that the number of neurons in the last layer should be, in a first approximation, proportional to the expected number of discontinuities in the target function. A comparison of SQP learning when error gradient is calculated by forward difference and analytically is shown in Figure 20. One evaluation of the network (x-axis) includes one evaluation for all 361 points of

Figure 14: Test 4, comparison between target and FFNN (3 neurons in one hidden layer) surfaces, 3.2% error

the training set. Eventhough one evaluation with analytical gradient is twice as long as an evaluation with forward differences gradient (because there is a forward and a backward neurons calculation), analytical gradient is more than twice faster than forward difference gradient. The advantage of analytical gradient will become more visible as the size of the network increases. It should be noted that, even if approximate gradients yield to a slower convergence, the network it converges to is the same.

4.6 Test 6 : interpolation dimensionally equivalent to the Flight Controller

The purpose of this test is to train a neural network that should perform a mapping of the same dimension as the flight controller does. The flight controller will typically take the (4) state variables, the current airplane position (2 variables) as inputs, and output the control surfaces deflections (4 typically). Test 6 is thus the following 6 inputs, 4 outputs nonlinear function:

$$f_1(\underline{\mathbf{x}}) \;=\; \cos(\frac{2\pi}{6}\sum_{i=1}^{6} x_i) \qquad (1)$$

Figure 15: Test 4, comparison between target and FFNN (3 neurons in one hidden layer) surfaces, 3.2% error

$$
\begin{aligned}
f_2(\underline{\mathbf{x}}) &= \cos(\frac{2\pi}{6}\sum_{i=1}^{6} x_i + \frac{\pi}{8}) \qquad (2) \\
f_3(\underline{\mathbf{x}}) &= \cos(\frac{2\pi}{6}\sum_{i=1}^{6} x_i + \frac{2\pi}{8}) \qquad (3) \\
f_3(\underline{\mathbf{x}}) &= \mathrm{sign}(x_1 + x_3 - 1) \qquad (4)
\end{aligned}
$$

The function $\underline{\mathbf{f}}$ is a (very!!) rough approximation of what the controller might look like.

To represent this function, a feedforward network with two hidden layers is chosen. There are 10 neurons in each hidden layer. The problem has 214 parameters to tune.

Three experiments have been performed using SQP, analytical gradient calculation, and random initial parameters between -1 and 1. The training set has 729 randomly located points. Interpolations with 0.09%, 2.31% and 5.93% error were obtained, respectively. On the average, 500 network evaluations (of the complete training set) are necessary to obtain 5% error or less. An example of SQP convergence on this problem is shown in Figure 21.

Figure 16: Test 4, comparison between target and 1 hidden layers FFNN, 7 hidden neurons, 1.9% error

5 Partial conclusions : how to interpolate multivariate functions using feedforward neural networks

Practical results for interpolating multivariate functions are given here. We do not discuss generalization properties of the trained networks (i.e. the ability of the networks to handle points that were not included in the training set)[5].

5.1 Avoiding Unnecessary Local Optima

The existence of local optima when training neural networks has been long aknowledged (see for example [17]). This has been a reason for trying stochastic optimizers such as evolutionary algorithms. However, the previous tests have shown that gradient based methods had a definite advantage over a general evolutionary algorithm for tuning weights and biases of feedforward neural networks. Other studies have noticed the relative inefficiency of generic evolu-

[5] Strategies to evaluate the generalization properties of a network involve setting aside some points that are not used for training. The purpose of these points is to quantify the generalization ability of the network.

Figure 17: Test 4, comparison between target and 2 hidden layers FFNN, 0.9% error

tionary algorithms for tuning neural networks[6] [18]. The efficiency advantage of optimizers such as SQP is reinforced by the availability of analytical gradients through chain derivations. However, typical scenarios where gradient based search will fail have been identified. They are analyzed next.

5.1.1 Symmetrical Networks

The following discussion arose from unsuccessful network trainings experienced by seeding all weights and biases initially to 1.

When training a network using a gradient based algorithm, the initial set of weights and biases should not be symmetrical, i.e., two or more neurons with the same predecessors and successors should not have identical weights and biases. Indeed, when two or more identical neurons are redundant, and have exactly the same gradient with respect to the network error. The gradient will never differentiate them, eventhough the optimal network is usually not symmetrical.

This is, in fact, a feature of all symmetrical functions.

[6] This might not be true for specialized evolutionary algorithms

Figure 18: Test function 5

Theorem 1 : *The gradient of a symmetrical function is along its plane of symmetry.*

proof :

Let f be a symmetrical function about the plane $\underline{\mathbf{D}}^T\underline{\mathbf{x}} = 0$. Then $f(\underline{\mathbf{x}}) = f(\underline{\mathbf{x}}_s)$, where $\underline{\mathbf{x}}_s$ is the symmetric of $\underline{\mathbf{x}}$ obtained by $\underline{\mathbf{x}}_\mathbf{s} = (1 - 2\underline{\mathbf{D}}\underline{\mathbf{D}}^T)\underline{\mathbf{x}}$, and $\underline{\mathbf{D}}$ is a unit vector.

Consider $\underline{\mathbf{x}}_\mathbf{0}$, a point on the symmetry plane. We show that necessarily $\underline{\mathbf{D}}^T \underline{\mathrm{grad}} f(\underline{\mathbf{x}}_\mathbf{0}) = 0$, which means that the gradient of f on the plane of symmetry is along the plane of symmetry. Let us take a point $\underline{\mathbf{x}}$ along the gradient line at $\underline{\mathbf{x}}_\mathbf{0}$, $\underline{\mathbf{x}} = \underline{\mathbf{x}}_\mathbf{0} + \alpha\ \underline{\mathrm{grad}} f(\underline{\mathbf{x}}_\mathbf{0})$. For an arbitrarily small α, $\underline{\mathbf{x}}$, $\underline{\mathbf{x}}_\mathbf{s}$ and $\underline{\mathbf{x}}_\mathbf{0}$ are arbitrarily close, therefore $f(\underline{\mathbf{x}})$ and $f(\underline{\mathbf{x}}_\mathbf{s})$ can be approximated by their first order Taylor expansion about $\underline{\mathbf{x}}_\mathbf{0}$.

$$f(\underline{\mathbf{x}}) = f(\underline{\mathbf{x}}_\mathbf{0}) + (\underline{\mathbf{x}} - \underline{\mathbf{x}}_\mathbf{0})^T \underline{\mathrm{grad}} f(\underline{\mathbf{x}}_\mathbf{0}) \qquad (5)$$

$$f(\underline{\mathbf{x}}) = f(\underline{\mathbf{x}}_\mathbf{0}) + \alpha\ \underline{\mathrm{grad}}^T f(\underline{\mathbf{x}}_\mathbf{0}) \underline{\mathrm{grad}} f(\underline{\mathbf{x}}_\mathbf{0}) \ . \qquad (6)$$

$$f(\underline{\mathbf{x}}_\mathbf{s}) = f(\underline{\mathbf{x}}_\mathbf{0}) + (\underline{\mathbf{x}}_\mathbf{s} - \underline{\mathbf{x}}_\mathbf{0})^T \underline{\mathrm{grad}} f(\underline{\mathbf{x}}_\mathbf{0}) \ . \qquad (7)$$

Figure 19: Test 5, comparison between target function and its neural approximation

But,

$$\underline{\mathbf{x}}_{\mathbf{s}} \;=\; \underline{\mathbf{x}} \;-\; 2\underline{\mathbf{D}}\underline{\mathbf{D}}^T\underline{\mathbf{x}}\;, \tag{8}$$

$$\underline{\mathbf{x}}_{\mathbf{s}} \;=\; \underline{\mathbf{x}}_{\mathbf{0}} \;+\; \alpha\;\underline{\mathrm{grad}}f(\underline{\mathbf{x}}_{\mathbf{0}}) \;-\; 2\alpha\;\underline{\mathbf{D}}\underline{\mathbf{D}}^T\underline{\mathrm{grad}}f(\underline{\mathbf{x}}_{\mathbf{0}})\;. \tag{9}$$

Substituting $(\underline{\mathbf{x}}_s - \underline{\mathbf{x}}_{\mathbf{0}})$ from equation (9) to equation (7) yields,

$$\begin{aligned} f(\underline{\mathbf{x}}_s) &= f(\underline{\mathbf{x}}_{\mathbf{0}}) \;+\; [\alpha\;\underline{\mathrm{grad}}^T f(\underline{\mathbf{x}}_{\mathbf{0}}) - 2\alpha\;\underline{\mathrm{grad}}^T f(\underline{\mathbf{x}}_{\mathbf{0}})\underline{\mathbf{D}}\underline{\mathbf{D}}^T]\underline{\mathrm{grad}}f(\underline{\mathbf{x}}_{\mathbf{0}}) \quad (10)\\ &= f(\underline{\mathbf{x}}) \;-\; 2\alpha\;\underline{\mathrm{grad}}^T f(\underline{\mathbf{x}}_{\mathbf{0}})\underline{\mathbf{D}}\underline{\mathbf{D}}^T\underline{\mathrm{grad}}f(\underline{\mathbf{x}}_{\mathbf{0}})\;. \quad (11) \end{aligned}$$

Since the function is symmetrical, $f(\underline{\mathbf{x}}) \;=\; f(\underline{\mathbf{x}}_s)$, the last term of equation (11) must cancel out, therefore,

$$\underline{\mathbf{D}}^T\underline{\mathrm{grad}}f(\underline{\mathbf{x}}_0) \;=\; 0\;. \tag{12}$$

$\square$.

Theorem 1 implies that when a gradient based search is started on the plane of symmetry of a function, the search will never leave this plane. It might, depending on the function (its Hessian), converge to a local or a global optimum, or not converge.

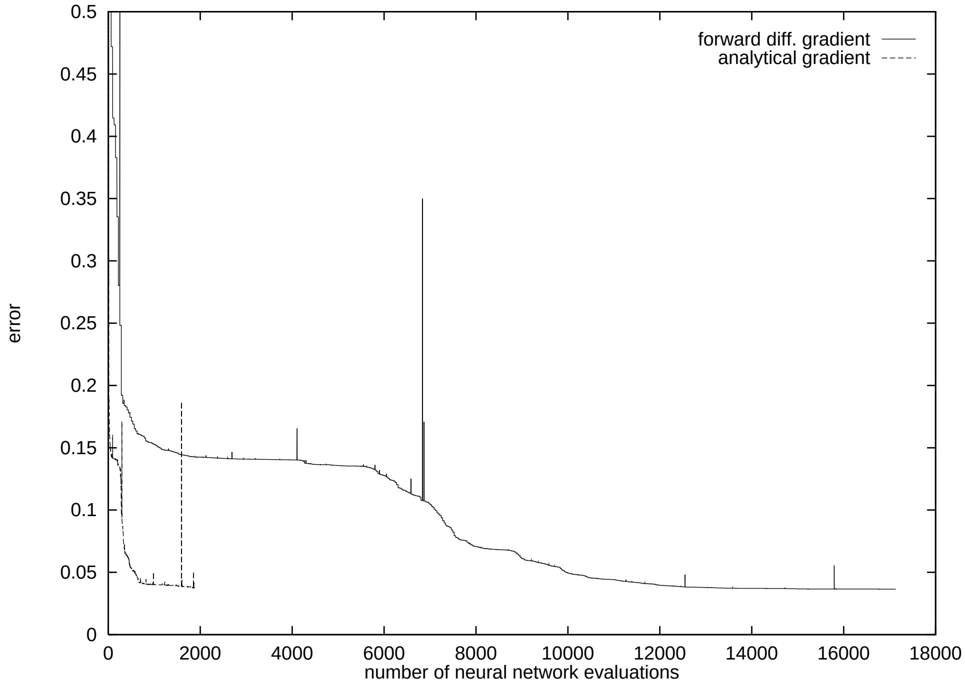

Figure 20: Test 5, SQP convergence with an approximate and an exact gradient calculation

5.1.2 Ridge Networks Sensitivity to Plateaus

When learning networks on Test 6 (multidimensional mapping with periodic functions), the first three attempts converged to 53%, 55% and 56% error. In all cases, the response of the trained network was characterized by perfect fitting of some of the data points, and saturation of the entire network (the network outputs were not sensitive to the network inputs). This is seen to be a combined result of the periodic functions to be fitted, and of a regularly spaced grid of data points. The period of the functions to be fitted is an integer multiple of the data points period. With such coinciding periods, many training points have the same output. Not knowing the functions to be interpolated, it looks as if there are many plateaus (cf. Figure 22).

We next explain why ridge networks (i.e., with sigmoid neurons) are prone to be saturated when fitting functions with plateaus. Let consider a least square fit of the function $f(x)$ using the approximate function $f_a(x,p)$. For the sake of simplicity, we take only two points x_1 and x_2 in the least square fit and a single parameter p to tune f_a. We are therefore minimizing the error $E(p)$,

$$E(p) \;=\; \frac{1}{2}(f_a(x_1,p) - f(x_1))^2 \;+\; \frac{1}{2}(f_a(x_2,p) - f(x_2))^2 \; . \qquad (13)$$

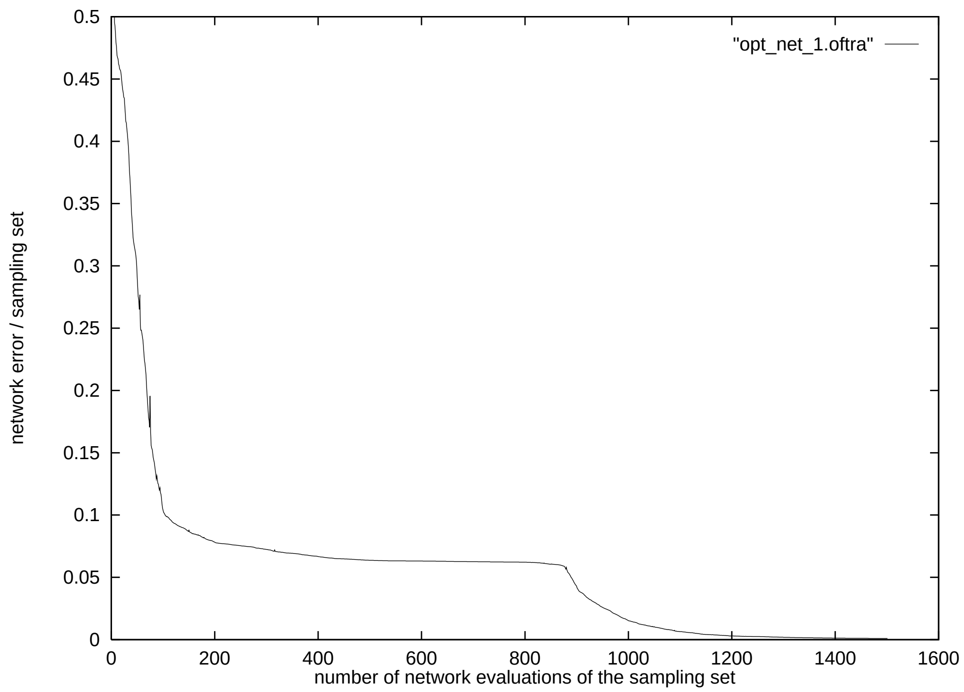

Figure 21: Test 6, SQP convergence with 6x10x10x4 feedforward network, analytical gradient

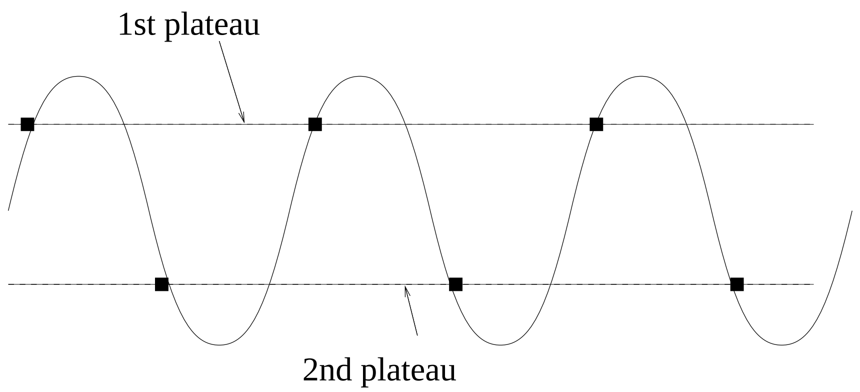

Figure 22: Apparition of plateaus due to regular sampling

The necessary first order optimality condition for E is,

$$\frac{\partial E(p)}{\partial p} \;=\; (f_a(x_1,p) - f(x_1))\frac{\partial f_a(x_1,p)}{\partial p} \;+\; (f_a(x_2,p) - f(x_2))\frac{\partial f_a(x_2,p)}{\partial p} \;=\; 0\;. \tag{14}$$

For well behaved least square problems, necessary optimality conditions are satisfied at $p = p^*$ when $f_a(x_1, p^*) = f(x_1)$ and $f_a(x_2, p^*) = f(x_2)$. There is however another scenario with sigmoid functions that is very likely to happen in the presence of plateaus: one of the points (say x_1) is almost perfectly fitted $(f_a(x_1,p) - f(x_1) \approx 0)$ with $(\partial f_a(x_1,p))/(\partial p) > 0$, while the other point is not fitted but the approximation function has no sensitivity to p there, i.e., $(\partial f_a(x_2,p))/(\partial p) \approx 0$. Figure 23 illustrates the case with a single neuron. In such a case, the necessary first order optimality conditions (equation (14)) are almost satisfied. The term $(f_a(x_1,p) - f(x_1))\frac{\partial f_a(x_1,p)}{\partial p}$ is dominant in the equation, i.e. it will attract any point in the region. This is a case where the chances of converging to a local optimum are large[7]. Test 6, with regularly spaced data is a good example of this phenomenon. The large number of neurons and of plateaus seem to multiply the number of local minima, so that there is a negligible probability of escaping local minima.

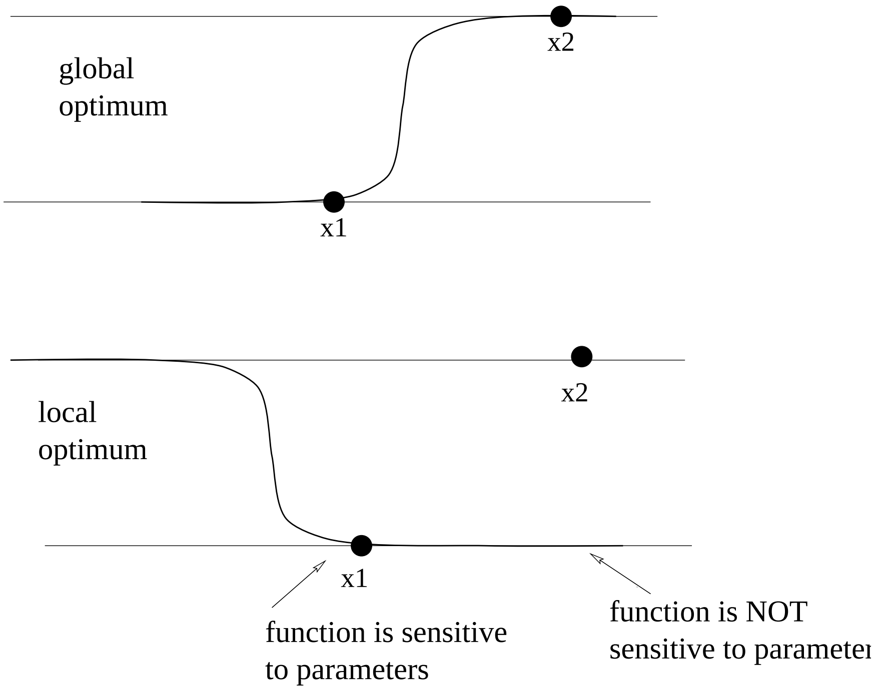

Figure 23: Local attractor with sigmoid neurons

The local attraction problem of Test 6 was solved by using randomly distributed input points. It has also been found advantageous to seed initial weights

[7]This observation fosters neural network specific mutation operator in evolutionary calculation. One could design a specific mutation operator to escape this type of local optimum. Such a mutation could work, for example, by changing the sign of all the links connected to a given neuron, which inverses the slope of the sigmoid function.

and biases with random numbers of small magnitude (between -1 and 1) because it prevents symmetry (randomness) and it is less likely to have saturated neurons (small magnitudes).

5.2 How many layers ?

There are arguments in favor of using two hidden layers instead of one. In [16], it is shown (on a particular "pinacle" function) that one hidden layer networks need a larger number of neurons to be accurate than two hidden layers networks. It is also shown that two layers networks have better generalization properties. This seems to be intuitively correct as one might expect that one layer acts as a classifier, while the other layer performs finer interpolation work. As discussed in section 4.4, however, there is a price for these properties as two layers networks seem to be harder to train. The learning optimization problem is less linear because there are less neurons directly connected to the output layer.

6 Flight Simulator

6.1 General Description

A longitudinal simulation of a wing with control surfaces has been implemented. Integration of the equations of motion is performed using a fifth order Runge Kutta integration with error estimation and automatic time stepping [19]. In order to anticipate an eventual on-line change in number of control surfaces, control surfaces are described by dynamic variables. Three models of atmospheric disturbance have been implemented and are described in the following section 6.2. Once again, the structure of the program makes it easy to add new wind disturbance models without interfering with the rest of the program.

An example of input file for the flight simulator is given in Figure 24. The simulation is divided into a controller and a wing part. The controller consists of a description of the control inputs, which is here the time history of the control surface deflections. They are described by tables, with linear interpolation between data. The wing data is composed of inertial and geometric terms, and aerodynamic sensitivity derivatives [8]. Data related to the integration of the equations of motion through time (initial state, flight length, integration method and its parameters) are given after in the input file. Note that the number of steps, `nb_steps`, controls the number of outputs during the run, and represents a minimum number of equally spaced steps taken by the integration routine. The number of local steps, `nb_local_steps`, is a further subdivision of the global steps, when no automatic time stepping is used. When automatic time stepping is set on (option `auto_time_step`), `nb_local_steps` is an indication of the step size at the first increment (the size of local steps is then controlled by the integration error). Input to the flight simulator is also a descrition of the atmosphere. This is the subject of the next section.

[8] from Ben Tigner

```
***controller
  **ctrl_surf cs1
    *time  0.  201.
    *value  0.  0.
  **ctrl_surf cs2
    *time  0.  10. 201.
    *value  0. 0.  0.
  **ctrl_surf cs3
    *time   0.  50.   52.  70.  72.  201.
    *value 0.  0.     5.  5.   0.    0.
  **ctrl_surf cs4
    *time  0. 99.  100.   1000.
    *value  0.  0.  0.   0.
***end

***wing
  surface  100.  chord  3.5355  weight  250.  mass  7.764
  I_yy  37.0455  cl_alpha  0.079  cl_q  0.0991  cl_0  -0.2715
  cm_alpha  -0.0122  cm_q  -0.0616  cm_0  0.1033
  cd_0  0.0088  cd_1  -0.008  cd_2  0.0441

  *control_surface  4
    cl_cs 0.0161  0.0127 0.0087 0.
    cm_cs  -9.12e-4 -0.006 -0.0082 6.2945e-4
    cd_cs  8.e-6 -3.04e-4 -5.e-4 0.0012

  flight_length  200.
  state_start  140.0  10.0  0.0  0.0
  *integration  rk5
  nb_steps  1000
  nb_local_steps  1
  auto_time_step
  precision  0.05

  *u_disturbance  gust_discrete
   magnitude 10.
   wavelength  1000.
   length 2000.
   start_position  4000.

   *w_disturbance  turbulence  karman
     wavelength_min   1000.
     wavelength_max   5000.
     nb_terms  5
     rms_intensity  10.
     scale_length  2500.

***end
```

Figure 24: Example of input file for the flight simulator

6.2 Gust and Turbulence Models

Three atmospheric disturbance models have been implemented. One is a simple cosine wave[9] (key word `gust_wave`), the other is a discrete gust model (`gust_discrete`), and the last is a random harmonic turbulence that follows a given power spectral density function (`turbulence karman`). All three models can be used independently to perturb the horizontal and vertical wind components (instructions `*u_disturbance` and `*w_disturbance`). An example of atmospheric disturbance is shown in Figure 24.

In the wind disturbance models, the following assumptions are underlying :

- Taylor's hypothesis is made, i.e., the disturbance field is frozen with respect to time. The change in wind condition is only due to the motion of the airplane. The time frequency ω (rad/sec), spacial frequency Ω (rad/ft), and airplane horizontal speed u (ft/sec) are linked through, $\omega = \Omega\, u$. For hovering flight, this assumption does not stand.
- A point like model of the airplane is used, therefore wind variation along the length of the aircraft is neglected, $q_d = \partial w_d/\partial x = \dot{w_d}/u \approx 0$.

6.2.1 Wave Gusts

Wave gusts are aerodynamic perturbations of the form $A[1 - cos(\frac{2\pi}{\lambda}x)]$, where x is the horizontal position of the wing, A is an amplitude factor and λ a turbulence wave length. Description of the wave gust is given to the program in the following way,

```
*u_disturbance  gust_wave
 magnitude 5.
 wavelength  1000.
```

6.2.2 Discrete Gusts

A discrete gust model with a "1 - cosine" shape such as shown in Figure 25 has been implemented. It is specified through its `magnitude`, `wavelength`, `length` and `start_position` (cf. Figure 24). The discrete gust is in compliance with military specifications as stated in [20], with the additional possibility of a limited length gust.

6.2.3 Turbulence Model

Wind turbulence are modelled by harmonic random processes (see [21] for a more general presentation). In addition to the general assumptions for wind disturbances, it is supposed that,

- atmospheric turbulence is a stationary process, i.e., its statistical properties are independent of time.

[9] The wave gust, contrarily to the two other types of disturbances, has no physical foundation, but it is convenient to test the reaction of the aircraft to arbitrary cosine signals.

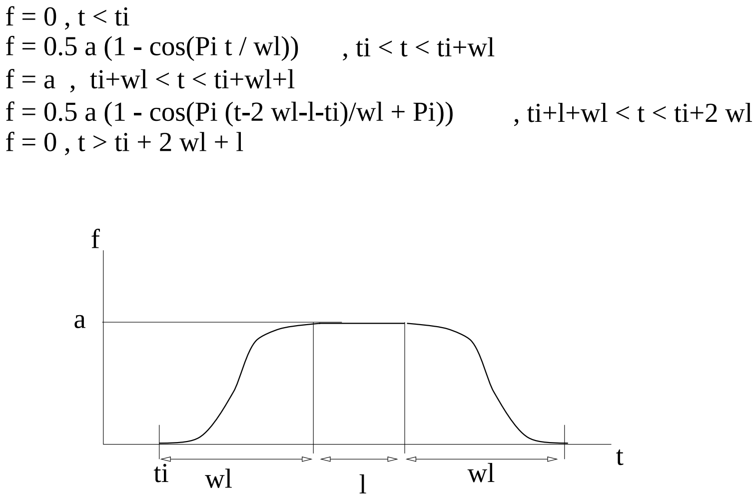

Figure 25: Discrete Gust Model

- the horizontal and vertical components of turbulence are independent random variables.

Note that, eventhough real turbulence is an ergodic process, Taylor's hypothesis of a frozen field makes the model non ergodic.

The turbulence model implemented is specified through a (two-sided) von Karman power spectral density function [20] :

$$\Phi_u = \frac{\sigma_u^2 2 L_u}{\pi(1+(1.339 L_u \Omega)^2)^{5/6}} \tag{15}$$

$$\Phi_w = \frac{\sigma_w^2 L_w (1+\frac{8}{3}(1.339 L_w \Omega)^2)}{\pi(1+(1.339 L_w \Omega)^2)^{11/6}} \tag{16}$$

where $L_u = L_w = L = 2500.\ ft$ is the turbulence scale length, $\sigma_u = \sigma_w$ is the root mean square intensity of the turbulence, and Ω is the spacial frequency of the harmonic of the turbulence considered. The turbulence $d(x)$ is approximated by a sum of sine functions with a random phase and a given amplitude. The amplitude is calculated so that the final turbulence follows the power spectral density $\Phi(\Omega)$. Such a model was first presented in [22]. The principle of the model and its proof are now described.

The turbulence is approximated by the sum,

$$d(x) = \sum_{i=1}^{m} A_i \sin(\Omega_i x + \phi_i) = \sum_{i=1}^{m} d_i(x) \ , \tag{17}$$

where the set of frequencies Ω_i are equally spaced in an interval of interest $[\Omega_l, \Omega_u]$ [10]. The phases ϕ_i are independent random variables uniformly dis-

[10] The frequencies of interest are dictated by the structural modes of the airplane and its speed.

tributed in $[0, 2\pi]$.

The autocorrelation function of d is defined as,

$$\begin{aligned} R_d(\tau) &= \lim_{T\to\infty} \tfrac{1}{T} \int_{-T/2}^{T/2} d(x)d(x+\tau)dx \\ &= \lim_{T\to\infty} \tfrac{1}{T} \int_{-T/2}^{T/2} \textstyle\sum_i d_i(x) \sum_j d_j(x)dx \\ &= \lim_{T\to\infty} \tfrac{1}{T} \int_{-T/2}^{T/2} \textstyle\sum_i d_i(x)d_i(x)dx \end{aligned} \tag{18}$$

where the last equality comes from the fact that $d_i(x)$ and $d_j(x)$ are independent random processes when $i \neq j$, i.e.,

$$\int_{-\infty}^{+\infty} d_i(x)d_j(x)dx \;=\; 0 \quad \text{when } i \neq j \; . \tag{19}$$

Inversing sum and integration in (19), $R_d(\tau)$ becomes,

$$R_d(\tau) \;=\; \sum_i \lim_{T\to\infty} \frac{1}{T} \int_{-T/2}^{T/2} d_i(x)d_i(x)dx \; . \tag{20}$$

One term of the sum represents the autocorrelation function of $d_i(x) = \sin(\Omega_i x + \phi_i)$, which is classically calculated as,

$$R_{d_i}(\tau) \;=\; \lim_{T\to\infty} \frac{1}{T} \int_{-T/2}^{T/2} \sin(\Omega_i x + \phi_i)\sin(\Omega_i(x+\tau)+\phi_i)dx \;=\; \frac{A_i^2}{2}\cos(\Omega_i \tau) \; . \tag{21}$$

Notice that the autocorrelation function $R_{d_i}(\tau)$ is independent of the phase angle ϕ_i. Now, the expected mean square of $d(x)$ is, by definition of the autocorrelation function and the spectral density function,

$$E[d^2(x)] \;=\; R_d(0) \;=\; \int_{-\infty}^{+\infty} \Phi(\Omega)d\Omega \;=\; \sum_i \frac{A_i^2}{2} \; . \tag{22}$$

If we now approximate the integral of the power spectral density function by a sum, we obtain,

$$\sum_i \Phi(\Omega_i)\Delta\Omega_i \;=\; \sum_i \frac{A_i^2}{2} \; . \tag{23}$$

Therefore,

$$A_i \;=\; \sqrt{2\Phi(\Omega_i)\Delta\Omega_i} \; . \tag{24}$$

The turbulence is modelled by $d(x)$, which is consistent the power spectral density Φ, where,

$$d(x) \;=\; \sum_{i=1}^{m} \sqrt{2\Phi(\Omega_i)\Delta\Omega_i}\,\sin(\Omega_i x + \phi) \; . \tag{25}$$

Note that in Shinozuka's method, the sine frequencies are randomly perturbed in order to avoid periodicity of the generated function. This was found, however, to be costly when integrating the equations of motion of the aircraft, and since it is not thought to be a necessary feature of the simulation, the small perturbations have been removed. An example of a signal generated by the method when $m = 5$, $\Omega_l = 1000.\ ft$, $\Omega_u = 5000.\ ft$, $\sigma = 10.\ ft/sec$ and $L = 2500.\ ft$ is given in Figure 27.

6.3 Tests

As an illustration, sample runs of the flight simulator are shown in Figures 26 to 29. The case corresponds to the input file of Figure 24. A time history of the third control surface (the only deflected here) is shown in Figure 26 with different combinations in the wind perturbations. The trajectories and angle of attack histories for a quiet flight, a flight with a discrete gust, and a flight with turbulence are compared in Figures 28 and 29, respectively. The intensities of the wind perturbations correspond to light to moderate cases according to [20]. Such a 200 sec. flight simulation takes 0.5 CPU sec. on a SUN alpha workstation.

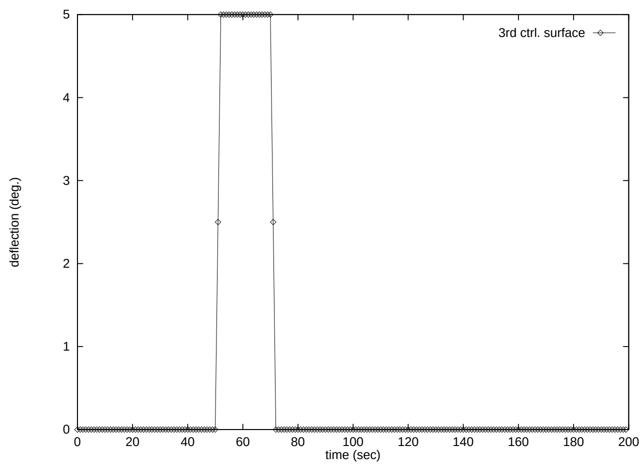

Figure 26: Time history of the 3rd Control Surface Deflection

7 Concluding Remarks

General C++ numerical tools have been implemented to create any type of neural network. Three principal types of function optimizers have been included, namely SQP, simplex and evolutionary methods. Experiments have validated the efficiency of the generator for C^0 and C^1 multivariate function interpolation. Some of the failure modes when training ridge feedforward neural networks have been analyzed.

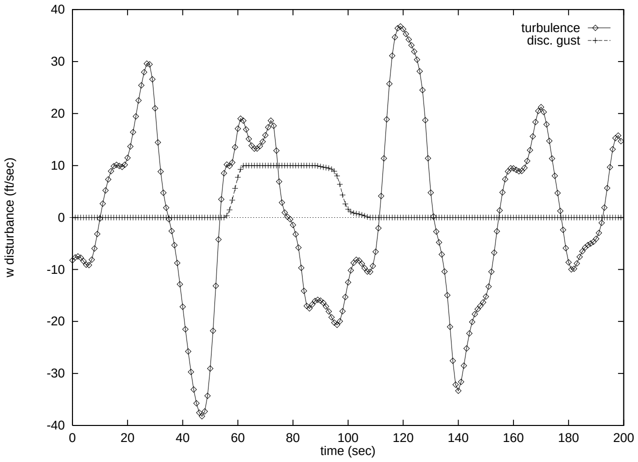

Figure 27: Discrete Gust and Turbulence Models

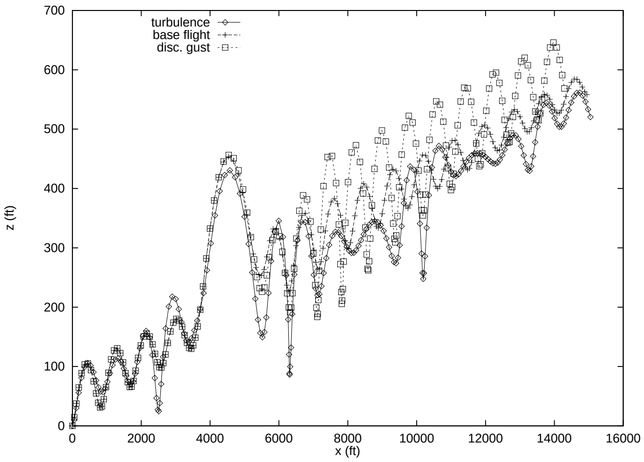

Figure 28: Comparison of trajectories, same controls, quiet and disturbed atmospheres

A flight simulator has been developped in a second part of the program. It includes the integration of the longitudinal nonlinear equations of motion of an airplane. Models for discrete wind gust and harmonic random turbulences have been included.

The next steps of this work will be,

1. implementing a neural network controller for trajectory control of an airplane in a quiet atmosphere. This part of the project involves determining the best neural network approach (structure of the network and training

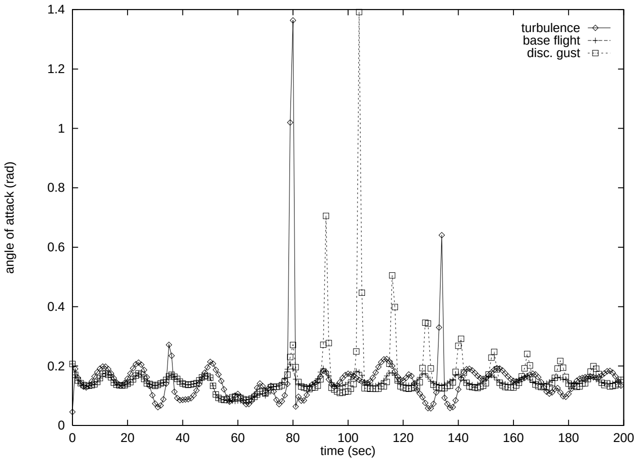

Figure 29: Comparison of angles of attack in quiet and disturbed atmosphere

method).

2. controlling an airplane trajectory in a turbulent atmosphere with a neural network. This part of the study will address the impact of random wind gusts on the optimal neural network structure and the optimal training method.

References

[1] Kůrková, V., "Kolmogorov's Theorem and Multilayer Neural Networks", *Neural Networks*, 5, 1992, pp. 501 - 506.

[2] Barron, A., "Universal Approximation Bounds for Superposition of a Sigmoidal Function", Technical Report 58, Statistics Dept., Univ. of Illinois at Urbana-Champaign, USA, 1991.

[3] *Neural Network Applications in Control*, G.W. Irwin, K. Warwick and K.J. Hunt, eds., IEE Control Eng. Series 53, 1995.

[4] Dracopoulos, D.C., "Genetic Algorithms and Genetic Programming for Control", in *Evolutionary Algorithms in Eng. Applications*, D. Dasgupta and Z. Michalewicz, eds., Springer Verlag, 1997, pp. 329 - 343.

[5] Michalewicz, Z., Le Riche, R., Dasgupta, D., and Schoenauer, M., *Evolutionary Algorithms for Constrained Engineering Problems*, Computers & Industrial Engineering Journal, 30(4), 1996, pp. 851-869.

[6] Shaffer, J.D., Whitley, D. and Eshelman, L.J., "Combination of Genetic Algorithms and Neural Networks: A Survey of the State of the Art", *Intl.*

Worshop on Combinations of Genetic Algorithms and Neural Networks, Baltimore, USA, June 6, 1992, pp. 1 - 37.

[7] Rumelhart, D.E. and McClelland, J.L., *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, vol. 1: Foundations, MIT Press, Cambridge, Mass., 1986.

[8] Williams, R.J., and Zipser, D., "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks", *Neural Computation*, 1, 1989, pp. 270-280.

[9] Stuttgart Neural Network Simulator, University of Stuttgart, Institute for Parallel and Distributed High Performance Systems (IPVR), Breitwiesenstr. 20-22, D-70565 Stuttgart, Fed. Rep. of Germany, available by ananymous ftp from ftp.informatik.uni-stuttgart.de (129.69.211.2), 1995.

[10] Besson, J. and Foerch, R., "Large Scale Object Oriented Finite Element Code Design", *Computer Methods in Applied Mechanics and Engineering*, 142, pp. 165 - 187, 1997.

[11] C. Lawrence, J.L. Zhou and A.L. Tits, *User's Guide for CFSQP Version 2.5*, Inst. for System Research TR-94-16r1, Univ. of Maryland, College Park, MD 20742, USA, 1997.

[12] Nelder, J.A. and Mead R., "A Simplex Method for Function Minimization", *Computer Journal*, 7(1), pp. 308 - 313, 1965.

[13] Bäck, T., *Evolutionary Algorithms in Theory and Practice*, Oxford Univ. Press, New York, 1996.

[14] K. Schittkowski, *QLD : A FORTRAN Code for Quadratic Programming, User's Guide*, Mathematisches Institut, Universität Bayreuth, Germany, 1986.

[15] Davis, L., ed., *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, New York, 1991.

[16] Chester, D.L., "Why Two Hidden Layers are Better than One", *proc. of the Int. Joint Conference on Neural Networks*, IJCNN-90-WASH-DC, 1, Washington DC, USA, INNS-IEEE, Jan. 15-19, 1990, pp. 265 - 268.

[17] Lippmann, R.P., "An Introduction to Computing with Neural Networks", *IEEE ASSP magazine*, April 1987, pp. 5 - 22.

[18] Whitley, D., "Applying Genetic Algorithms to Neural Network Learning", *Proc. of the 7th Conf. of the Society of Artificial Intelligence and Simulation of Behaviour*, Sussex, England, Pitman Publishing, 1989, pp. 137 - 144.

[19] *Numerical Recipees in C*

[20] "Military Specifications, Flying Qualities of Piloted Aircraft", MIL-F-8785C, Dept. of Defense, USA, Nov. 1980.

[21] Etkin, B., *Dynamics of Atmospheric Flight*, Wiley, 1972.

[22] Shinozuka, M., "Response to Earthquakes Creep and Fatigue Failure ", Session IV G. Lecture Notes, ed. N.C. Nigam, Probabilistic Methods in Engineering, IIT Kanpur, 1972.

8 Appendix: More Comments on the Generalized Neural Network Generator

We present here comments on the possibilities and limitations of the neural network generator.

1. This neural network generator allows the user to chain neurons in any fashion, including all possible reccurences. This opens the possibility, for example, of running networks in parallel, as sketched in Figure 30.

2. Lists of neurons (`**list_of_neurons`) permit evaluation of neurons by groups, which helps in complex topologies with reccurent loops. The same neuron can be evaluated many times, either within the same list, or through different lists. Two uses of this feature are shown in Figure 31, one for recurrent networks, the other for probabilistic neurons that do not behave twice the same. There is not interference between multiple evaluation of a neuron and gradient calculation through backpropagation of the error. The sensitivity of the reccurent neuron is just evaluated as many times as the neuron is evaluated.

3. Gradient calculation is indeed not implemented yet for recurrent networks. However, from a program implementation point of view, the only thing that is missing is a reasonable guess for the initial values of `dnetout_dout`. Because of recurrent networks, it is necessary that the initial neuron output value, `out` be initialized to 0.

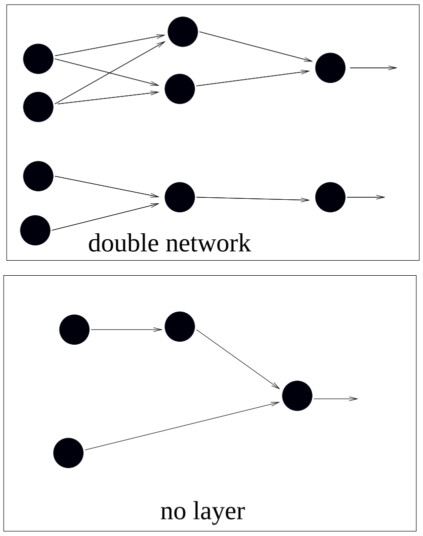

Figure 30: Examples of possible network topologies

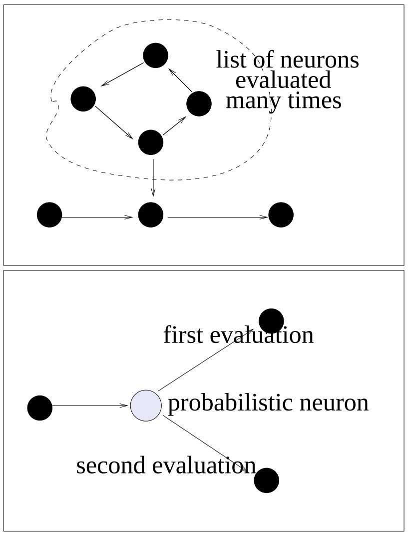

Figure 31: Utility of multiple evaluations of a Neuron